Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture

Document number	DDI0606
Document version	A.k
Document confidentiality	Non-confidential

Important message

Morello is a prototype architecture, which has a particular meaning to Arm of which the recipient must be aware as follows:

Subject to change without consent of all parties, and it is not committed for product development.

Includes the majority of expected features.

Includes detail on the majority of expected features.

Includes some necessary information from documentation relating to earlier architectures, but some cross-referencing might be necessary.

See the architecture release notes for more detail.

No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.

Release information

Date	Version	Changes
2022/Jan/17	A.k	• PROTO_REL 04 • PROTO_REL 04 external release
2021/Jun/25	A.j	• PROTO_REL 03 • PROTO_REL 03 external release
2021/Mar/19	A.i	• PROTO_REL 02 • PROTO_REL 02 external release
2020/Dec/18	A.h	• PROTO_REL 01 • PROTO_REL 01 external release
2020/Oct/28	A.g	• PROTO_REL 00 • PROTO_REL 00 external release • CHERI reference updated to version 8
2020/Sep/30	A.f	• PROTO_REL 00 • PROTO_REL 00 external release
2020/Aug/13	A.e	• PROTO_EAC 01 • PROTO_EAC 01 release, limited circulation
2020/Jul/02	A.d	• PROTO_EAC 00 • PROTO_EAC release, limited circulation
2020/May/29	A.c	• Beta 02 • Beta release, limited circulation
2020/Apr/09	A.b	• Beta 01 • Beta release, limited circulation
2020/Mar/25	A.b	• Beta 01 RC • Beta release candidate, limited circulation
2020/Jan/20	Beta 00	• Beta draft, limited circulation
2019/Dec/09	Alpha 01	• First draft for review

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349 version 21.0

Product Status

The information in this document is for a prototype extension to the Armv8-A architecture.

Changes in PROTO_REL_04

[1633]
The Embedded Trace Macrocell Architecture Specification expects instructions to be classified in to direct branches, indirect branches, and not a branch. The Morello specification did not contain enough information to make this classification. This has been corrected by adding a new rule KHQMC.

[1623]
The ASL of the registers CDBGDTR_EL0 and CDLR_EL0 is corrected with respect to the trapping due to CPTR_EL2.CEN at EL2 and EL3.

[1618]
Rule R KDDZF step 10 references to B_ie[4] and T_ie[4] have been corrected to B_ie[3] and T_ie[3].

[1616]
In section 2.5.1 (Morello Bounds format), sub-section "Setting and encoding Bounds", the syntax used for SignExtend() when oE < 48 was incorrect, making it difficult to read. This has been corrected.

In section 2.5.2 (Representability checks), rule R LMXSB is clarified to describe the guaranteed range of the Capability Value, with respect to the base and the limit.

In section 2.8 (Capability memory relocation), rule R GFXBJ is clarified to apply the CCTLR_ELx.DDCBO field when executing in ELx.

[1594]
AArch64.SecondStageTranslate() is corrected for the case where the first stage of translation results in a Capability fault due to a store of a valid capability (CDBM == 0 && SC == 1). The second stage translation should not perform the hardware update if the second stage entry has CDBM == 1.

[1593]
The CheckLoadTagsPermission() function is corrected to use the Exception level for the translation regime, instead of the current Exception level.

[1592]
The code in MemAttrDefaults() that initialized fields related to the handling of the LC and SC bits has been removed, because generally these fields have been initialized before this function is called. The initialization code was missing in AArch64.TranslateAddressS1Off(), and so it has been added there.

[1590]
The ASL for MSR (immediate) is corrected to show that writes of SPSel are ignored when the PE is in Restricted.

[1589]
In R KDDZF, step 5 for bounds setting contains a wrong variable E, which has been corrected to E'.

[1585]
The pseudocode functions AArch64.SysInstrInputIsCapability() was incorrectly checking for DC IVAC using op1 == 3 and crm == 6, and has been corrected to check for op1 == 0 and crm == 6.

[1584]
The ASL functions AArch64.MemSingle(), AArch64.CapabilityTag(), MemAtomicCompareAndSwapC(), and MemAtomicC() are updated to correctly handle faulting due to the LC bit on Device memory.

[1582]
The text in section 2.5.1 describing how the Capability Bounds are decoded was corrected in order to match the related pseudocode in section 5. The specific correction is in the decoding of T[15:14].

[1580]
The description for BLRS (pair of capabilities) is corrected in removing the suggestion that this instruction would perform a switch to Restricted.

[1156]
In section 2.9, Compartment ID, the informational text state BYCZR is clarified to explain the intention with respect to how an implementation might use this information.

Known issues

[635]

The pseudo-instruction "MOV Cn,CZR", which maps to "MOV Xn, XZR", is not described in the instruction set.

[626]
The <extend> specifier on the following instructions is shown as a mandatory part of the syntax.
* ADD (extended register),
* Load/Store with a offset register.

This does not match the syntax for the equivalent instructions in the base architecture

Contents

Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture

Chapter 5

Pseudocode definitions

Preface

About this book

This book is the Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture. This book describes only the architectural changes that are introduced by Morello to the Armv8-A architecture. Therefore, this supplement must be read in conjunction with the specific version of *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile* listed in the Additional reading section of this supplement. Together, the manual and this supplement provide a full description of the Armv8-A architecture, including Morello functionality. For details about the base Armv8-A architecture, the *Arm® Architecture Reference Manual* is the definitive source of information.

It is assumed that the reader is familiar with the Armv8-A architecture.

Conventions

Typographical 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 for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the Glossary.

Colored text

Indicates a link. This can be:

- A URL, for example http://developer.arm.com
- A cross-reference to another location within the document
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term.

{ 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.

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.

Lower-case 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 book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book contains numerous syntax descriptions for assembler instructions and for components of assembler instructions. These are shown in a `monospace` font.

Rules-based writing

This specification consists of a set of individual rules. Each rule is clearly identified by the letter R.

Rules must not be read in isolation, and where more than one rule relating to a particular feature exists, individual rules are grouped into sections and subsections to provide the proper context. Where appropriate, these sections contain a short introduction to aid the reader. An implementation which is compliant with the architecture must conform to all of the rules in this specification.

Some architecture rules are accompanied by rationale statements which explain why the architecture was specified as it was. Rationale statements are identified by the letter X.

Some sections contain additional information and guidance that do not constitute rules. This information and guidance is provided purely as an aid to understanding the architecture. Information statements are clearly identified by the letter I.

Implementation notes are identified by the letter U.

Software usage descriptions are identified by the letter S.

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

Rules, rationale statements, information statements, implementation notes and software usage statements are collectively referred to as *content items*.

Identifiers

Each content item may have an associated identifier which is unique within the context of this specification.

When the document is prior to beta status:

- Content items are assigned numerical identifiers, in ascending order through the document (*0001*, *0002*, . . .).
- Identifiers are volatile: the identifier for a given content item may change between versions of the document.

After the document reaches beta status:

- Content items are assigned random alphabetical identifiers (*HJQS*, *PZWL*, . . .).
- Identifiers are preserved: a given content item has the same identifier across versions of the document.

Examples

Below are examples showing the appearance of each type of content item.

R This is a rule statement.

$\mathrm{R}_{\mathrm{X001}}$ This is a rule statement.

I This is an information statement.

X This is a rationale statement.

U This is an implementation note.

S This is a software usage description.

Additional reading

This section lists publications by Arm and by third parties.

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

Arm publications

- *Arm® Architecture Reference Manual, Armv8-A, for Armv8-A architecture profile* (ARM DDI 0487 F.c).
- *Arm® Embedded Trace Macrocell Architecture Specification, ETMv4.0 to ETMv4.5* (ARM IHI 0064 G.b).

Other publications

- Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Technical Report Number 951, Hardware Enhanced RISC Instructions: CHERI Instruction- Set Architecture (Version 8) , the University of Cambridge, Computer Laboratory, September 2020.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send your feedback using the Support Portal. Alternatively, send an e-mail to support-morello@arm.com. For us to address your comments accurately, provide the following information:

- The title (Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture).
- The number (DDI0606 A.k).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- 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.

We believe that this document contains no offensive terms. If you find offensive terms in this document, please contact terms@arm.com.

Chapter 1
Introduction

1.1 About the Morello architecture

The Morello architecture aims to improve the robustness and security of systems using the following design goals:

- Fine-grained memory protection leading to increased memory safety.
- Scalable compartmentalization.

To achieve these goals, the Morello architecture introduces the principles defined in the Technical Report Number 951, Hardware Enhanced RISC Instructions: CHERI Instruction- Set Architecture (Version 8) , including the principles of least privilege and intentional use. The Morello architecture is backwards compatible with and complementary to the existing Armv8-A architecture.

The CHERI model introduces *architectural capabilities*. Capabilities are tokens of authority that are unforgeable and delegable. In the CHERI model, they are integer values that have been extended with metadata to protect their integrity, limit how they are manipulated, and control their use.

This introduction summarizes the concept of capabilities by extracting content from Technical Report Number 951, Hardware Enhanced RISC Instructions: CHERI Instruction- Set Architecture (Version 8) . It also illustrates how the existing system incorporates the addition of capabilities, in order to benefit from the security features provided. The subsequent chapters expand this introduction in broadly two parts: the first part provides definition a conceptual of a new data type, the capability; the second part delineates expected hardware behavior in the context of the Armv8-A system. A list of registers that are changed by or added to the Morello architecture is added, followed by A64 and C64 instruction sets, as well as pseudocode for the functional description.

Arm acknowledges the contribution of the following named individuals and institutions in the derivation of the concepts within this architecture: Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel

Wesley Filardo, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven Murdoch, Kyndylan Nienhuis, Robert Norton, Alex Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia, the University of Cambridge, and SRI International.

The Morello architecture is based on concepts first described and developed in the Technical Report Number 927, Hardware Enhanced RISC Instructions: CHERI Instruction- Set Architecture , developed by the University of Cambridge and SRI International, with support from DARPA. In this supplement, some material from the Technical Report Number 927, Hardware Enhanced RISC Instructions: CHERI Instruction- Set Architecture has been extracted and modified. The incorporation of these concepts in Morello is in accordance with an existing agreement between Arm Limited and the Department of Computer Science and Technology, the University of Cambridge.

1.2 The CHERI protection model

A capability in the CHERI model consists of a value and the following additional metadata:

- Validity Tag: Providing integrity protection.
- Permissions: Limiting operations that can be performed.
- Bounds: Limiting how the value can be used, for example, for memory access.
- An object type: Supporting higher-level software encapsulation.

The CHERI model enforces several important security properties on changes to capability metadata:

- Provenance validity: Valid capabilities can only be constructed by instructions that do so explicitly, for example, from other valid capabilities.
- Capability monotonicity: Instructions cannot exceed the permissions and bounds of the original capability when creating valid capabilities, other than in controlled non-monotonicity, such as exception entry.

and a number of important security properties on sets of capabilities:

- Reachable capability monotonicity: In any execution of arbitrary code, until execution is yielded to another domain, the set of accessible capabilities cannot increase.
- Controlled non-monotonicity: Enables access to more capabilities on a control-flow transfer to a protected entry point.

Capabilities can be held in registers or in memory, and are accessed, manipulated, loaded, stored, and used as memory addresses by instructions that expect capability operands rather than integer values. The CHERI model adds new instructions to perform the following operations:

- Retrieving capability fields: Retrieves properties defined by capabilities, for example, a lower bound.
- Manipulating capability fields: Sets or modifies capability fields within the constraints of monotonicity.
- Loading or storing using capabilities: Loads or stores integer, capability, or other values using a suitably authorized capability.
- Controlling execution flow: Performs a branch or branch-and-link-register to a capability destination.
- Non-monotonic execution flow: Transferring control to a domain with a different set of accessible capabilities.

See also:

- *Technical Report Number 951, Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8)* listed in Other publications.

1.3 The Morello architecture in the Armv8-A profile

The Morello architecture extends the Armv8.2-A profile with features that implement the CHERI protection model. It implements 129-bit CHERI capabilities, simplified as capabilities in this supplement, with compressed bounds which provide a compromise between memory consumption and bounds precision.

The Morello architecture inherits the rules for architectural features and extensions from Armv8.2-A. This supplement describes changes to those rules, and defines any new features added.

The Morello Architecture is only supported in AArch64 state. An implementation supporting Morello does not support AArch32. To support the properties of the Morello architecture, some existing definitions of terms are modified.

See also:

- 2.3 *Changes to Armv8 terminology*

1.3.1 Capability registers and memory

General-purpose registers, certain System registers, and certain Special-purpose registers are extended to 129 bits to hold capabilities. A Program Counter Capability(PCC) extends the existing Program Counter(PC) to be a capability, providing validity, permission, bounds, and other checks on instruction fetch, along with some ambient permissions on certain classes of instructions.

1.3.2 Capability tagged memory

To prevent forgery, when a Capability is stored in memory, bit 128 of a capability, containing the Capability Tag, is stored in a separate location that is not accessible by normal load and store instructions. The other 128 bits of the capability are stored in regular memory locations.

See also:

- 2.2 *Capability registers*

1.3.3 ISA

The Morello Architecture is supported in AArch64 state. The A64 ISA is extended with instructions to manipulate, copy, and retrieve fields from capabilities. To a limited extent, the A64 ISA also allows using capabilities for instruction fetch and memory access. A variant of the A64 ISA, C64, is added to provide a richer set of instructions to use capabilities, at the expense of instructions using 64-bit values as address to access memory.

See also:

- Chapter 4 *Instruction definitions*

1.3.4 Controlled non-monotonicity

The Morello architecture provides the following methods for controlled non-monotonicity:

- Exception handling: The addition of capability exception handling registers enables access to new sets of capabilities via capability exception entry.

- Executive/Restricted: The PE can switch between two states, Executive and Restricted, on a capability branch or return. This option provides controlled access to a selection of capability registers within an Exception level.
- Unsealing operations: The operations allowing sealed capabilities to be unsealed for different purposes as defined by the Capability ObjectType field. Unsealing operations include the following operations:
  - Unseal pair of capabilities and branch.
  - Unseal using an unsealing capability.
  - Unseal, Load pair of capabilities and branch.
  - Check subset and unseal.
  - Unseal and branch.

See also:

- 2.6.2 *Controlled non-monotonic manipulation*

1.3.5 Capability memory protection

The Morello architecture provides an additional layer of memory protection, requiring that any access using a virtual address is checked implicitly or explicitly against a capability. Instructions using a capability as an address check every location accessed against that capability. Instructions not using a capability as an address, check every location accessed against the capability in a Default Data Capability (DDC).

For instruction fetch, and loads relative to the PC, the memory protection is provided by the capability in PCC.

See also:

- 2.7.2 *Capability memory protection*

1.3.6 Capability protection for System registers and instructions

Particularly at higher Exception levels, access to System registers and System instructions gives significant privilege. The Morello architecture provides a capability System permission which, when absent from the capability in PCC, prevents access to most System registers and System instructions.

See also:

- 2.7.1 *System permission*

1.3.7 Capability memory relocation

The Morello architecture adds controls to support a degree of relocation of capability-unaware code, and its access to data, within an address space, facilitating compartmentalization of that code.

See also:

- 2.8 *Capability memory relocation*

1.3.8 Recursive immutability

The Morello architecture introduces a capability mutability permission which, when absent from a capability used to load other capabilities, removes both write and mutability permission from any valid unsealed capability that is loaded.

This feature provides a recursive property on capabilities such that any memory reachable from an initial capability, other than via controlled non-monotonicity, can be made read-only.

See also:

- 2.7.4 *Recursive immutability*

1.3.9 The Virtual Memory System Architecture

The Morello architecture extends the virtual memory system with new permissions in page table entries to control access to capabilities in memory, and also to track the writing of capabilities to memory.

See also:

- 2.14 *The Virtual Memory System Architecture*

1.3.10 Debug and trace

The external debug architecture is extended to allow both capability-aware and capability-unaware debuggers.

Performance monitoring events are added monitor Morello specific architectural and micro-architectural behavior.

The Statistical Profiling Extension is extended to track loads and stores of capabilities.

See also:

- 2.16 *The Embedded Trace Macrocell architecture*
- 2.17 *Performance Monitoring Unit*
- 2.19 *External debug*

1.4 The Morello architecture features

The Morello architecture is an extension to the Armv8-A architecture version Armv8.2-A.

An implementation of the Morello architecture includes all of the mandatory Armv8.2-A features, and the following optional features:

- FEAT_FP16, Half-precision floating-point data processing.
- FEAT_DotProd, SIMD Dot Product.
- FEAT_HPDS2, Translation table page-based hardware attributes.
- FEAT_LVA, Large VA support.
- FEAT_IESB, Implicit error synchronization event.
- FEAT_EVT, Enhanced Virtualization Traps.

In addition to the Armv8.2-A extension, a Morello implementation includes the following additional features:

- The Statistical Profiling Extension.
- FEAT_LRCPC, Load-acquire RCpc instructions.
- FEAT_SSBS, Speculative Store Bypass Safe.

Other features defined in the Arm architecture after Armv8.2-A are not supported in the Morello architecture.

An implementation of the Morello architecture does not support the following:

- The AArch32 state.
- Mixed-endian at any Exception level.
- Fixed big-endian. The architecture only supports fixed little-endian.

The feature names have been changed in the *Arm® Architecture Reference Manual, Armv8-A* and this document uses the feature names updated in the *Arm® Architecture Reference Manual, Armv8-A* listed in Arm publications. A mapping between the legacy feature names and new names has been provided.

See also:

- Appendix K13, *Legacy Feature Naming Convention*, *Arm® Architecture Reference Manual, Armv8-A*: Mapping of the legacy feature names for the Armv8.x extensions.

Chapter 2
Capability architecture rules

2.1 Capabilities

R_{GGSXN} A capability is a composite data type with the following fields:

Name	Description
Value	Provides values used in capability-based operations.
Bounds	Limits how the Capability Value can be used.
Permissions	Limits how the capability can be used.
ObjectType	Determines whether a capability is sealed and, for a sealed capability, how the capability is sealed.
Global	Restricts the locations where a capability can be stored.
Executive	Controls banking of certain System registers.
Flags	Holds unrestricted user data.
Tag	Defines the validity of a capability.

R_{GKNXV} The Capability Value is 64 bits.

R_{VHRRV} The Capability Value can be accessed as one of the following:

- An absolute value.
- An offset from the bounds base defined by the Capability Bounds.

I$_{HTNXS}$ The Capability Bounds define a 65-bit upper and 64-bit lower bound, depending on how a capability is used.

R$_{YSBDT}$ The Capability Tag defines the validity of a capability in one of the following ways:

- If the Capability Tag is 1, the capability is valid.
- If the Capability Tag is 0, the capability is invalid.

R$_{KFRHT}$ The Capability Permissions contain all of the following permission controls:

Name	Permission
Load	Load from memory
Store	Store to memory
Execute	Execute instructions
LoadCap	Load a valid capabilty to a Capability register
StoreCap	Store a valid capabilty from a Capability register
StoreLocalCap	Store a Local capability to memory
Seal	Seal an unsealed capability
Unseal	Unseal a sealed capability
System	Access System registers and instructions
BranchSealedPair	Use in an unsealing branch
CompartmentID	Use as a compartment ID
MutableLoad	Load to a Capability register with mutable permissions
User[N]	Software defined permissions

R$_{VYQWL}$ A capability is either sealed or unsealed.

R$_{RVFDY}$ The ObjectType of a capability determines if that capability is sealed:

- If the ObjectType of a capability is 0, the capability is unsealed.
- If the ObjectType of a capability is nonzero, the capability is sealed.

2.2 Capability registers

R_{BVJJF}

R$_{\text{BVJJF}}$ The Morello architecture introduces the term “Capability register” to define a register that can hold a capability.

R$_{\text{NWDGC}}$ Capability registers are 129 bits.

R$_{\text{YXLPL}}$ When Morello is implemented, general-purpose registers, some System registers, and some Special-purpose registers, are extended to be Capability registers.

R$_{\text{PMLYT}}$ Capability registers can have the following access views:

- 129-bit: the Capability access view.
- 64-bit.
- 32-bit.

R$_{\text{JXNGH}}$ The following table provides an overview of general-purpose registers when the Morello architecture is implemented:

General-purpose register name (n=0-30)	Access view provided (bits)	Register names based on access view (n=0-30)
Rn	64	Xn
	32	Wn
	129	Cn

In a general-purpose register field, the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position, as summarized in the following tables:

Access view provided (bits)	Register names based on access view
64	SP
32	WSP
129	CSP

Register size (bits)	Register names based on size accessed
64	XZR
32	WZR
129	CZR

I$_{\text{TSPCV}}$ The Morello architecture adds a set of Default Data Capability registers:

- DDC_EL0.
- DDC_EL1.
- DDC_EL2.
- DDC_EL3.
- RDDC_EL0.

The mnemonic DDC is used as an accessor to refer to the current (R)DDC_ELx register based on other contexts and settings.

$\mathrm{I}_{\mathrm{HXBKV}}$ The Program Counter (PC) is extended to be a Program Counter Capability register (PCC).

$\mathrm{R}_{\mathrm{VJSVC}}$ No explicit synchronization is required between accessing a System register using different access views.

$\mathrm{R}_{\mathrm{RWCXN}}$ When writing to a register using an access view narrower than the maximum access view, the upper bits, including the Capability Tag, of the register are set to 0.

See also:

- Chapter B1.2, *Registers in AArch64 Execution state*, *Arm® Architecture Reference Manual, Armv8-A*: more details about Armv8-A registers.
- Chapter C5.1.5, *op0==0b11, Moves to and from Special-purpose registers*, *Arm® Architecture Reference Manual, Armv8-A*: more details about special-purpose registers.

2.3 Changes to Armv8 terminology

$\mathrm{R}_{\mathtt{TRWTV}}$ If an UNPREDICTABLE operation writes a capability register, the write does not increase the set of reachable capabilities.

$\mathrm{R}_{\mathtt{TSNJF}}$ If an UNKNOWN value is written to a capability register or to capability-tagged memory, the write does not increase the Capability defined rights available to software.

2.4 Capabilities in memory

R_{MPSCL} The Morello architecture introduces capability tag locations, separate to byte locations.

R_{RBKYF} A capability-tagged location is a byte location associated with a capability tag location.

R_{PSBDR} The set of 16 contiguous capability-tagged locations starting at a 16-byte aligned address is associated with the same distinct Capability Tag.

I_{JYMJV} In a system implementing the Morello architecture extension, all byte locations in general-purpose memory are capability-tagged locations.

R_{BYQDV} The lower 128 bits of a capability in memory are in little-endian byte order.

R_{DHDNX} A capability store to a 16-byte aligned address, N, atomically stores the following:

- The lower 128 bits of the capability to the 16 byte locations starting at N.
- The Capability Tag to the capability tag location associated with those byte locations.

R_{RVNCT} A capability load from a 16-byte aligned address, N, atomically loads the following:

- The lower 128 bits of the capability from the 16 byte locations starting at N.
- The Capability Tag from the capability tag location associated with those byte locations.

R_{VRTNV} If a capability store is not to a 16-byte aligned address, the store generates an alignment fault.

R_{WQKKP} If a capability load is not from a 16-byte aligned address, the load generates an alignment fault.

R_{HGFYZ} A non-capability store to a capability-tagged location atomically writes the capability tag location associated with that capability-tagged location to 0.

R_{DYYBT} If a capability is written to a non-capability-tagged location, it is IMPLEMENTATION DEFINED which of the following applies:

- The byte locations are written and the Capability Tag is ignored.
- The byte locations become UNKNOWN and the Capability Tag is ignored.
- An External abort is generated.

R_{PKWPL} If a capability is read from a non-capability-tagged location, it is IMPLEMENTATION DEFINED which one of the following applies:

- The byte locations are read and the Capability Tag is read as 0.
- The destination Capability register becomes UNKNOWN.
- An External abort is generated.

R_{DLCPG} For a non-capability atomic operation writing to a byte location associated with a capability tag location, if the operation does not change the value in the byte location, it is IMPLEMENTATION DEFINED whether the capability tag location is written to 0.

See also:

- Chapter B2.3.1 *Basic definitions*, *Arm® Architecture Reference Manual*: Definition of byte location.
- Chapter B2.3 *Definition of the Armv8 memory model*, *Arm® Architecture Reference Manual*: Introduction to the concept of locations in Armv8-A architecture.

2.5 Capability encoding

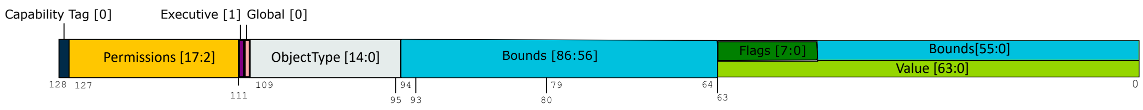
I_{PQHKQ} The Morello Capability format is similar but not identical to the CHERI-concentrate format.

R_{HRVBQ} A Capability value comprises the following fields:

- Value: 64 bits.
- Bounds: 87 bits.
- Flags: 8 bits.
- ObjectType: 15 bits.
- Permissions: 16 bits.
- Tag: 1 bit.
- Global: 1 bit.
- Executive: 1 bit.

The Flags and the lower 56 bits of the Capability Bounds share encoding with the Capability Value.

R_{ZLYBF}



For the encoding of a capability, the following fields are encoded together:

- Global [0].
- Executive [1].
- Permissions [17:2]

The Permissions field [17:2] is encoded as the following:

Bits	Permission
17	Load
16	Store
15	Execute
14	LoadCap
13	StoreCap
12	StoreLocalCap
11	Seal
10	Unseal
9	System
8	BranchSealedPair
7	CompartmentID

Bits	Permission
6	MutableLoad
5:2	User[4]

See also:

- CHERI Instruction-Set Architecture.

2.5.1 Morello Bounds format

I_{DWRPY} The 87 bits of Capability Bounds can be accessed as one of the following:

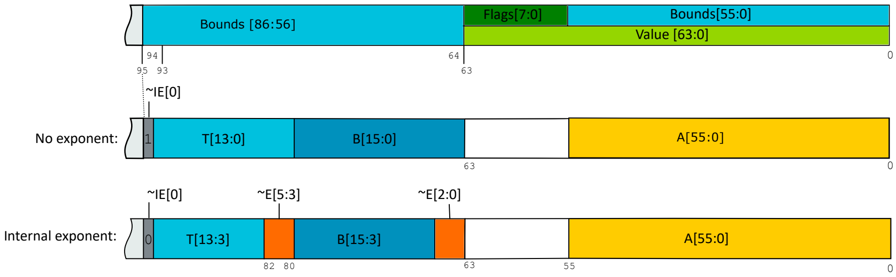
- A base, b, and limit, t.
- A base and length, l.

For the base, limit, and length of bounds, all of the following are true:

- Base is a 64-bit quantity.
- Limit is a 65-bit quantity.
- Length is a 65-bit quantity.

R_{XKVDF} The Bounds field encodes the following 5 values used to encode and decode the base and limit of a capability:

Element	Description
Bottom(B)	16-bit quantity used to derive the base.
The Internal Exponent(IE)	The value of IE determines if E is encoded in the bounds or treated as 0: • When IE is 0: E is treated as 0. • When IE is 1: E is encoded in the lower bits of T and B. This bit is stored inverted.
Top(T)	A 16-bit quantity used to derive the limit. T[15:14] are encoded using B, IE, and the other bits of T.
The Exponent(E)	A 6-bit quantity that determines the position at which B and T are inserted into A to recover base and limit. E is stored inverted.
A	A 66-bit value used to define the base and limit when E<48. Bits [55:0] are encoded in Bounds, the other bits are derived from A[55] or are set to 0.



R_{SFKZW} A, B, T, E, and IE are decoded in the following ways:

- A is derived using the following:

  $A[65:64] = 0$

  $A[63:0] = SignExtend(Value[55:0], 64)$

- IE is derived using the following:

  $IE = {\sim}Bounds[86]$

- E is derived using the following:

  $$E[5:0] = \begin{cases} 0, & \text{if } IE == 0 \\ {\sim}Bounds[74:72] : {\sim}Bounds[58:56], & \text{if } IE == 1 \end{cases}$$

- The T and B values are decoded as follows:

  $B[15:3] = Bounds[71:59]$

  $$B[2:0] = \begin{cases} Bounds[58:56], & \text{if } IE == 0 \\ 0, & \text{if } IE == 1 \end{cases}$$

  $T[13:3] = Bounds[85:75]$

  $$T[2:0] = \begin{cases} Bounds[74:72], & \text{if } IE == 0 \\ 0, & \text{if } IE == 1 \end{cases}$$

  T[15:14] is decoded as follows:

  $$T[15:14] = \begin{cases} B[15:14], & \text{if } (T[13:0] \geq B[13:0]) \wedge (IE == 0) \\ B[15:14] + 1, & \text{if } (T[13:0] < B[13:0]) \wedge (IE == 0) \\ B[15:14] + 1, & \text{if } (T[13:3] \geq B[13:3]) \wedge (IE == 1) \\ B[15:14] + 2, & \text{if } (T[13:3] < B[13:3]) \wedge (IE == 1) \end{cases}$$

R_{DJZDW} A, B, T, E, and IE are encoded into the Capability Bounds field as the following:

$Bounds[86] = \sim IE$

$Bounds[85:75] = T[13:3]$

$$Bounds[74:72] = \begin{cases} T[2:0], & \text{if } IE == 0 \\ {\sim}E[5:3], & \text{if } IE == 1 \end{cases}$$

$Bounds[71:59] = B[15:3]$

$$Bounds[58:56] = \begin{cases} B[2:0], & \text{if } IE == 0 \\ {\sim}E[2:0], & \text{if } IE == 1 \end{cases}$$

$Bounds[55:0] = A[55:0]$

$\mathrm{I}_{\mathrm{CKZPG}}$ The Capability Bounds are valid or invalid.

$\mathrm{R}_{\mathrm{FPZNM}}$ If any of the following is true, the Capability Bounds are valid:

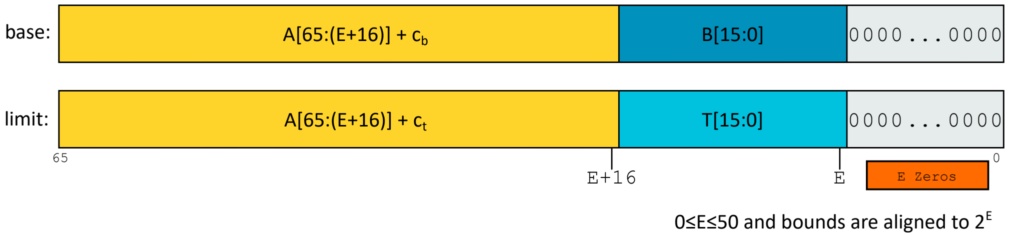
- The value of the Exponent equals to 63.
- The value of the Exponent is less than 51.

Otherwise, the Capability Bounds are invalid.

Decoding Bounds

$\mathrm{R}_{\mathrm{GZYKG}}$

1. The Capability Bounds field is decoded to the Capability Base, base, and the Capability Limit, limit. Base and limit are derived from A, B, T, and E. Base is a 64-bit value. Limit is a 65-bit value.
   - If E == 63:
     - base = 0
     - limit = 2^{64}
     - The Capability Bounds are valid.
   - If $51 \leq$ E ≤ 62:
     - base = 0
     - limit = 2^{64}
     - The Capability Bounds are invalid.
   - If E<51:
     - $base[65:0] = (A[65:(E+16)] + C_b) : B[15:0] : Zeros(E)$
     - $limit[65:0] = (A[65:(E+16)] + C_t) : T[15:0] : Zeros(E)$
     - The Capability Bounds are valid.



The upper regions of base and limit (those derived from A) are subject to a correction factor of +/- 1, where C_b and C_t are derived using the following:

$A3 = A[E+15 : E+13]$

$$B3 = B[15:13]$$

$$T3 = T[15:13]$$

$$R3 = B3 - 0b001$$

$$aHi = \begin{cases} 1, & \text{if } A3 < R3 \\ 0, & \text{otherwise} \end{cases}$$

$$bHi = \begin{cases} 1, & \text{if } B3 < R3 \\ 0, & \text{otherwise} \end{cases}$$

$$tHi = \begin{cases} 1, & \text{if } T3 < R3 \\ 0, & \text{otherwise} \end{cases}$$

$$C_b = bHi - aHi$$

$$C_t = tHi - aHi$$

2. The base and limit are generated as follows:

$$base[65:0] = (A[65:(E+16)] + C_b) : B[15:0] : Zeros(E)$$

$$limit[65:0] = (A[65:(E+16)] + C_t) : T[15:0] : Zeros(E)$$

Setting and encoding Bounds

R_{KDDZF}

Bounds setting uses a Capability Value, Value, and an Exponent, oE, to derive a requested base, nb, along with a requested length, nl, to derive a requested limit, nt. The requested base and limit are used to generate A, B, T, E, and IE fields, to be encoded in a Capability Bounds field.

The encoded A, B, T, E, and IE are generated as follows:

1. Calculate the requested base, nb:

$$nb[65:64] = 0$$

$$nb[63:0] = \begin{cases} SignExtend(Value[55:0], 64), & \text{if } oE < 48 \\ Value[63:0], & \text{otherwise} \end{cases}$$

2. Calculate the requested limit, nt:

$$nt[65:0] = nb[65:0] + 0 : nl[64:0]$$

3. Calculate A:

$$A = SignExtend(Value[55:0], 66)$$

4. Calculate a candidate exponent, E':

$$E' = 50 - CountLeadingZeroes(nl[64:15])$$

Lengths less than 2^{15} are encoded with $E' == 0$

5. Calculate IE:

$$IE = \begin{cases} 0, & \text{if } (E' == 0) \wedge (nl[14] == 0) \\ 1, & \text{otherwise} \end{cases}$$

6. Calculate a candidate Bottom, B_ne, and a candidate Top, T_ne, for the no internal exponent encoding:

$$B\_ne[15:0] = nb[15:0]$$

$$T\_ne[15:0] = nt[15:0]$$

7. Calculate a candidate Bottom, B_ie, and a candidate Top, T_ie, for the internal exponent encoding:

$$B\_ie[15:0] = nb[E'+15:E'+3] : 000$$

$$T\_ie[15:0] = nt[E'+15:E'+3]:000$$

8. Calculate rounded base and rounded limit to check whether rounding is required on the new base and limit in the internal exponent encoding, and a new candidate top that is rounded up, not down:

$$rounded\_base = nb[E'+2:0] \neq 0$$

$$rounded\_limit = nt[E'+2:0] \neq 0$$

$$T\_ie' = \begin{cases} T\_ie + 8, & \text{if } rounded\_limit \\ T\_ie, & \text{otherwise} \end{cases}$$

9. Calculate a new candidate exponent, E”, for the internal exponent encoding, increased by 1 if the candidate Top has the top bit set:

$$adjust\_E = T\_ie' - B\_ie \geq 2^{15}$$

$$E'' = \begin{cases} E'+1, & \text{if } adjust\_E \\ E', & \text{otherwise} \end{cases}$$

10. Calculate a new candidate Top, T_ie”, a new candidate Bottom, B_ie’, rounded_base, and rounded_limit, based on whether E was adjusted. Again ensure that the candidate Top is rounded up, not down:

$$T\_ie'' = \begin{cases} nt[E''+15:E''+3]:000, & \text{if } adjust\_E \\ T\_ie', & \text{otherwise} \end{cases}$$

$$B\_ie' = \begin{cases} nb[E''+15:E''+3]:000, & \text{if } adjust\_E \\ B\_ie, & \text{otherwise} \end{cases}$$

$$rounded\_base' = \begin{cases} True, & \text{if } (adjust\_E) \wedge (B\_ie[3] == 1) \\ rounded\_base, & \text{otherwise} \end{cases}$$

$$rounded\_limit' = \begin{cases} True, & \text{if } (adjust\_E) \wedge (T\_ie'[3] == 1) \\ rounded\_limit, & \text{otherwise} \end{cases}$$

$$T\_ie''' = \begin{cases} T\_ie''+8, & \text{if } (adjust\_E) \wedge (rounded\_limit') \\ T\_ie'', & \text{otherwise} \end{cases}$$

11. Select the appropriate candidate T, B, and E:

$$E = \begin{cases} E'', & \text{if } IE == 1 \\ E', & \text{otherwise} \end{cases}$$

$$T = \begin{cases} T\_ie''', & \text{if } IE == 1 \\ T\_ne, & \text{otherwise} \end{cases}$$

$$B = \begin{cases} B\_ie', & \text{if } IE == 1 \\ B\_ne, & \text{otherwise} \end{cases}$$

12. Calculate whether the Capability Bounds were encoded exactly:

$$inexact = \begin{cases} rounded\_base' \vee rounded\_limit', & \text{if } IE == 1 \\ False, & \text{otherwise} \end{cases}$$

A, T, B, E, and IE are then encoded in a Capability Bounds field as described in $\mathrm{R}_{\mathrm{DJZDW}}$.

$\mathrm{R}_{\mathrm{STDNY}}$ If any of the following are true, the Bounds are considered invalid:

- The request was for exact bounds and the encoded bounds are inexact.
- The requested base is lower than the original base.

- The requested limit is above the original limit.

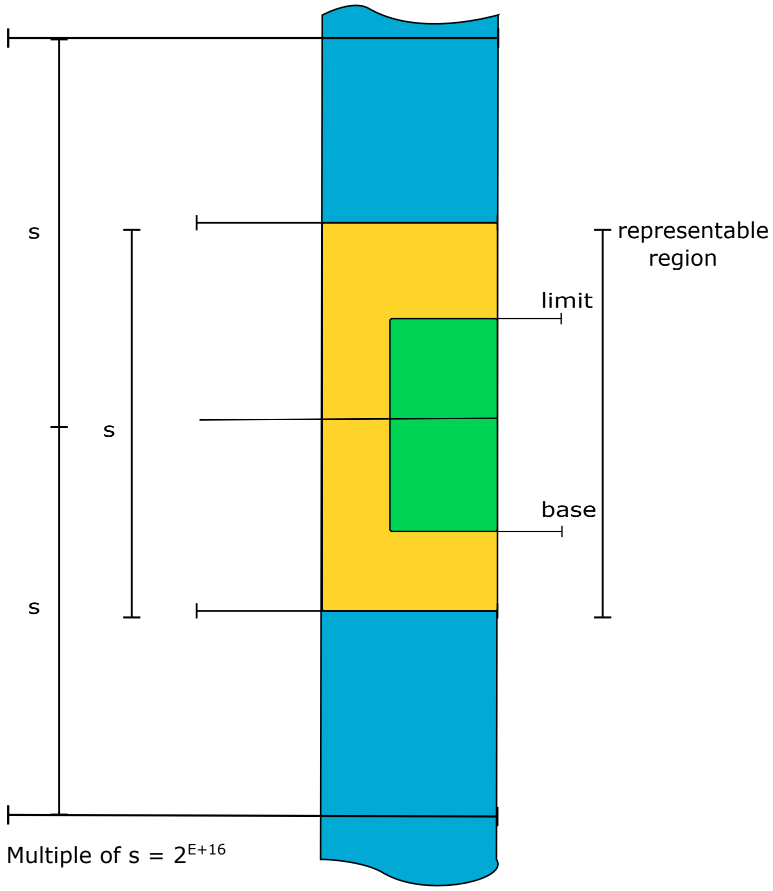
I_{QKCRL} If all of the following are true, the bounds are guaranteed to be exactly representable:

- `(nb)AND (NOT nlMask)== 0`: Where nb is the requested base and nlMask is the value returned by the `RRMASK` instruction when passed the requested length, nl, as its source.
- `nl == Rnl`: Where nl is the requested length and Rnl is the value returned by the `RRLEN` instruction when passed the requested length, nl, as its source.

2.5.2 Representability checks

R_{CYMZJ} Not all combinations of Capability base, limit, and Value are representable. When modifying a Capability Value field, an operation may cause the Capability Bounds to change, and the encode base and limit to become unrepresentable. If the modification causes the base and limit to become unrepresentable, the Capability Tag is set to 0.

The concept of the representability of capabilities:



The representable region covers the range of values that are between 12.5% below the base and 25% above the limit.

Note: Not all capabilities with large bounds have a contiguous representable region.

R_{JXHKF} A representability check is applied when manipulating a Capability Value.

R_{LCBNH} If modifying a Capability Value causes the base or limit to change, a representabilty check fails. Some versions of the check may fail in additional cases.

R_{LMXSB} If the Capability Value is modified, the encoding of the capability allows the Capability Value to change within the representable region, otherwise the base and limit may become unrepresentable.

R`BYTMV` If modifying a capability causes a representability check to fail, the Capability Tag on the generated capability is set to 0.

I`SMYZK` The Representable check has two versions: “full” and “fast”. The full check confirms that the Capability Bounds are unchanged by a change in Capability Value. The fast check determines whether incrementing the Capability Value leads to it being unrepresentable. In some cases the fast check returns a false negative result, but never returns a false positive result.

R`SVFVW` None of following operations can make Capability Bounds unrepresentable:

- Modifying the Capability Flags field directly.
- Modifying the Capability Flags field indirectly by modifying the Capability Value.

Fast Representability Check

R`YJVDC` The Fast representability check uses the following elements:

- An increment, I, modified by sign extending from bit 55
- The E field encoded in the Capability Bounds.
- The A field encoded in the Capability Bounds.

The Fast representability check comprises the following tests:

1. `BigExp`:

   If the Exponent is large enough, the Capability Value is not used to reconstruct base and limit:

   $BigExp == E \geq 48$

2. `InRange`:

   If the absolute value of the increment is larger than the Representable range, s, the result is not representable.

   $InRange = (I[63 : E + 16] == -1) \vee (I[63 : E + 16] == 0)$

3. `InLimit`:

   A Representable limit, R, is defined as the following:

   $R[15 : 13] = B[15 : 13] - 1$

   $R[12 : 0] = 0$

   Then a comparison is made depending on sign of the increment, as follows:

   $$InLimit = \begin{cases} I[E + 15 : E] < R[15 : 0] - A[E + 15 : E] - 1, & \text{if } I \geq 0 \\ (I[E + 15 : E] \geq R[15 : 0] - A[E + 15 : E]) \wedge (R[15 : 0] \neq A[E + 15 : E]), & \text{otherwise} \end{cases}$$

4. `FixedMSBVal`:

   If E < 48, A is used to form the base and A must not change sign:

   $FixedMSBVal = (A[55] == (A + I)[55])$

The Fast Representability check combines the four tests as:

$FastRep = (InRange \wedge InLimit \wedge FixedMSBVal) \vee BigExp$

2.6 Manipulating capabilities

R_{HRBLB} Manipulating a capability is defined as copying a capability, possibly changing the value of capability fields of the copy.

R_{PLJJR} A valid capability can only be created by one of the following:

- Monotonic manipulation.
- Controlled non-monotonic manipulation.

R_{FLXBS} Monotonic manipulation includes the following operations:

- Modifying the Capability Value.
- Reducing the Capability Bounds.
- Reducing the Capability Permissions.
- Modifying the Capability Flags
- Sealing operations.

R_{LZSVB} Controlled non-monotonic manipulation includes the following operations:

- Unsealing a capability using an unsealing operation.
- Using a permitted, privileged capability creating instruction to mark a register or memory location as holding a valid capability.

R_{PXLGP} When a capability is manipulated, any of the following clears the Capability Tag:

- If the capability is sealed, an attempt to manipulate the capability other than using an unsealing operation.
- An attempt to increase the Capability Bounds.

R_{JGSBX} Sealing and then unsealing a capability does not increase the rights granted by that capability.

2.6.1 Monotonic manipulation: sealing operations

R_{MFMKV} Sealing a capability restricts its use to compatible unsealing operations.

R_{ZJHJX} A valid unsealed capability can be sealed by one of the following instructions:

- Sealing with a sealing capability:
  - SEAL (capability), Seal capability.
  - CSEAL, Conditionally Seal capability.
- Sealing with a branch with link instruction.
- Sealing without a capability:
  - SEAL (immediate), Seal capability (immediate).

R_{FJNDZ} For a sealing instruction that is not CSEAL, if any of conditions in a sealing operation fails, the Capability Tag of the source capability is cleared.

For CSEAL instruction, if any of conditions in a sealing operation fails, the source capability is written to the destination capability unchanged.

R_{DRTLX} If all of the following are true, SEAL (capability) and CSEAL generate a valid sealed capability:

- The unsealed capability is valid.
- For the sealing capability, all of the following are true:
  - The capability is valid.
  - The capability is unsealed.
  - The capability has the Seal permission.

  - The Capability Value is within the Capability Bounds.
  - The Capability Value is within the range of Capability ObjectType values.

R_{XXKYX} If a capability is sealed by SEAL (capability) or CSEAL, the ObjectType of the capability to be sealed is set to the sealing Capability Value.

R_{DXGLZ} If a branch with link instruction generates a sealed capability in C30, the sealed capability ObjectType is set to 1.

R_{GCQCJ} If all of the following are true, SEAL (immediate) generates a valid sealed capability:

- The capability to be sealed is unsealed.
- The capability to be sealed is valid.

R_{JBPWS} If a capability is sealed by SEAL (immediate), the sealed capability ObjectType is set to the value of the form field in the instruction encoding.

2.6.2 Controlled non-monotonic manipulation

Privileged capability creation:

R_{DBXPL} A privileged capability creating instruction is one of the following:

- Set the Capability Tag of a register: SCTAG.
- Store Capability Tags to memory: STCT.

I_{GSKXG} If CSCR_EL3.SETTAG is 0 and the PE is in an Exception level that is lower than EL3, a privileged capability creating instruction can not create a valid capability.

I_{PKDYL} If CHCR_EL2.SETTAG is 0 and the PE is in an Exception level that is lower than EL2, a privileged capability creating instruction can not create a valid capability.

R_{NZDTP} A privileged capability creating instruction is not permitted to create capabilities in EL0: the instruction is UNDEFINED in EL0.

See also:

- *2.4 Capabilities in memory*

Unsealing operations

R_{WTHDH} A valid sealed capability can only be used in a capability unsealing operation.

R_{YQZKX} A permitted unsealing operation on a valid sealed capability generates a valid unsealed capability.

R_{VNPYT} A non-permitted unsealing operation does one of the following:

- Clears the Capability Tag of the generated capability.
- Leaves the generated capability sealed.

R_{SXXQW} All of the following are unsealing operations:

- Unsealing with an unsealing capability, UNSEAL.
- Unsealing with a check subset, setting flags and conditionally unseal instruction, CHKSSU.
- A branch or return with a capability register as the target.
- A load capability pair and branch, LDPBR, using C29.
- A load and branch, BR (memory indirect), using C29.
- A branch to sealed capability pair.

R_{SXVWB} If all of the following are true, unsealing with an unsealing capability is a permitted unsealing operation:

- For the capability being unsealed, all of the following are true:
  - The capability is valid.
  - The capability is sealed.

- For the unsealing capability, all of the following are true:
  - The capability is valid.
  - The capability is unsealed.
  - The capability has the Unseal permission.
  - The Capability Value is within the Capability Bounds.
  - The Capability Value is within the range of Capability ObjectType values.
  - The Capability Value is equal to the ObjectType of the capability to be unsealed.

R_{JZTTZ} If the ObjectType of a capability is 1, the following are permitted unsealing operations:

- A branch operation using that capability as a target.
- A return to that capability.

R_{FLNXF} If all of the following are true, unsealing a sealed capability using a testing capability by a Check Subset, setting flags and conditionally unseal instruction, CHKSSU, is a permitted unsealing operation:

- The sealed capability is valid.
- The testing capability is valid.
- The testing capability is unsealed.
- The Capability Bounds of the sealed capability are a subset of Capability Bounds of the testing capability.
- The Capability Permissions of the sealed capability are a subset of the Capability Permissions of the testing capability.

R_{PKKBS} If all of the following are true, unsealing a capability using a Load Pair of capabilities and Branch instruction, LDPBR, is a permitted unsealing operation:

- The capability is valid.
- The capability is sealed.
- The capability ObjectType is 2.
- The destination capability register of the instruction is C29.

R_{FWMNR} If all of the following are true, unsealing a capability using an Unseal load and branch (immediate) instruction, BR (memory indirect), is a permitted unsealing operation:

- The capability is valid.
- The capability is sealed.
- The capability ObjectType is 3.
- The base capability register of the load and branch is C29.

R_{TZRYW} If all of the following are true, branch to sealed capability pair instruction with a first and a second capability is a permitted unsealing operation:

- The first and second capabilities are valid sealed capabilities.
- The first and second capabilities have BranchSealedPair permission.
- The first capability ObjectType is greater than 3.
- The ObjectType of the first and the second capabilities are the same.
- The first capability has Execute permission.
- The second capability does not have Execute permission.

Executive/Restricted banking

R_{NHGSJ} The Executive permission in PCC determines whether the PE is in Executive or Restricted:

- 0: The PE is in Restricted.
- 1: The PE is in Executive.

R_{MXBDJ} The combination of the Executive permission in PCC, PSTATE.SP, and the current Exception level, ELx, determines the registers selected to be accessed, as outlined in the following table:

Register mnemonic	Executive, when PSTATE.SP is 1	Executive, when PSTATE.SP is 0	Restricted, PSTATE.SP is treated as 0
DDC	DDC_ELx	DDC_EL0	RDDC_EL0
SP	SP_ELx	SP_EL0	RSP_EL0
TPIDR_ELx	TPIDR_ELx	TPIDR_ELx	RTPIDR_EL0

When a register is accessible using the register mnemonic listed in the register mnemonic column in the table above, accessing that register using other register mnemonics is UNDEFINED.

In Restricted, accessing the Executive registers is UNDEFINED.

R$_{YNLZF}$ Transition from Executive to Restricted is only permitted in one of the following ways:

- A branch (restricted) instruction, BRR, BLRR.
- A Return from subroutine with possible switch to Restricted, RETR.
- Capability exception return.
- Capability exception entry.

I$_{QXPDW}$ When the PE is in Restricted, branch (restricted) instructions are UNDEFINED.

I$_{JGDJF}$ If a transition from Executive to Restricted is not permitted, the Capability Tag of PCC is cleared.

R$_{GNBDH}$ Transition from Restricted to Executive is only permitted in one of the following ways:

- A branch instruction that meets all of the following conditions:
  - The target of the instruction is a capability.
  - The instruction is not a branch (restricted) instruction.
- Capability exception return.
- Capability exception entry.

R$_{VMGDS}$ For a PE in Restricted, RDDC_EL0 is used as the current DDC for loads and stores.

R$_{RGKSN}$ For a PE in Restricted, SPSel is RAZ/WI.

R$_{QFFSK}$ For a PE in Executive in ELx, if PSTATE.SP is 1, DDC_ELx is used as the current DDC for loads and stores in ELx.

R$_{LHLFL}$ For a PE in Executive in ELx, if PSTATE.SP is 0, DDC_EL0 is used as the current DDC for loads and stores.

2.7 Using capabilities

R_{VBQMJ} Using a capability is defined as performing an operation that relies on the rights granted by that capability.

R_{DHFGV} A capability-restricted resource is one of the following:

- A virtual memory location.
- A System register.
- A System instruction.

2.7.1 System permission

R_{CRYKT} The System permission bit in PCC determines whether access to capability-restricted System registers and instructions is permitted:

- When the System permission of PCC is 1, System permission is enabled.
- When the System permission bit of PCC is 0, System permission is disabled and the MRS and MSR instruction access to System registers is limited in the following ways:
  - 64-bit MRS and MSR instruction access to System registers is limited to the following register mnemonics only:
    - TPIDR_ELx.
    - RTPIDR_EL0.
    - TPIDRRO_EL0.
    - DCZID_EL0.
    - CTR_EL0.
    - CNTVCT_EL0, unless CCTLR_ELx.PERMVCT for the current Exception level is 0.
  - Capability MRS and MSR instruction access to System registers is limited to the following register mnemonics only:
    - CTPIDR_ELx.
    - RCTPIDR_EL0.
    - CTPIDRRO_EL0.
    - CID_EL0.

R_{SKQFC} If MRS and MSR instructions are used to access System registers without the required System permission, a trap is generated based on the access view used:

- For 64-bit MRS and MSR instructions, the access generates a Trapped MSR, MRS, or System instruction execution in AArch64 state exception.
- For capability MRS and MSR instructions, the access generates a Trapped capability MSR or MRS instruction execution exception.

R_{NZSZL} Access to Special-purpose registers is not restricted by System permission.

R_{WRJDH} If the System permission of PCC is 0, it is IMPLEMENTATION DEFINED which IMPLEMENTATION DEFINED System registers and System instructions are trapped.

I_{BCGWP} In the condition mentioned in R_{WRJDH}, it is expected that most, if not all, IMPLEMENTATION DEFINED System registers and instructions are trapped.

R_{DFMNS} If the System permission of PCC is 0, any of the following generate a Trapped MSR, MRS, or System instruction execution in AArch64 state exception:

- Data cache operations, other than operations by VA.
- Instruction cache operations, other than operations by VA.
- TLBI operations.

- AT operations.

R_{BFSBQ} If the System permission of PCC is 0, all of the following are true:

- ERET causes the Capability Tag on the capability written to PCC to be cleared.
- SCTAG does not set the Capability Tag on the destination register.
- STCT treats the Capability Value in the transfer register as 0.

I_{DMBKP} The behavior of SVC, HVC, and SMC are not affected by System permission.

2.7.2 Capability memory protection

R_{GMYFJ} Every access to a memory location using a VA is restricted by a capability.

R_{CLBHX} If a load, store, or cache maintenance by VA instruction uses a capability base register, all of the following are true:

- The instruction uses the Capability Value of that capability base register as the base address for the operation.
- Memory locations accessed by the instruction are restricted by that capability base register.

I_{NRTCV} Following R_{CLBHX}, the full 64 bits of the Capability Value, including the Capability Flags, is used as the base address. To avoid an address size fault, software must ensure one of the following:

- The Capability Flags are canonicalized before using these bits in a memory access instruction.
- The MMU is configured to ignore bits [63:56] of the address.

R_{QWGWC} For a load, store, or cache maintenance by VA instruction using a 64-bit base register, memory locations accessed by the instruction are restricted by the capability in the current DDC.

R_{KBMFJ} For the purpose of Capability memory protection, the STCT instruction is treated as a store of capabilities.

R_{BLNHL} For the purpose of Capability memory protection, the LDCT instruction is treated as a load of capabilities.

R_{TLVCS} For Load (literal), LDR, memory locations accessed by the instruction are restricted by the capability in PCC.

R_{CXQNV} Memory locations accessed by instruction fetch are restricted by the capability in PCC.

R_{CNSTH} For a cache maintenance by VA instruction, the required Capability Permissions are as follows:

- `IC IVAU`: Load permission.
- `DC C(I)VA*`: Load permission.
- `DC IVAC`: Store permission.

R_{CTCDF} For a cache maintenance by VA operation, the input capability provides an address that is contained in a contiguous set of memory locations. This set of memory locations is required to be within the bounds of that capability, with the alignment and number of memory locations in the set defined by the following fields:

- `IC*`: CTR_EL0.IminLine.
- `DC*`, except `DC IVA*`: CTR_EL0.DminLine.
- `DC IVA*`: CTR_EL0.CWG

I_{FQXVN} The requirement in R_{CTCDF} means that, for a cache clean operation or a cache clean and invalidate operation that uses a capability as an input, if the capability used does not describe all bytes of the cache line being cleaned in the Capability Bounds, the operation is not permitted by the Morello architecture.

Software must ensure that cache clean operations, and cache clean and invalidate operations, meet this requirement.

See also:

- Chapter D13.2.33 *CTR_EL0, Cache Type Register*, *Arm® Architecture Reference Manual*.

2.7.3 Capability memory protection exceptions

Load, store, and cache maintenance by VA instructions

R$_{\mathrm{YZYBQ}}$ If a load, store, or cache maintenance by VA instruction uses an invalid capability, the instruction generates a synchronous Data Abort with a capability tag fault.

R$_{\mathrm{GHBFX}}$ If a load, store, or cache maintenance by VA instruction uses a valid sealed capability, but the instruction is a non-permitted unsealing operation, the instruction generates a synchronous Data Abort with a capability sealed fault.

R$_{\mathrm{SZLNW}}$ If a load instruction with an unsealing operation uses a valid sealed capability, but the sealed capability has the wrong ObjectType for the instruction, the instruction generates a synchronous Data Abort with a capability sealed fault.

R$_{\mathrm{QTRFK}}$ An atomic memory access instruction always performs a load and a store operation from the perspective of capability Store, Load, StoreCap, and LoadCap permission checking.

R$_{\mathrm{JQYTZ}}$ A Load, store, or cache maintenance by VA instruction uses the Capability Bounds as an upper and lower limit on the memory locations that can be accessed.

R$_{\mathrm{PWQTJ}}$ If a load, store, or cache maintenance by VA instruction accesses any location at a VA outside of the Capability Bounds, the instruction generates a synchronous Data Abort with a capability bounds fault.

R$_{\mathrm{ZCYYB}}$ If all of the following are true, a store of a valid capability to memory generates a synchronous Data Abort with a capability permission fault:

- The source Capability Global bit is set to 0.
- The StoreLocalCap permission of the capability used for the store is set to 0.

R$_{\mathrm{NTJQD}}$ If the LoadCap permission of the capability used is set to 0, a load to a Capability register clears the Capability Tag of the loaded capability.

R$_{\mathrm{RJZNK}}$ If the StoreCap permission of the capability used is set to 0, a store of a valid capability generates a synchronous Data Abort with a capability permission fault.

R$_{\mathrm{HMXNK}}$ If the Load permission of the capability used is set to 0, a load generates a synchronous Data Abort with a capability permission fault.

R$_{\mathrm{TTHKK}}$ For a cache maintenance by VA which requires read access permission, if the Load permission of the capability used is set to 0, the instruction generates a synchronous Data Abort with a capability permission fault.

R$_{\mathrm{YPPQB}}$ If the Store permission of the capability used is set to 0, a store generates a synchronous Data Abort with a capability permission fault.

R$_{\mathrm{MGWWD}}$ For a cache maintenance by VA which requires write access permission, if the Store permission of the capability used is set to 0, the instruction generates a synchronous Data Abort with a capability permission fault.

R$_{\mathrm{ZFMVL}}$ If a load or store instruction generates a synchronous Data Abort with one of the following, the faulting address is one of the locations accessed by the instruction:

- A capability tag fault.
- A capability sealed fault.
- A capability bounds fault.
- A capability permission fault.

R$_{\mathrm{ZGHNJ}}$ An instruction that both uses a capability and modifies the Capability Value of that capability has two sets of checks:

- The capability checks on using the capability.
- The representability check on modifying the Capability Value.

The capability checks are performed before the representability check.

R$_{\mathrm{PVKGX}}$ An instruction that both uses a sealed capability and modifies that sealed capability has two sets of checks:

- The capability checks on using the capability.
- The sealed capability check on modifying the capability.

The capability checks are performed before the sealed capability check.

R_{VVXZL} If a cache maintenance by VA instruction or a data cache zero by VA instruction generates a synchronous Data Abort with one of the following, the faulting address is the address specified in the register argument of the instruction:

- A capability tag fault.
- A capability sealed fault.
- A capability bounds fault.
- A capability permission fault.

Instruction fetch

R_{GZTVP} If the capability in PCC is invalid, instruction fetch generates a synchronous Instruction Abort with a capability tag fault.

R_{ZZWCP} If the capability in PCC does not have Execute permission, instruction fetch generates a synchronous Instruction Abort with a capability permission fault.

R_{FZVKC} If an instruction fetch accesses any location at a VA outside of the Capability Bounds in PCC, the access generates a synchronous Instruction Abort with a capability bounds fault.

R_{MDMPG} If the capability in PCC is sealed, instruction fetch generates a synchronous Instruction Abort with a capability sealed fault.

IMPLEMENTATION DEFINED behavior

R_{KPSBV} If an atomic operation with a conditional store does not perform a store, it is IMPLEMENTATION DEFINED whether that operation performs a required capability Store, StoreCap, or StoreLocalCap permission check.

R_{HCBBH} If a cache maintenance by VA instruction is implemented as a NOP, it is IMPLEMENTATION DEFINED whether capability memory protection is applied to that operation.

R_{YZHQD} For a memory access, cache maintenance operation, or instruction fetch operation, if any of the following conditions are true, it is IMPLEMENTATION DEFINED whether the operation can cause a capability tag fault, capability sealed fault, capability bounds fault, or capability permission fault.

- Stage 1 translation is enabled and the operation is to an address outside the maximum VA range or VA subranges for that stage of translation.
- Stage 1 translation is disabled and the operation is to an address larger than the implemented PA size.

R_{ZXDMZ} If an LDCT or STCT instruction accesses a Non-cacheable location, it is IMPLEMENTATION DEFINED whether the access generates a Data Abort caused by a LDCT/STCT to Non-cacheable memory.

See also:

- Chapter D1.13.5, *Taking an interrupt or other exception during a multi-access load or store*, *Arm® Architecture Reference Manual*.
- 2.13 *Exception model*

2.7.4 Recursive immutability

R_{YYPMC} If a valid unsealed capability is loaded using a capability without MutableLoad permission, the MutableLoad, Store, StoreCap, and StoreLocalCap permissions of the loaded capability are cleared.

2.8 Capability memory relocation

$\mathrm{R}_{\mathrm{BZSPS}}$ For a branch instruction variant using a 64-bit target address, and for return instructions returning to a 64-bit return address, if CCTLR_ELx.PCCBO is 1 and the PE is in ELx, the capability base in PCC is added to the address written to the PC.

$\mathrm{R}_{\mathrm{PHVFM}}$ For branch with link instructions writing a 64-bit return address to X30, if CCTLR_ELx.PCCBO is 1 and the PE is in ELx, the instructions subtract the PCC base from the PC used to generate the link address.

$\mathrm{R}_{\mathrm{VTNGL}}$ For a PC-relative address calculation instruction writing a 64-bit address to a destination register, if CCTLR_ELx.PCCBO is 1 and the PE is in ELx, the instruction subtracts the PCC base from the PC used to generate the address.

$\mathrm{R}_{\mathrm{GFXBJ}}$ For load and store, cache maintenance by VA, and prefetch instructions using a 64-bit base address, if CCTLR_ELx.DDCBO is 1 and the PE is executing in ELx, the instructions add the DDC base to the address used to perform the access.

$\mathrm{R}_{\mathrm{WLPTB}}$ For a `CVTD*` instruction writing a 64-bit value to a destination register, if CCTLR_ELx.DDCBO is 1 and the PE is executing in ELx, the instruction subtracts the DDC base from the value written.

$\mathrm{R}_{\mathrm{ZZSZP}}$ If CCTLR_ELx.DDCBO is 1 and the PE is executing in ELx, `CVT(flag setting)` subtracts the base of the second source register from the 64-bit value written to the destination register.

$\mathrm{I}_{\mathrm{PJKGP}}$ Software must be aware of $\mathrm{R}_{\mathrm{ZZSZP}}$ to ensure that a suitable capability is written to the second source register for `CVT(flag setting)`. If CCTLR_ELx.DDCBO is 1 and the PE is executing in ELx, the DDC used by the subtraction is the one in the same context as the instruction.

$\mathrm{R}_{\mathrm{WSWGD}}$ For a `CVT(D)(Z)` instruction writing a capability to a destination register, if CCTLR_ELx.DDCBO is 1 and the PE is executing in ELx, the instruction adds the DDC base to the Capability Value.

$\mathrm{R}_{\mathrm{MDMXN}}$ For a `CVTP` instruction writing a 64-bit value to a destination register, if CCTLR_ELx.PCCBO is 1 and the PE is executing in ELx, the instruction subtracts the PCC base from the value written.

$\mathrm{R}_{\mathrm{MKGPV}}$ For a `CVTP(Z)` instruction writing a capability to a destination register, if CCTLR_ELx.PCCBO is 1 and the PE executes in ELx, the instruction adds the PCC base to the Capability Value.

2.9 Compartment ID

I_{LRZVM} The CompartmentID permission does not have an architecturally observable effect. The intent is to provide an unforgeable value that is distinct from other capability and non-capability values which hardware can use to partition the behavior of execution prediction resources to reduce the opportunity for side-channel attacks.

R_{XLYMV} The Morello architecture defines a compartment context ID as a value that can be used by hardware to partition the behavior of execution prediction resources to reduce the opportunity for side-channel attacks.

R_{BWXVW} A compartment context ID is a capability.

R_{CSPXQ} If all of the following are true for a capability, it represents a compartment context ID that is distinct from a compartment context ID defined by a capability where any of the following are not true, or where the Capability Value is different:

- The capability is valid.
- The capability is unsealed.
- The value is within the Capability Bounds.
- The capability has CompartmentID permission.

I_{HTCXS} The capability in CID_EL0 is the current compartment context ID.

I_{BYCZR} The current compartment context ID may be used by an implementation as part of the execution context of an execution prediction resource.

R_{CGMBK} If the current compartment context ID is part of the execution context of an execution prediction resource, any predictions made by the execution prediction resource cannot be based on information gathered from an execution with a compartment context ID that is distinct from the current compartment context ID.

See also:

- 2.5 *Capability encoding*: information about modifications which can make a capability non-representable.

2.10 Instruction set selection

R_{ZRMXS} PSTATE.C64 determines the current instruction set:

- PSTATE.C64 is 0: The current instruction set is A64.
- PSTATE.C64 is 1: The current instruction set is C64.

R_{ZTMWK} If executing an instruction, PSTATE.C64 is updated by any of the following:

- The Capability Value[0] of a branch with a capability target.
- A BX #4.

R_{GVJPY} When a branch with link instruction writes a capability to C30, PSTATE.C64 is copied to the Capability Value[0] in C30.

R_{XQNPW} If PSTATE.C64 is 0, all of the following are true:

- A branch and link instruction writes the link address to X30.
- A PC-relative address generation instruction writes an address to Xd.
- A Cache maintenance by VA instruction uses the 64-bit address in Xn, with capability memory relocation applied.

R_{TXVNQ} If PSTATE.C64 is 1, all of the following are true:

- A branch and link instruction writes the link address to C30.
- A PC-relative address generation instruction writes an address to Cd.
- A Cache maintenance by VA instruction uses Capability address in Cn.

I_{QQMVV} In Morello instruction forms are encoded the same in A64 and C64 but with a different interpretation of the operands depending on the state of PSTATE.C64.

In particular, memory access instructions encoded in A64 to use a 64-bit base register, use a Capability base register in C64, and vice versa.

See also:

- 4.1 *The instruction sets*: information about the A64 and C64 instruction sets.

2.11 Reset

R_{VFMMV} CMAX is a capability with all of the following:

- Maximum Capability Bounds: the base is `0x0` and the limit is 2ˆ64.
- Maximum Capability Permissions.
- Executive is 1.
- ObjectType is 0.
- Tag is 1.

R_{FHMFL} On a reset, the following state is defined:

- PCC:
  - The Capability Value of PCC is determined by RVBAR_ELx for the highest implemented Exception level.
  - The rest of PCC is set to CMAX.
- All DDC_ELx:
  - The Capability Value of DDC_ELx is 0.
  - The rest of DDC_ELx is set to CMAX.
- PSTATE.C64 is set to 0.
- CPTR_EL3.EC is set to 0.
- All other Capability registers are UNKNOWN.

R_{LFSPN} On a reset, the state of caches is IMPLEMENTATION DEFINED.

R_{CGXJK} On a reset, the sequence of operations to invalidate capabilities from caches is IMPLEMENTATION DEFINED.

I_{GPJYK} On a system reset, the state of system memory and system caches is IMPLEMENTATION DEFINED.

I_{NNHHF} On a system reset, the sequence of operations to invalidate capabilities from system memory and system caches is IMPLEMENTATION DEFINED.

See also:

- Chapter D1.9.1, *PE state on reset to AArch64 state*, *Arm® Architecture Reference Manual, Armv8-A*: more details about PE state on reset.
- Chapter D4.4.5, *Behavior of caches at reset*, *Arm® Architecture Reference Manual, Armv8-A*: more details about caches on reset.

2.12 Access to the Morello architecture

R_{XWTKD} Access to the Morello architecture can be trapped at each Exception level.

R_{GRLBX} If access to the Morello architecture is trapped at an Exception level, ELx, access to the Morello architecture at all Exception levels lower than ELx is also trapped.

R_{PZHJT} Access to the Morello architecture is controlled by the following:

- CPACR_EL1.CEN.
- CPTR_EL2.TC.
- CPTR_EL2.CEN.
- CPTR_EL3.EC.

R_{TPNMD} If access to the Morello architecture is trapped at ELx and when the PE executes in ELx, all of the following are true:

- Access to any CCTLR_ELy is trapped unless it is UNDEFINED in ELx.
- If executing at EL2, CHCR_EL2 is trapped.
- If executing at EL3, CSCR_EL3 and CHCR_EL2 are trapped.
- Instructions added to A64 by the Morello architecture are trapped.

R_{VCNGF} If access to the Morello architecture is trapped at ELx, the architecture has no effect on the following:

- The effects of controls in CCTLR_ELx.
- The effects of PCC.
- The effects of DDC.
- Capability memory relocation.
- The effect of PSTATE.C64.

R_{KWQVW} If access to the Morello architecture is trapped, accessing the Morello architecture causes a synchronous exception.

R_{RPPMH} A synchronous exception due to an access to the Morello architecture being trapped is reported with an Exception class of Access to the Morello architecture trapped as a result of CPACR_EL1.CEN, CPTR_EL2.TC, CPTR_EL2.CEN, or CPTR_EL3.EC.

R_{NHZKT} For an instruction that is UNPREDICTABLE in an Exception level due to access to the Morello architecture being disabled, it is IMPLEMENTATION DEFINED whether that instruction can cause a capability exception.

2.13 Exception model

R_{FVQQT} The Morello architecture provides the following Exception model variants:

- If access to the Morello architecture is trapped at ELx, a non-capability exception entry to ELx, and return from ELx.
- If access to the Morello architecture is not trapped at ELx, a capability exception entry to ELx, and return from ELx.

R_{THRFG} The following registers determine which variant of an exception entry or return is configured:

- CPACR_EL1.CEN.
- CPTR_EL2.TC.
- CPTR_EL2.CEN.
- CPTR_EL3.EC.

I_{MVGRH} For the Morello architecture, the exception vectors used when taking an exception are the same as described in *Arm® Architecture Reference Manual, Armv8-A* apart from R_{GXNXG}.

R_{GXNXG} If the PE is in Restricted and an exception is taken from the current Exception level, exception entry uses the same exception vector as an exception taken from the current Exception level with SP_EL0.

R_{RZTFR} On an illegal exception return from ELx, the effect on PSTATE.C64 is one of the following:

- If a non-capability exception return from ELx is configured, it is set to 0.
- If a capability exception return from ELx is configured, it is unchanged.

2.13.1 Non-capability exception entry or return

R_{JLYXK} If a non-capability exception entry to ELx is configured, on exception entry to ELx, the Morello architecture changes the following aspects in the existing Armv8-A architecture:

- PSTATE.C64 is set to 0.
- The Capability Value of PCC is set to VBAR_ELx, with VBAR_ELx[10:0] treated as zero, plus the vector offset.

R_{YTFBY} If a non-capability exception return from ELx is configured, on exception return from ELx, the Morello architecture changes the following aspects of the existing Armv8-A architecture:

- ELR_ELx[63:0] is copied to the Capability Value in PCC.
- PSTATE.C64 is set to 0.

2.13.2 Capability exception entry and return

R_{FBWJT} The following registers are extended to 129-bit to support capability exception handling:

Register mnemonic	Description
SP_ELx	Stack Pointer registers
ELR_ELx	Exception Link Registers
VBAR_ELx	Vector Base Address Registers

R_{YSMLC} If capability exception entry and return are configured, the preferred exception return capability generated on an exception is a capability with the Capability Value set to the preferred return address for the exception.

R_{KHSLH} If capability exception entry is configured for ELx, on exception entry to ELx, the Morello architecture changes the existing Armv8-A architecture in all of the following aspects:

- ELR_ELx is set to the preferred exception return capability.
- PSTATE.C64 is set to CCTLR_ELx.C64E.
- PCC is set to the capability in VBAR_ELx, with VBAR_ELx[10:0] treated as zero, plus the vector offset.

R_{VPMLJ} If capability exception return is configured for ELx, on exception return from ELx, the Morello Architecture changes the existing Armv8-A architecture in all of the following aspects:

- ELR_ELx is copied to PCC.
- If the exception return is to an Exception level where access to the Morello architecture is not trapped, SPSR_ELx.C64 is copied to PSTATE.C64.
- If the exception return is to an Exception level where access to the Morello architecture is trapped, PSTATE.C64 is set to 0.

I_{BRTMS} If capability exception return is configured, and the value in ELR_ELx[1:0] is not 0, a subsequent instruction fetch using PCC generates a PC alignment fault.

R_{LNNPH} If capability exception return is configured for ELx and the Capability Bounds to be written to PCC are invalid, on an exception return from ELx the Capability Tag of the capability written to PCC is cleared.

See also:

- Chapter E1.2.4 *Process state, PSTATE*, *Arm® Architecture Reference Manual, Armv8-A*.
- Chapter D1.10 *Exception entry*, *Arm® Architecture Reference Manual, Armv8-A*.
- 2.5.1 *Morello Bounds format*: information about valid and invalid Capability Bounds.

2.13.3 Exception types

I_{MMJJD} The Morello architecture introduces new types of exception reported using both existing Exception classes and new Exception classes:

Name of the fault	Exception class	Section for more information
Alignment fault	Data Abort	2.4 *Capabilities in memory*
Capability access fault due to SC and LC bits in the translation table	Synchronous Data Abort	2.14.1 *Translation table descriptors*
Capability bounds fault on data access	Synchronous Data Abort	2.7.3 *Capability memory protection exceptions*
Capability bounds fault on instruction fetch	Synchronous Instruction Abort	2.7.3 *Capability memory protection exceptions*
Capability permission fault on data access	Synchronous Data Abort	2.7.3 *Capability memory protection exceptions*
Capability permission fault on instruction fetch	Synchronous Instruction Abort	2.7.3 *Capability memory protection exceptions*
Capability sealed fault on data access	Synchronous Data Abort	2.7.3 *Capability memory protection exceptions*

Name of the fault	Exception class	Section for more information
Capability sealed fault on instruction fetch	Synchronous Instruction Abort	*2.7.3 Capability memory protection exceptions*
Capability tag fault on data access	Synchronous Data Abort	*2.7.3 Capability memory protection exceptions*
Capability tag fault on instruction fetch	Synchronous Instruction Abort	*2.7.3 Capability memory protection exceptions*
Trap due to any of the following: • CPACR_EL1.CEN. • CPTR_EL2.TC. • CPTR_EL2.CEN. • CPTR_EL3.EC.	Access to the Morello architecture trapped as a result of any of the following: • CPACR_EL1.CEN. • CPTR_EL2.TC. • CPTR_EL2.CEN. • CPTR_EL3.EC.	*2.12 Access to the Morello architecture*
Trapped 64-bit MRS, MSR due to System permission	Trapped MSR, MRS, or System instruction execution in AArch64 state exception	*2.7.1 System permission*
Trapped capability MRS, MSR due to System permission	Trapped capability MSR or MRS instruction execution exception	*2.7.1 System permission*

R_{MSLGB} On a stage 2 fault that is caused by the access of a capability, ESR_EL2.ISV is 0.

See also:

- Chapter G1.16.8, *Data Abort exception*, *Arm® Architecture Reference Manual, Armv8-A*.

2.13.4 Exception routing

R_{TYNPY} An exception caused by use of the Capability Tag, Capability ObjectType, Capability Permissions, or Capability Bounds in a capability is called a *capability exception*.

R_{WFQXC} The Morello architecture defines the following capability exceptions:

- Capability tag fault.
- Capability sealed fault.
- Capability permission fault.
- Capability bounds fault.
- Trapped capability MRS, MSR due to System permission.
- Trapped 64-bit MRS, MSR due to System permission.

R_{KLRDW} If a capability exception targets an Exception level where access to the Morello architecture is trapped, it is routed to the lowest Exception level where access to the Morello architecture is not trapped. If access to the Morello architecture is trapped at all Exception levels, the exception is routed to the highest implemented Exception level.

2.13.5 Exception priorities

I_{MKBWQ} This section outlines the priority of the exceptions introduced by the Morello architecture regarding the synchronous exception prioritization list in Chapter D1.12.4 *Synchronous exception prioritization for exceptions taken to AArch64 state*, *Arm® Architecture Reference Manual, Armv8-A*.

R_{NNLGC} The following table introduces the prioritization of Morello faults and exceptions within existing exception prioritization in the base architecture, where 1 is the highest priority. The base priority refers to the specific issue of *Arm® Architecture Reference Manual, Armv8-A* indicated in Arm publications section of this document.

Name of the fault	Reporting mechanism	Base priority	Sub-priority
Capability tag fault	Synchronous Instruction Abort	6.5	1
Capability sealed fault	Synchronous Instruction Abort	6.5	2
Capability permission fault	Synchronous Instruction Abort	6.5	3
Capability bounds fault	Synchronous Instruction Abort	6.5	4
Executive/Restricted banking	Attempting to execute an instruction that is UNDEFINED	13	-
Trapped capability `MRS`, `MSR` due to System permission	Trapped capability MSR or MRS instruction execution exception	13.5	-
Trapped 64-bit `MRS`, `MSR` due to System permission	Trapped MSR, MRS, or System instruction execution in AArch64 state exception	13.5	-
Trap due to CPACR_EL1	Access to the Morello architecture trapped as a result of CPACR_EL1.CEN, CPTR_EL2.CEN, CPTR_EL2.TC, or CPTR_EL3.EC	14	-
Trap due to CPTR_EL2	Access to the Morello architecture trapped as a result of CPACR_EL1.CEN, CPTR_EL2.CEN, CPTR_EL2.TC, or CPTR_EL3.EC	16	-
Trap due to CPTR_EL3	Access to the Morello architecture trapped as a result of CPACR_EL1.CEN, CPTR_EL2.CEN, CPTR_EL2.TC, or CPTR_EL3.EC	23	-
Capability tag fault	Synchronous Data Abort	28.5	1
Capability sealed fault	Synchronous Data Abort	28.5	2
Capability permission fault	Synchronous Data Abort	28.5	3
Capability bounds fault	Synchronous Data Abort	28.5	4
Alignment fault caused by LDCT/STCT to Non-cacheable memory	Synchronous Data Abort	29	27.5
Capability access fault - SC stage 1	Synchronous Data Abort	30.5	1
Capability access fault - SC stage 2	Synchronous Data Abort	30.5	2
Capability access fault - LC on an access to Device memory	Synchronous Data Abort	30.5	3
Capability access fault - LC on an Atomic access	Synchronous Data Abort	30.5*	3
Capability access fault - LC on an access to Normal memory	Synchronous Data Abort	32	-

* It is IMPLEMENTATION DEFINED whether Capability access fault - LC on an Atomic access is prioritized at 30.5 or 32.

The Morello architecture does not allow synchronous External aborts to be prioritized at 29.

A 0.5 increment in the base priority indicates that the Morello exception is located in between two exception priorities of the base architecture.

A decimal number in the subpriority indicates that the base architecture has sublists and the Morello exception is inserted into the sublist.

I_{TQHYY} In the base architecture, exceptions due to attempting to execute an instruction that is defined to be inaccessible at the current Exception level, regardless of any enables or traps, are in priority 13. The Morello architecture clarifies that this also includes instructions which are not accessible due to the current Security state.

R_{HBVNP} For capability exceptions reported as a Synchronous Data Abort, if an instruction results in more than one single-copy atomic memory access, the prioritization between synchronous exceptions generated on each of those different memory accesses is not defined by the architecture.

See also:

- Chapter D1.12.4, *Synchronous exception prioritization for exceptions taken to AArch64 state*, *Arm® Architecture Reference Manual, Armv8-A*: Main prioritization of exceptions for the base architecture.
- Chapter D5.8.3, *AArch64 state prioritization of synchronous aborts from a single stage of address translation*, *Arm® Architecture Reference Manual, Armv8-A*: Sublist for some Synchronous Data Abort.

2.14 The Virtual Memory System Architecture

I_{BFVBR} This section requires understanding of the Armv8 Virtual Memory System Architecture (VMSA).

A group of Translation Table Base Registers, TTBRy_ELx, and Capability Control Registers, CCTLR_ELx, are used, and the value of x and y depends on the relevant translation stage and the translation table.

In this section, the variable y is used to indicate the address range and therefore the relevant TTBRy_ELx. The combination of x and y in TTBRy_ELx correlates to the combination used in the Page table tag generation bit in CCTLR_ELx.TGENy, which controls whether to fault a load of a valid capability.

MMU capability access controls

R_{WXVWF} When the Morello architecture is implemented, MMU capability access controls provide control of access to valid capabilities in memory.

R_{JJNSN} For the purpose of MMU capability access controls, an atomic access is treated as both loading and storing a capability.

MMU faulting of stores of valid capabilities

R_{ZGDGP} A memory location can be marked as faulting stores of valid capabilities.

R_{GQRQJ} If a location is marked as faulting stores of valid capabilities, a store of a valid capability to that location causes a capability access fault, and the write to the location does not occur.

R_{NTKXV} Each stage of translation for a translation regime can mark a location as faulting stores of valid capabilities.

R_{JQHGK} Stage 1 faulting of stores of valid capabilities to a location in a translation regime is controlled by the SC and CDBM bits in the stage 1 translation table entry block and page descriptor for that location.

R_{GKLNJ} Stage 2 faulting of stores of valid capabilities to a location in a translation regime is controlled by the SC and CDBM bits in the stage 2 translation table entry block and page descriptor for that location.

R_{PQKQY} If a location is marked as faulting stores of valid capabilities, and an atomic operation with a conditional store of a valid capability to that location does not perform the store, it is IMPLEMENTATION DEFINED whether that operation causes a Capability access fault.

R_{DLYTV} If a stage of translation for a translation regime is disabled, that stage of translation does not cause a Capability access fault due to a store of a valid capability.

R_{FQDQJ} If an exception due to a Capability access fault on a store of a valid capability is taken to ELx, the lowest faulting address is recorded in FAR_ELx.

R_{SLRGN} If an exception is taken to ELx due to a Capability access fault on a store of a valid capability as part of an atomic access, the exception is reported as a write in ESR_ELx.WnR.

R_{XNLFJ} For the purpose of faulting stores of valid capabilities, a STCT instruction is treated as storing capabilities.

R_{ZKDFC} If an instruction stores more than one capability, and at least one of the stores causes a capability access fault, it is CONSTRAINED UNPREDICTABLE whether any capability stored by the instruction which does not cause a fault is stored to memory.

MMU tracking of capability stores of valid capabilities

R_{BVHDN} A memory location can be marked as tracking stores of valid capabilities.

R_{GQKPD} If a location is marked as tracking stores of valid capabilities, and if a valid capability is stored to that location, that location is marked as Capability dirty, instead of generating a Capability access fault.

R_{PGBNQ} If an instruction stores more than one capability to memory, each store of a valid capability is tracked independently.

R_{BBTCZ} Each stage of translation can independently mark a location as tracking stores of valid capabilities.

R_{MTCWN} Each stage of translation can independently mark a location as Capability dirty.

R_{GXLYY} Stage 1 tracking of stores of valid capabilities to a location in a translation regime is controlled by the CDBM bit in the stage 1 translation table entry block and page descriptor for that location.

R_{LXSXB} Stage 2 tracking of stores of valid capabilities to a location in a translation regime is controlled by the CDBM bit in the stage 2 translation table entry block and page descriptor for that location.

R_{JFSGC} Stage 1 Capability dirty state for a location in a translation regime is recorded by setting the SC bit to 1 in the stage 1 translation table entry block and page descriptor for that location.

R_{QJDRL} Stage 2 Capability dirty state for a location in a translation regime is recorded setting the SC bit to 1 in the stage 2 translation table entry block and page descriptor for that location.

I_{HBKYK} Tracking of capability writes follows the same principles as Hardware management of dirty state as defined in Chapter D5.4.11, *Hardware management of the Access flag and dirty state*, *Arm® Architecture Reference Manual, Armv8-A*.

R_{GVCBG} If a location is marked as tracking stores of valid capabilities, and an atomic operation with a conditional store of a valid capability to that location does not perform the store, it is IMPLEMENTATION DEFINED whether the store is tracked.

R_{QBLBN} If a stage of translation for a translation regime is disabled, that stage of translation does not track stores of valid capabilities.

R_{DHRCK} For the purpose of tracking stores of valid capabilities, a STCT instruction is treated as storing capabilities.

MMU faulting of loads of valid capabilities

R_{SXCVB} A memory location can be marked as faulting loads of valid capabilities.

R_{CQJDQ} If a location is marked as faulting loads of valid capabilities, a load of a valid capability from that location causes a Capability access fault.

R_{QDKBL} If a location is marked as Device and as faulting loads of valid capabilities, a load of a capability from that location causes a Capability access fault, and the location is not read.

R_{HQVST} The stage 1 translation for a translation regime can mark a location as faulting loads of valid capabilities.

R_{CPRKD} Stage 1 faulting of loads of valid capabilities from a location in the translation regime for ELx is controlled by the LC bit in the stage 1 translation table entry block and page descriptor, and the CCTLR_ELx.TGENy field, for that location.

R_{RKGLC} If a stage of translation for a translation regime is disabled, that stage of translation cannot cause a Capability access fault due to a load of a valid capability.

R_{GFNJJ} If an exception is taken to ELx due to a Capability access fault on a load of a valid capability, the lowest faulting address is recorded in FAR_ELx.

R_{NKSBV} If a location is marked as faulting loads of valid capabilities, and an atomic operation to that location causes a Capability access fault, the location is not written.

R_{VVNDW} If a location is marked as faulting loads of valid capabilities, an atomic operation to that location which would read a valid capability from that location causes a Capability access fault.

R_{TKKMV} If a location is marked as faulting loads of valid capabilities, and an atomic operation to that location would read an invalid capability from that location, it is IMPLEMENTATION DEFINED whether the operation causes a Capability access fault.

R_{KRJXL} For the purpose of faulting loads of valid capabilities, a LDCT instruction is treated as loading capabilities.

R_{JFHGB} If an instruction loads more than one capability, and at least one of the loads causes a capability access fault, it is CONSTRAINED UNPREDICTABLE whether any capability loaded by the instruction that does not cause a fault is read from memory.

MMU zeroing of Capability Tags when loading capabilities

R_{RBHHQ} A memory location can be marked as zeroing Capability Tags on loads of capabilities

R_{DJSPV} If a location is marked as zeroing Capability Tags on loads of capabilities, the Capability tag on a capability loaded from that memory is set to zero.

R_{QWGTB} Each stage of translation for a translation regime can mark a location as zeroing Capability Tags on loads of capabilities.

R_{HQBZK} Stage 1 zeroing of Capability Tags on capabilities loaded from a location in a translation regime is controlled by the LC bit in stage 1 translation table entry block and page descriptor for that location.

R_{TRMCY} Stage 2 zeroing of Capability Tags on capabilities loaded from a location in a translation regime is controlled by the LC bit in stage 2 translation table entry block and page descriptor for that location.

R_{YVRVV} If a location is marked as zeroing Capability Tags on loads by Stage 2, a capability loaded from the location is treated as invalid for the purpose of faulting of loads of valid capabilities.

R_{GVMCL} If a stage of translation for a translation regime is disabled, that stage of translation does not cause zeroing of Capability Tags on loaded capabilities.

R_{RHTRV} For the purpose of MMU zeroing of Capability Tags when loading capabilities, a LDCT instruction is treated as loading capabilities.

R_{CVTTF} If a memory location is marked as zeroing Capability Tags on loads of capabilities, the zeroing is applied before the application of faulting of loads of valid capabilities from that location.

R_{HFGKN} If an instruction loads more than one capability, each capability is treated independently for the purpose of zeroing of capability Tags on loading capabilities.

2.14.1 Translation table descriptors

R_{HTXWL} For each stage of translation, the following registers contain hardware use control bits for the Block and Page descriptor fields used by the Morello architecture.

If a Hardware Use control bit is 0, its corresponding bit in the Block and Page descriptor field is treated as 0:

Hardware use control bit	Translation stage	Corresponding Block and Page descriptor bit
TCR_ELx.HWU62	Stage 1	LC, bit 62
TCR_ELx.HWU61	Stage 1	LC, bit 61
TCR_ELx.HWU60	Stage 1	SC, bit 60
TCR_ELx.HWU59	Stage 1	CDBM, bit 59
VTCR_EL2.HWU61	Stage 2	LC, bit 61
VTCR_EL2.HWU60	Stage 2	SC, bit 60
VTCR_EL2.HWU59	Stage 2	CDBM, bit 59

R_{LBFNG} The table below outlines the stage 1 Block and Page descriptor fields, which are part of the PBHA bits:

Name	Field	Description
LC	62:61	Control of loads of capabilities from memory: • `0b00`: Zero Capability Tags. • `0b01`: No effect. • `0b10`: If CCTLR_ELx.TGENy is 1, fault loads of valid capabilities; otherwise no effect. The value of x and y is determined by the translation table base register TTBRy_ELx used for the access. • `0b11`: If CCTLR_ELx.TGENy is 0, fault loads of valid capabilities; otherwise no effect. The value of x and y is determined by the translation table base register TTBRy_ELx used for the access.
SC	60	Control of stores of valid capabilities to memory: • `0b0`: If CDBM is 0, fault stores of valid capabilities, otherwise no effect. • `0b1`: No effect.
CDBM	59	Control tracking of stores of valid capabilities: • `0b0`: No effect • `0b1`: Track stores of valid capabilities.

R_{KPDCT} The stage 2 Block and Page descriptors are extended to control access to capabilities in capability-tagged memory.

The table below outlines the stage 2 Block and Page descriptor fields, which are part of the PBHA bits:

Name	Field	Description
LC	61	Control of loads capabilities from memory: • `0b00`: Zero Capability tags • `0b01`: No effect
SC	60	Control of stores of valid capabilities to memory: • `0b0`: If CDBM is 0, fault stores of valid capabilities , otherwise no effect. • `0b1`: No effect.
CDBM	59	Control tracking of stores of valid capabilities: • `0b0`: No effect. • `0b1`: Track stores of valid capabilities.

See also:

- Chapter D5.3.3, *Memory attribute fields in the VMSAv8-64 translation table format descriptors*, *Arm® Architecture Reference Manual, Armv8-A*.

2.15 Self-hosted debug

2.15.1 Watchpoints

$\mathrm{R}_{\mathtt{BGBZW}}$ For the purpose of watchpoint checking, the following instructions are treated as accessing four capabilities:

- STCT.
- LDCT.

$\mathrm{R}_{\mathtt{HXVZS}}$ For the purpose of watchpoint checking, the following instructions are treated as accessing an entire cacheline:

- STXP.
- STLXP.

See also:

- Chapter D2, *AArch64 Self-hosted Debug*, *Arm® Architecture Reference Manual, Armv8-A*.

2.16 The Embedded Trace Macrocell architecture

2.16.1 Exception instruction trace element

$\mathrm{R}_{\mathrm{TCXZX}}$ The Embedded Trace Macrocell architecture groups exceptions into different types. For the exceptions added by the Morello architecture, the exception types used in the Embedded Trace Macrocell are the following:

The Morello architecture exception types	Exception type
Trap due to any of the following: • CPACR_EL1.CEN. • CPTR_EL2.TC. • CPTR_EL2.CEN. • CPTR_EL3.EC.	Trap
Trapped capability `MRS`, `MSR` due to System permission	Trap
Trapped 64-bit `MRS`, `MSR` due to System permission	Trap
Capability permission fault on instruction fetch	Inst Fault
Capability sealed fault on instruction fetch	Inst Fault
Capability bounds fault on instruction fetch	Inst Fault
Capability access fault due to SC and LC bits in the translation table	Data Fault
Capability bounds fault on data access	Data Fault
Capability permission fault on data access	Data Fault
Capability sealed fault on data access	Data Fault
Capability tag fault on data access	Data Fault

See also:

- Chapter 5.2.7, *Exception instruction trace element*, *Arm® Embedded Trace Macrocell Architecture Specification*.

2.16.2 Address and Context tracing packets

$\mathrm{I}_{\mathrm{KMSXD}}$ The instruction set can be decoded by the state of the SF bit and the header byte of an Address packet.

$\mathrm{R}_{\mathrm{NJWNK}}$ The instruction set is indicated by the combination of the SF bit and the header byte of an Address packet, as the following table shows:

SF bit value	Instruction set	Alignment	ISA in use
1	IS1	Halfword-aligned	C64
1	IS0	Word-aligned	A64

I_{KMWPR} The table in R_{NJWNK} only includes information for when SF bit is 1, because the Morello architecture does not support the instruction sets A32 and T32, which are indicated by the SF bit being 0.

R_{KHQMC} The Embedded Trace Macrocell architecture groups instructions into different types.

For the instructions added by the Morello architecture, the instructions categorized as direct branches by the Embedded Trace Macrocell are the following:

Instruction	Description	Link?	Return from exception?
BX	Branch Exchange	No	No

For the instructions added by the Morello architecture, the instructions categorized as indirect branches by the Embedded Trace Macrocell are the following:

Instruction	Description	Link?	Return from exception?
BLR (indirect)	Branch with Link to capability Register	Yes	No
BLR (memory indirect)	Unseal load, branch and link	Yes	No
BLRR	Branch with Link to capability Register with possible switch to Restricted	Yes	No
BLRS (capability)	Branch with Link to sealed capability	Yes	No
BLRS (pair of capabilities)	Branch with Link to sealed capability Register with possible switch to Restricted	Yes	No
BR (indirect)	Branch to capability Register	No	No
BR (memory indirect)	Unseal load and branch	No	No
BRR	Branch to capability Register with possible switch to Restricted	No	No
BRS (capability)	Branch to sealed capability	No	No
BRS (pair of capabilities)	Branch to sealed capability pair	No	No
LDPBLR	Load Pair of capabilities and Branch with Link	Yes	No
LDPBR	Load Pair of capabilities and Branch	No	No
RET	Return from subroutine	No	No
RETR	Return from subroutine with possible switch to Restricted	No	No
RETS (capability)	Return to sealed capability	No	No
RETS (pair of capabilities)	Return to sealed capability pair	No	No

See also:

- Chapter D3, *AArch64 Self-hosted Trace*, *Arm® Architecture Reference Manual, Armv8-A*.
- Chapter 6.4.12, *Address and Context tracing packets*, *Arm® Embedded Trace Macrocell Architecture Specification*.
- Chapter F.1.1, *A64 Instruction set*, *Arm® Embedded Trace Macrocell Architecture Specification*.

2.17 Performance Monitoring Unit

R_{LCHRS} The Morello architecture adds the following performance events, using the IMPLEMENTATION DEFINED events space defined for an Armv8 implementation, `0x00C0-0x03FF`.

Events added by the Morello architecture are in the range 0x0200-0x03FF.

Morello PMU events

`0x0200`, BR_MIS_PRED_RS Branch mispredict restricted.
The counter counts each correction to the predicted program flow that occurs because of a misprediction or no prediction, and relates to switches between Restricted and Executive.

`0x0201`, BR_MIS_PRED_C64 Branch mispredict C64.
The counter counts each correction to the predicted program flow that occurs because of a misprediction or no prediction, and relates to switches between A64 and C64.

`0x0202`, BR_MIS_PRED_SYS Branch mispredict system permission.
The counter counts each correction to the predicted program flow that occurs because of a misprediction or no prediction, and relates to System permission.

`0x0203`, PCCRF_FULL PCC register file full.
The counter counts every cycle counted by the CPU_CYCLES event on which no operation was issued because the PCC write tracking register file was full.

`0x0204`, EXECUTIVE_ENTRY Entry to Executive, Operations Speculatively Executed.
The counter counts speculatively executed operations that cause an entry into Executive.

`0x0205`, EXECUTIVE_EXIT Exit from Executive, Operations Speculatively Executed.
The counter counts speculatively executed operations that cause an exit from Executive.

`0x0206`, INST_SPEC_A64 Instructions in A64, Operations Speculatively Executed.
The counter counts speculatively executed operations due to all instructions in A64.

`0x0207`, INST_SPEC_C64 Instructions in C64, Operations Speculatively Executed.
The counter counts speculatively executed operations due to all instructions in C64.

`0x0208`, CID_EL0_WRITE_RETIRED Instruction architecturally executed, Write to CID_EL0.
The counter counts architecturally executed instructions which write to the Compartment ID Register.

`0x0209`, DDC_WRITE_RETIRED Instruction architecturally executed, Write to DDC_ELx, RDDC_EL0.
The counter counts architecturally executed instructions which write to any Default Data Capability.

`0x020A`, DDC_READ_SPEC Read from DDC_ELx, RDDC_EL0, Operations Speculatively Executed.
The counter counts speculatively executed operations which read from any Default Data Capability.

`0x020B`, INST_SPEC_CVTD CVTD Instructions, Operations Speculatively Executed.
The counter counts speculatively executed operations due to the following instructions:

- CVTD (not flag setting): Convert pointer to capability offset from DDC.
- CVTD (flag setting): Convert capability to pointer offset from DDC, setting flags.
- CVTDZ: Convert pointer to capability offset from DDC, with null capability from zero semantics.

`0x020E`, INST_SPEC_SCBNDS_NONEXACT SCBNDS or SCBNDSE Instructions which do not set exact bounds, Operations Speculatively Executed.
The counter counts speculatively executed operations due to any of the following instructions not succeeding in setting the requested bounds exactly:

- SCBNDS (register): Set Bounds (register).
- SCBNDS (immediate): Set Bounds (immediate).
- SCBNDSE: Set Bounds Exact.

`0x020F`, CDBM_SET_SC SC set due to CDBM.

The counter counts each setting of the permission bit to write Capability Tags to memory in a translation table entry which is due to the CDBM bit being set.

`0x0210`, CAP_LD_SPEC Capability Load Instructions, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Capability load instructions.

`0x0211`, **CAP_ST_SPEC** Capability Store Instructions, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Capability store instructions.

`0x0212`, **CAP_ALT_LD_SPEC** Alternate Base Capability Load Instructions, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Alternate Base Capability load instructions.

`0x0213`, **CAP_ALT_ST_SPEC** Alternate Base Capability Store Instructions, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Alternate Base Capability store instructions.

`0x0214`, **ALT_LD_SPEC** Alternate Base Load Instructions, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Alternate Base load instructions.

`0x0215`, **ALT_ST_SPEC** Alternate Base Store Instructions, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Alternate Base store instructions.

`0x0216`, **LDCT_SPEC** LDCT Instructions, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Load Tags instructions.

`0x0217`, **LDCT_NO_CAP_SPEC** LDCT Instructions When Capability Tags are Zero, Operations Speculatively Executed.

The counter counts speculatively executed operations due to Load Capability Tags instructions where the Capability Tags to be loaded are all zero.

`0x0218`, **DC_ZVA_RET** Data Cache Zero.

The counter counts architecturally executed DC ZVA instructions.

`0x021A`, **LDCT_REFILL** Data cache refill due to LDCT, Operations Speculatively Executed.

The counter counts each access counted by L1D_CACHE that causes a demand refill of any cache due to execution of an LDCT instruction.

`0x021B`, **STCT_REFILL** Data cache refill due to STCT, Operations Speculatively Executed.

The counter counts each access counted by L1D_CACHE that causes a demand refill of any cache due to execution of an STCT instruction.

`0x021C`, **L1D_CACHE_RD_CTAG** Attributable Level 1 data cache access, read, valid capability.

The counter counts each access counted by L1D_CACHE_RD which loaded a valid capability.

`0x021D`, **L1D_CACHE_WR_CTAG** Attributable Level 1 data cache access, write, valid capability.

The counter counts each access counted by L1D_CACHE_WR which stored a valid capability.

`0x021E`, **L1D_CACHE_WB_CTAG** Attributable Level 1 data cache write-back, valid capability.

The counter counts each access counted by L1D_CACHE_WB where at least one valid capability was present in the cache line.

`0x021F`, **L1D_CACHE_REFILL_RD_CTAG** Attributable Level 1 data cache refill, capability.

The counter counts each access counted by L1D_CACHE_REFILL_RD where at least one valid capability was present in the cache line.

`0x0220`, **L1D_CACHE_REFILL_WR_CTAG** Attributable Level 1 data cache refill, capability.

The counter counts each access counted by L1D_CACHE_REFILL_WR where at least one valid capability was present in the cache line.

`0x0221`, L1D_CACHE_REFILL_INNER_CTAG Attributable Level 1 data cache refill, inner, valid capability.

The counter counts each access counted by L1D_CACHE_REFILL_INNER where at least one valid capability was present in the cache line.

`0x0222`, L1D_CACHE_REFILL_OUTER_CTAG Attributable Level 1 data cache refill, outer, valid capability.

The counter counts each access counted by L1D_CACHE_REFILL_OUTER where at least one valid capability was present in the cache line.

`0x0223`, L1D_CACHE_WB_VICTIM_CTAG Attributable Level 1 data cache Write-Back, victim, valid capability.

The counter counts each access counted by L1D_CACHE_WB_VICTIM where at least one valid capability was present in the cache line.

`0x0224`, L1D_CACHE_WB_CLEAN_CTAG Attributable Level 1 data cache Write-Back, cleaning, and coherency, valid capability.

The counter counts each access counted by L1D_CACHE_WB_CLEAN where at least one valid capability was present in the cache line.

`0x0226`, L2D_CACHE_RD_CTAG Attributable Level 2 data cache access, read, valid capability.

The counter counts each access counted by L2D_CACHE_RD which loaded a valid Capability.

`0x0227`, L2D_CACHE_WR_CTAG Attributable Level 2 data cache access, write, valid capability.

The counter counts each access counted by L2D_CACHE_WR which stored a valid Capability.

`0x0228`, L2D_CACHE_REFILL_RD_CTAG Attributable Level 2 data cache refill, valid capability.

The counter counts each access counted by L2D_CACHE_REFILL_RD where at least one valid capability was present in the cache line.

`0x022A`, L2D_CACHE_WB_VICTIM_CTAG Attributable Level 2 data cache Write-Back, victim, valid capability.

The counter counts each access counted by L2D_CACHE_WB_VICTIM where at least one valid capability was present in the cache line.

`0x022B`, L2D_CACHE_WB_CLEAN_CTAG Attributable Level 2 data cache Write-Back, cleaning and coherency, valid capability.

The counter counts each access counted by L2D_CACHE_WB_CLEAN where at least one valid capability was present in the cache line.

`0x022C`, L2D_CACHE_INVAL_CTAG Attributable Level 2 data cache invalidate, valid capability.

The counter counts each access counted by L2D_CACHE_INVAL where at least one valid capability was present in the cache line.

`0x022D`, BUS_ACCESS_RD_CTAG Bus access, read, valid capability.

The counter counts each access counted by BUS_ACCESS_RD where a Capability Tag was set in at least one beat of the access.

`0x022E`, BUS_ACCESS_WR_CTAG Bus access, write, valid capability.

The counter counts each access counted by BUS_ACCESS_WR where a Capability Tag was set in at least one beat of the access.

`0x022F`, CNT_ST_ZERO_BYTE Store of zeros.

In combination with the CNT_ST_ZERO_16TH_BYTE, the counter counts the number of bytes written by architecturally executed store instructions, not including DC ZVA where only zeros are stored and not including stores which store 16 bytes of zero.

`0x0230`, CNT_ST_ZERO_16_BYTES Store of zeros, 16 byte stores.

The counter counts when 16 bytes of zero are written by an architecturally executed store instruction.

`0x0233`, MEM_ACCESS_RD_CTAG Data memory access, read, valid capability.

The counter counts each access counted by MEM_ACCESS_RD where a Capability Tag was set in at least one part of the access.

`0x0234`, MEM_ACCESS_WR_CTAG Data memory access, write, valid capability.

The counter counts each access counted by MEM_ACCESS_WR where a Capability Tag was set in at least one part of the access.

`0x0235`, CAP_MEM_ACCESS_RD Data memory access, read, capability.

The counter counts each access counted by MEM_ACCESS_RD due to an instruction which loads a capability. It is not sensitive to the validity of the capability.

`0x0236`, CAP_MEM_ACCESS_WR Data memory access, write, capability.

The counter counts each access counted by MEM_ACCESS_WR due to an instruction which stores a capability. It is not sensitive to the validity of the capability.

`0x0237`, INST_SPEC_RESTRICTED Instructions in Restricted, Operations Speculatively Executed.

The counter counts speculatively executed operations due to all instructions in Restricted.

`0x0238`, LD_CAP_PERM_CLR_CTAG Load permission cleared, Operations Speculatively Executed.

The counter counts speculatively executed operations due to load instructions where the capability tag is cleared due to the operation having been performed without LoadCap permission.

See also:

- Chapter D7.11.2 *The PMU event number space and common events*, *Arm® Architecture Reference Manual, Armv8-A*.

2.18 Statistical profiling extension

R_{FXNWM} For the purpose of Statistical profiling, an LDCT instruction is treated as a load of capabilities.

R_{WCKHL} For the purpose of Statistical profiling, an STCT instruction is treated as a store of capabilities.

R_{PWXRJ} For the purpose of Statistical profiling, it is IMPLEMENTATION DEFINED whether LDPBR, LDPBLR, BR (memory indirect), and BLR (memory indirect) are treated as one of the following:

- A load of capabilities or a branch.
- A load of capabilities and a branch.

2.18.1 The Statistical Profiling Buffer

R_{JYXCQ} The writes to the Profiling Buffer are checked against DDC_ELx for the controlling Exception level, after capability memory relocation is applied.

I_{PTYCB} R_{JYXCQ} means that the Profiling Buffer is associated with Executive state in the controlling Exception level.

R_{BDWLM} The DDC_ELx base is added to the Profiling Buffer address defined by PMBPTR_EL1.

R_{DXDVH} For a VA with capability memory relocation applied, the Address packet payload ADDR contains the post-relocation VA.

R_{PPRMG} For a VA with capability memory relocation applied, the buffer pointer value is relocated.

R_{JSDVB} The Profiling Buffer full condition is determined using an unrelocated value derived from PMBPTR_EL1 and a value taken from PMBLIMITR_EL1.

R_{DQDSZ} Faults due to capability memory protection on buffer writes are reported in PMBPTR_EL1.

I_{XFNCQ} Synchronous faults on writes to the Profiling Buffer are prioritized as described in Exception priorities section.

See also:

- Chapter D9.7.1 *Restrictions on the current write pointer*, *Arm*$^{®}$ *Architecture Reference Manual, Armv8-A*.
- Chapter D10.2.1 *Address packet*, *Arm*$^{®}$ *Architecture Reference Manual, Armv8-A*.

2.18.2 Statistical profiling extension packets

R_{BKFLY} The following Operation Type packet payload (load/store) bit assignments are defined for subclasses:

SUBCLASS	Description	Bit assignments are same as
`0b0010000x`	A load/store targeting 129-bit general-purpose registers	General-purpose load/store
`0b001xxx1x`	An atomic operation, load-acquire, store-release, or exclusive targeting 129-bit general-purpose registers	An extended load/store

R_{YCMGT} For the Address packet type, if the INDEX field is `0b00001`, branch target address, the Address packet payload ADDR[0] is always zero.

See also:

- Chapter D10.2.7 *Operation Type packet*, *Arm*$^{®}$ *Architecture Reference Manual, Armv8-A*.

2.19 External debug

2.19.1 Entering Debug state

$\mathrm{R}_{\mathrm{XRJJB}}$ On entry to Debug state, all of the following apply:

- PCC is copied to CDLR_EL0 with the Capability Value set to the preferred restart address for the debug event.
- PSTATE.C64 is copied to DSPSR_EL0.C64.
- PSTATE.C64 is set to 0.

All other behavior is as described in the *Arm® Architecture Reference Manual, Armv8-A*.

See also:

- Chapter H2.3 *Entering Debug state*, *Arm® Architecture Reference Manual, Armv8-A*.

2.19.2 Exiting Debug state

$\mathrm{R}_{\mathrm{YJBHN}}$ On exit from Debug state in ELx, if non-capability exception return from ELx is configured, the Morello architecture changes the following aspects of the existing Armv8-A architecture:

- PCC is set to the Capability in CDLR_EL0.
- PSTATE.C64 is set to 0.

$\mathrm{R}_{\mathrm{LGDCX}}$ On exit from Debug state in ELx, if capability exception return from ELx is configured, the Morello architecture changes the existing Armv8-A architecture in all of the following aspects:

- PCC is set to the Capability in CDLR_EL0.
- If the Debug state exit is an illegal exception return, PSTATE.C64 is left unchanged.
- If the Debug state exit is not an illegal exception return, and is to an Exception level where access to the Morello architecture is not trapped, DSPSR_EL0.C64 is copied to PSTATE.C64.
- If the Debug state exit is not an illegal exception return, and is to an Exception level where access to the Morello architecture is trapped, PSTATE.C64 is set to 0.

See also:

- Chapter H2.5 *Exiting Debug state*, *Arm® Architecture Reference Manual, Armv8-A*.

2.19.3 Executing instructions in Debug state

$\mathrm{R}_{\mathrm{HLGQQ}}$ If the PE is in Debug state, all of the following are true:

- The PE is treated as if in Executive.
- System permission of PCC is treated as 1.
- PCC is UNKNOWN.

$\mathrm{R}_{\mathrm{QHCRC}}$ A write to DLR_EL0 writes to bits [63:0] of CDLR_EL0. It does not change CDLR_EL0 [128:64].

$\mathrm{I}_{\mathrm{VNPCB}}$ The effect of a write to DLR_EL0 on CDLR_EL0 differs to a write to other System registers using a 64-bit access view. This permits a Morello-unaware external debugger to correctly modify the return address without overwriting the rest of the preserved PCC.

2.19.4 Instructions in Debug state

Instructions changed in Debug state

R$_{QYDGQ}$ On executing an instruction other than MSR, where the Armv8-A architecture defines the behavior of the instruction as setting DLR_EL0 to an UNKNOWN value, this behavior is changed by the Morello architecture to preserve the original value of DLR_EL0.

I$_{ZLCRF}$ The change described in R$_{QYDGQ}$ applies in cases where executing an instruction in Debug state is described as CONSTRAINED UNPREDICTABLE in the Armv8-A architecture. One or more of these permitted behaviors include the setting of DLR_EL0 to an UNKNOWN value. All other aspects of the permitted behaviors are as defined in the Armv8-A architecture.

R$_{NVSTF}$ On executing a DCPSx instruction, the Morello architecture changes the following aspects of the existing Armv8-A architecture:

- CCTLR_ELx.C64E is copied to PSTATE.C64.
- DLR_EL0 is left unchanged.

R$_{RQDKQ}$ If non-capability exception return from ELx is configured, on executing a DRPS instruction in ELx, the Morello architecture changes the following aspects of the existing Armv8-A architecture:

- PSTATE.C64 is set to 0.
- DLR_EL0 is left unchanged.

If capability exception return is configured for ELx, on executing a DRPS instruction in ELx, the Morello architecture changes the existing Armv8-A architecture in all of the following aspects:

- If the exception return is to an Exception level where access to the Morello architecture is not trapped, SPSR_ELx.C64 is copied to PSTATE.C64.
- If the exception return is to an Exception level where access to the Morello architecture is trapped, PSTATE.C64 is set to 0.
- DLR_EL0 is left unchanged.

Instructions added in Debug state

I$_{VLXZM}$ The availability of existing instructions in Debug state is unchanged.

R$_{SBYXB}$ The following instructions added by Morello are available in Debug state:

- Add (immediate).
- Subtract (immediate).
- Move from Capability register to System register.
- Move from System register to Capability register.
- Move from Capability register to Special-purpose Capability register.
- Move from Special-purpose Capability register to Capability register.
- Load and store of all data types with and without alternate mode base, other than literal and non-exclusive pair forms.
- Load and store of Capability Tags.
- All atomics.
- Copy From High.
- Copy To High.
- Set the Capability Tag field.
- Get the Tag field of a capability.
- Copy Capability register.
- Load and store of Capability single or exclusive, with or without acquire or release.
- Set Value field of a capability.
- Branch Exchange.

If an instruction added by the Morello architecture is not available in the Debug state, the instruction is CONSTRAINED UNPREDICTABLE and behaves in one of the following ways:

- It is UNDEFINED.
- It executes as a NOP.

- It has the same behavior as in Non-debug state with instructions that read the PC, PCC, or PSTATE fields using an UNKNOWN value for those registers or fields.

R$_{TCMPQ}$ The following instructions are defined in Debug state, and are UNDEFINED in Non-debug state:

- MRS Cd, CDLR_EL0.
- MRS Cd, CDBGDTR_EL0.
- MSR CDLR_EL0, Cn.
- MSR CDBGDTR_EL0, Cn.

2.19.5 Debug Communications Channel (DCC) access

I$_{XHSMC}$ Three 32-bit external Debug registers allow external debug to access the Morello architecture within the PE.

DCC and capabilities

R$_{VYGYG}$ In Debug state, software can transfer a capability to or from external debug by accessing CDBGDTR_EL0.

R$_{BKDHS}$ In Debug state, external debug can transfer a capability to or from software by accessing the following 32-bit External Debug registers:

- DTRTX.
- DTRRX.
- DBGDTR2A.
- DBGDTR2B.
- EDSCR2.

Memory access mode

I$_{NTSWF}$ If the PE is in Debug state and in Memory access mode, and when PSTATE.C64 is 0, memory access is subject to capability memory relocation.

R$_{RTTJG}$ If the PE is in Debug state and in Memory access mode and when PSTATE.C64 is 1, the Morello architecture changes all of the following from the base architecture:

- External reads from DBGDTRTX_EL0 causes the equivalent of `LDR W1, [C0], #4` to be executed.
- External writes to DBGDTRRX_EL0 causes the equivalent of `STR W1, [C0], #4` to be executed.

See also:

- Chapter H4.3.2, *Memory access mode*, *Arm® Architecture Reference Manual, Armv8-A*: behavior resulted from an access by the external debug interface.
- *2.7.2 Capability memory protection*
- *2.8 Capability memory relocation*

Chapter 3

Register definitions

3.1 Register index

I_{XHWZG} This chapter describes the following:

- The base architecture registers extended by the Morello architecture.
- The new registers added in the Morello architecture.

Registers described in this document

I_{JMGWF} Be aware of the following when reading the descriptions of the registers for the base architecture in this supplement:

The register descriptions include references to AArch32, which do not apply in Morello.

Registers that are extended in the Morello architecture to be 129-bit include new accessor descriptions that use the name prefixed with a ‘C’.

Effects of System permission

I_{KHTDY} This chapter does not include detailed descriptions of registers defined in the base architecture where the only change in the Morello architecture is the addition of access controls due to System permission.

For a register that can be accessed at EL0 or EL1, the following code is added to the accessibility pseudocode:

```
if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
    if TargetELForCapabilityExceptions() == EL1 then
        AArch64.SystemAccessTrap(EL1, 0x18);
    elsif TargetELForCapabilityExceptions() == EL2 then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
```

For a register that can be accessed at EL2, the following code is added to the accessibility pseudocode:

```
if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
    if TargetELForCapabilityExceptions() == EL2 then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        AArch64.SystemAccessTrap(EL3, 0x18);
```

For a register that can be accessed at EL3, the following code is added to the accessibility pseudocode:

```
if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
    AArch64.SystemAccessTrap(EL3, 0x18);
```

3.1.1 AArch64 registers

Name	Description
CCTLR_EL0	Capability Control Register (EL0)
CCTLR_EL1	Capability Control Register (EL1)
CCTLR_EL2	Capability Control Register (EL2)
CCTLR_EL3	Capability Control Register (EL3)
CDBGDTR_EL0	Capability Debug Data Transfer Register, half-duplex
CDLR_EL0	Capability Debug Link Register
CHCR_EL2	Capability Hypervisor Configuration Register
CID_EL0	Compartment ID Register
CNTVCT_EL0	Counter-timer Virtual Count register
CPACR_EL1	Architectural Feature Access Control Register
CPTR_EL2	Architectural Feature Trap Register (EL2)
CPTR_EL3	Architectural Feature Trap Register (EL3)
DDC_EL0	Default Data Capability (EL0)
DDC_EL1	Default Data Capability (EL1)
DDC_EL2	Default Data Capability (EL2)
DDC_EL3	Default Data Capability (EL3)
DSPSR_EL0	Debug Saved Program Status Register
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
ELR_EL3	Exception Link Register (EL3)
ESR_EL1	Exception Syndrome Register (EL1)
ESR_EL2	Exception Syndrome Register (EL2)
ESR_EL3	Exception Syndrome Register (EL3)
FAR_EL1	Fault Address Register (EL1)
FAR_EL2	Fault Address Register (EL2)
FAR_EL3	Fault Address Register (EL3)
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1

Name	Description
PMBSR_EL1	Profiling Buffer Status/syndrome Register
RDDC_EL0	Restricted Default Data Capability
RSP_EL0	Restricted Stack Pointer
RTPIDR_EL0	Restricted Read/Write Software Thread ID Register
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL0)
SP_EL2	Stack Pointer (EL0)
SP_EL3	Stack Pointer (EL0)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_EL3	Saved Program Status Register (EL3)
TPIDR_EL0	EL0 Read/Write Software Thread ID Register
TPIDR_EL1	EL1 Software Thread ID Register
TPIDR_EL2	EL2 Software Thread ID Register
TPIDR_EL3	EL3 Software Thread ID Register
TPIDRRO_EL0	EL0 Read-Only Software Thread ID Register
VBAR_EL1	Vector Base Address Register (EL1)
VBAR_EL2	Vector Base Address Register (EL2)
VBAR_EL3	Vector Base Address Register (EL3)

3.1.2 Changes to existing registers

Name	Description
CNTVCT_EL0	Counter-timer Virtual Count register
CPACR_EL1	Architectural Feature Access Control Register
CPTR_EL2	Architectural Feature Trap Register (EL2)
CPTR_EL3	Architectural Feature Trap Register (EL3)
DSPSR_EL0	Debug Saved Program Status Register
ELR_EL1	Exception Link Register (EL1)
ELR_EL2	Exception Link Register (EL2)
ELR_EL3	Exception Link Register (EL3)
ESR_EL1	Exception Syndrome Register (EL1)
ESR_EL2	Exception Syndrome Register (EL2)
ESR_EL3	Exception Syndrome Register (EL3)
FAR_EL1	Fault Address Register (EL1)

Name	Description
FAR_EL2	Fault Address Register (EL2)
FAR_EL3	Fault Address Register (EL3)
ID_AA64PFR1_EL1	AArch64 Processor Feature Register 1
PMBSR_EL1	Profiling Buffer Status/syndrome Register
SP_EL0	Stack Pointer (EL0)
SP_EL1	Stack Pointer (EL0)
SP_EL2	Stack Pointer (EL0)
SP_EL3	Stack Pointer (EL0)
SPSR_EL1	Saved Program Status Register (EL1)
SPSR_EL2	Saved Program Status Register (EL2)
SPSR_EL3	Saved Program Status Register (EL3)
TPIDR_EL0	EL0 Read/Write Software Thread ID Register
TPIDR_EL1	EL1 Software Thread ID Register
TPIDR_EL2	EL2 Software Thread ID Register
TPIDR_EL3	EL3 Software Thread ID Register
TPIDRRO_EL0	EL0 Read-Only Software Thread ID Register
VBAR_EL1	Vector Base Address Register (EL1)
VBAR_EL2	Vector Base Address Register (EL2)
VBAR_EL3	Vector Base Address Register (EL3)

3.1.3 New registers added by Morello

Name	Description
CCTLR_EL0	Capability Control Register (EL0)
CCTLR_EL1	Capability Control Register (EL1)
CCTLR_EL2	Capability Control Register (EL2)
CCTLR_EL3	Capability Control Register (EL3)
CDBGDTR_EL0	Capability Debug Data Transfer Register, half-duplex
CDLR_EL0	Capability Debug Link Register
CHCR_EL2	Capability Hypervisor Configuration Register
CID_EL0	Compartment ID Register
DDC_EL0	Default Data Capability (EL0)
DDC_EL1	Default Data Capability (EL1)
DDC_EL2	Default Data Capability (EL2)
DDC_EL3	Default Data Capability (EL3)

Name	Description
RDDC_EL0	Restricted Default Data Capability
RSP_EL0	Restricted Stack Pointer
RTPIDR_EL0	Restricted Read/Write Software Thread ID Register

3.1.4 External registers

Name	Description
DBGDTR2A	Debug Data Transfer Register 2A
DBGDTR2B	Debug Data Transfer Register 2B
EDSCR2	External Debug Status and Control Register 2

3.2 Alphabetical list of registers

3.2.1 CCTLR_EL0, Capability Control Register (EL0)

The CCTLR_EL0 characteristics are:

Purpose

Provides control of capability-related functionality at EL0.

Attributes

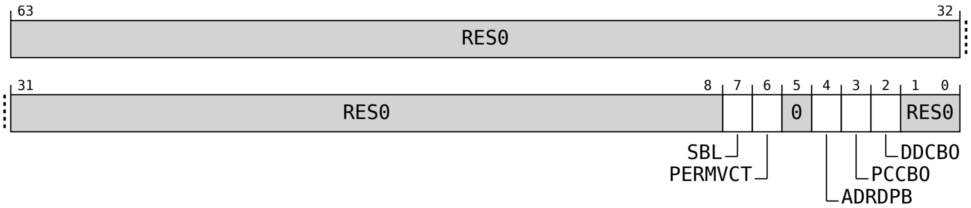
CCTLR_EL0 is a 64-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to CCTLR_EL0 are UNDEFINED.

Field descriptions

The CCTLR_EL0 bit assignments are:



Bits [63:8]

Reserved, RES0.

SBL, bit [7]

Controls whether branch-and-link instructions at EL0 seal the capability generated in C30.

Controls whether the following instructions at EL0 require a target capability with ObjectType set to 1:

BLRR, BLRS (capability), BRR, BRS (capability), RETR, RETS (capability).

Value	Meaning
`0b0`	Branch-and-link instructions which generate a capability in C30 do not seal the capability. The specified instructions do not require a target capability with ObjectType set to 1.
`0b1`	Branch-and-link instructions which generate a capability in C30 seal the generated capability with ObjectType set to 1. The specified instructions require a target capability with ObjectType set to 1.

This field resets to an architecturally UNKNOWN value.

PERMVCT, bit [6]

Permits access to CNTVCT_EL0 without PCC System permission at EL0

Value	Meaning
`0b0`	Access to CNTVCT_EL0 at EL0 requires PCC System permission
`0b1`	This field has no effect

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

ADRDPB, bit [4]

ADRDP instruction base register selection at EL0

Value	Meaning
`0b0`	ADRDP uses DDC as a base register
`0b1`	ADRDP uses C28 as a base register

This field resets to an architecturally UNKNOWN value.

PCCBO, bit [3]

PCC base offset enable for A64 instructions writing PC or generating a PC derived 64-bit value at EL0

Value	Meaning
`0b0`	Accesses do not add PCC base to the address written to PC, and do not subtract PCC base from the address read from PCC.
`0b1`	Accesses add PCC base to the address written to PC, and subtract PCC base from the address read from PCC.

Note: this affects the following instructions:

- BR Xn
- RET Xn
- BL imm (the value written to LR)
- BLR Xn (both the Xn and LR values)
- ADR(P) Xd, label

This field resets to an architecturally UNKNOWN value.

DDCBO, bit [2]

DDC base offset enable for accesses using a 64-bit base register at EL0

Value	Meaning
`0b0`	Accesses do not add or subtract DDC base from the accessed address.
`0b1`	Accesses add or subtract DDC base from the accessed address, depending on the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [1:0]

Reserved, RES0.

Accessing the CCTLR_EL0

Read using name CCTLR_EL0

The assembler syntax is:

```
MRS <Xt>, CCTLR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11'
      ↪then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL0;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
```

```
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL0;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL0;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL0;
```

Write using name CCTLR_EL0

The assembler syntax is:

```
MSR CCTLR_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11'
        ↪then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL0 = X[t];
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPACR_EL1.CEN == 'x0' then
```

```
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL0 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL0 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL0 = X[t];
```

3.2.2 CCTLR_EL1, Capability Control Register (EL1)

The CCTLR_EL1 characteristics are:

Purpose

Provides control of capability-related functionality at EL1.

Attributes

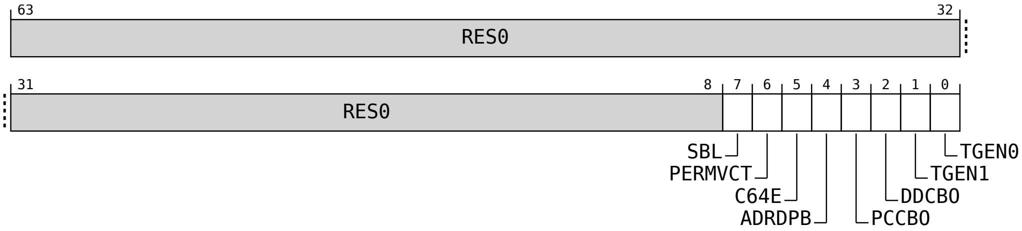
CCTLR_EL1 is a 64-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to CCTLR_EL1 are UNDEFINED.

Field descriptions

The CCTLR_EL1 bit assignments are:



Bits [63:8]

Reserved, RES0.

SBL, bit [7]

Controls whether branch-and-link instructions at EL1 seal the capability generated in C30.

Controls whether the following instructions at EL1 require a target capability with ObjectType set to 1:

BLRR, BLRS (capability), BRR, BRS (capability), RETR, RETS (capability).

Value	Meaning
`0b0`	Branch-and-link instructions which generate a capability in C30 do not seal the capability. The specified instructions do not require a target capability with ObjectType set to 1.
`0b1`	Branch-and-link instructions which generate a capability in C30 seal the generated capability with ObjectType set to 1. The specified instructions require a target capability with ObjectType set to 1.

This field resets to an architecturally UNKNOWN value.

PERMVCT, bit [6]

Permits access to CNTVCT_EL0 without PCC System permission at EL1

Value	Meaning
`0b0`	Access to CNTVCT_EL0 at EL1 requires PCC System permission
`0b1`	This field has no effect

This field resets to an architecturally UNKNOWN value.

C64E, bit [5]

Capability mode on exception entry to EL1

Value	Meaning
`0b0`	On exception entry PSTATE.C64 is set to 0.
`0b1`	On exception entry PSTATE.C64 is set to 1.

This field resets to `0b0`.

ADRDPB, bit [4]

ADRDP instruction base register selection at EL1

Value	Meaning
`0b0`	ADRDP uses DDC as a base register
`0b1`	ADRDP uses C28 as a base register

This field resets to an architecturally UNKNOWN value.

PCCBO, bit [3]

PCC base offset enable for A64 instructions writing PC or generating a PC derived 64-bit value at EL1

Value	Meaning
`0b0`	Accesses do not add PCC base to the address written to PC, and do not subtract PCC base from the address read from PCC.
`0b1`	Accesses add PCC base to the address written to PC, and subtract PCC base from the address read from PCC.

Note: this affects the following instructions:

- BR Xn
- RET Xn
- BL imm (the value written to LR)
- BLR Xn (both the Xn and LR values)
- ADR(P) Xd, label

This field resets to an architecturally UNKNOWN value.

DDCBO, bit [2]

DDC base offset enable for accesses using a 64-bit base register at EL1

Value	Meaning
`0b0`	Accesses do not add or subtract DDC base from the accessed address.
`0b1`	Accesses add or subtract DDC base from the accessed address, depending on the instruction.

This field resets to an architecturally UNKNOWN value.

TGEN1, bit [1]

Tag generation bit for TTBR1_EL1 based memory accesses

Value	Meaning
`0b0`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b11.
`0b1`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b10.

This field resets to an architecturally UNKNOWN value.

TGEN0, bit [0]

Tag generation bit for TTBR0_EL1 based memory accesses

Value	Meaning
`0b0`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b11.
`0b1`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b10.

This field resets to an architecturally UNKNOWN value.

Accessing the CCTLR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CCTLR_EL1 or CCTLR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name CCTLR_EL1

The assembler syntax is:

```
MRS <Xt>, CCTLR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        return CCTLR_EL2;
    else
        return CCTLR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL1;
```

Write using name CCTLR_EL1

The assembler syntax is:

```
MSR CCTLR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
```

```
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        CCTLR_EL2 = X[t];
    else
        CCTLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL1 = X[t];
```

Read using name CCTLR_EL12

The assembler syntax is:

```
MRS <Xt>, CCTLR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            return CCTLR_EL1;
    else
```

```
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        elsif CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            return CCTLR_EL1;
    else
        UNDEFINED;
```

Write using name CCTLR_EL12

The assembler syntax is:

```
MSR CCTLR_EL12, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            CCTLR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        elsif CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            CCTLR_EL1 = X[t];
    else
        UNDEFINED;
```

3.2.3 CCTLR_EL2, Capability Control Register (EL2)

The CCTLR_EL2 characteristics are:

Purpose

Provides control of capability-related functionality at EL2.

Attributes

CCTLR_EL2 is a 64-bit register.

Configuration

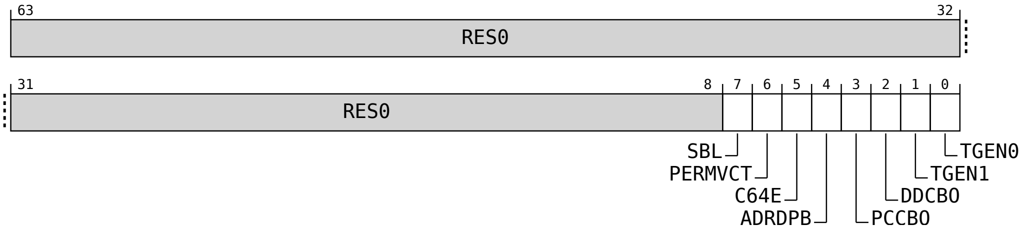
If EL2 is not implemented, this register is RES0 from EL3.

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

This register is present only when Morello is implemented. Otherwise, direct accesses to CCTLR_EL2 are UNDEFINED.

Field descriptions

The CCTLR_EL2 bit assignments are:



Bits [63:8]

Reserved, RES0.

SBL, bit [7]

Controls whether branch-and-link instructions at EL2 seal the capability generated in C30.

Controls whether the following instructions at EL2 require a target capability with ObjectType set to 1:

BLRR, BLRS (capability), BRR, BRS (capability), RETR, RETS (capability).

Value	Meaning
`0b0`	Branch-and-link instructions which generate a capability in C30 do not seal the capability. The specified instructions do not require a target capability with ObjectType set to 1.
`0b1`	Branch-and-link instructions which generate a capability in C30 seal the generated capability with ObjectType set to 1. The specified instructions require a target capability with ObjectType set to 1.

This field resets to an architecturally UNKNOWN value.

PERMVCT, bit [6]

Permits access to CNTVCT_EL0 without PCC System permission at EL2

Value	Meaning
`0b0`	Access to CNTVCT_EL0 at EL2 requires PCC System permission
`0b1`	This field has no effect

This field resets to an architecturally UNKNOWN value.

C64E, bit [5]

Capability mode on exception entry to EL2

Value	Meaning
`0b0`	On exception entry PSTATE.C64 is set to 0.
`0b1`	On exception entry PSTATE.C64 is set to 1.

This field resets to `0b0`.

ADRDPB, bit [4]

ADRDP instruction base register selection at EL2

Value	Meaning
`0b0`	ADRDP uses DDC as a base register
`0b1`	ADRDP uses C28 as a base register

This field resets to an architecturally UNKNOWN value.

PCCBO, bit [3]

PCC base offset enable for A64 instructions writing PC or generating a PC derived 64-bit value at EL2

Value	Meaning
`0b0`	Accesses do not add PCC base to the address written to PC, and do not subtract PCC base from the address read from PCC.
`0b1`	Accesses add PCC base to the address written to PC, and subtract PCC base from the address read from PCC.

Note: this affects the following instructions:

- BR Xn

- RET Xn
- BL imm (the value written to LR)
- BLR Xn (both the Xn and LR values)
- ADR(P) Xd, label

This field resets to an architecturally UNKNOWN value.

DDCBO, bit [2]

DDC base offset enable for accesses using a 64-bit base register at EL2

Value	Meaning
`0b0`	Accesses do not add or subtract DDC base from the accessed address.
`0b1`	Accesses add or subtract DDC base from the accessed address, depending on the instruction.

This field resets to an architecturally UNKNOWN value.

TGEN1, bit [1]

When ARMv8.1-VHE is implemented and HCR_EL2.E2H == 1:

Tag generation bit for TTBR1_EL2 based accesses

Value	Meaning
`0b0`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b11.
`0b1`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b10.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

TGEN0, bit [0]

Tag generation bit for TTBR0_EL2 based accesses

Value	Meaning
`0b0`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b11.
`0b1`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b10.

This field resets to an architecturally UNKNOWN value.

Accessing the CCTLR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic CCTLR_EL2 or CCTLR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name CCTLR_EL2

The assembler syntax is:

```
MRS <Xt>, CCTLR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL2;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL2;
```

Write using name CCTLR_EL2

The assembler syntax is:

```
MSR CCTLR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
```

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL2 = X[t];
```

Read using name CCTLR_EL1

The assembler syntax is:

```
MRS <Xt>, CCTLR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        return CCTLR_EL2;
    else
        return CCTLR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL1;
```

Write using name CCTLR_EL1

The assembler syntax is:

```
MSR CCTLR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        CCTLR_EL2 = X[t];
    else
        CCTLR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL1 = X[t];
```

3.2.4 CCTLR_EL3, Capability Control Register (EL3)

The CCTLR_EL3 characteristics are:

Purpose

Provides control of capability-related functionality at EL3.

Attributes

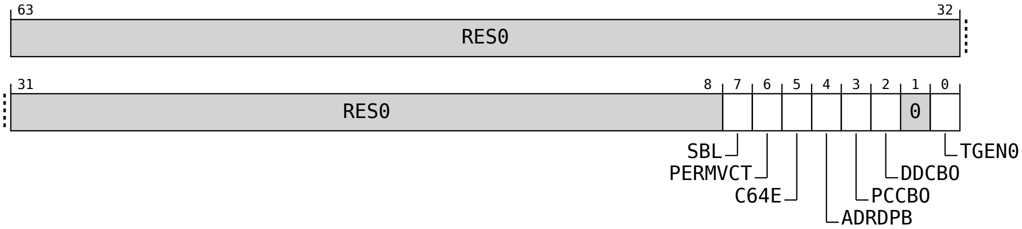
CCTLR_EL3 is a 64-bit register.

Configuration

This register is present only when Morello is implemented and HaveEL(EL3). Otherwise, direct accesses to CCTLR_EL3 are UNDEFINED.

Field descriptions

The CCTLR_EL3 bit assignments are:



Bits [63:8]

Reserved, RES0.

SBL, bit [7]

Controls whether branch-and-link instructions at EL3 seal the capability generated in C30.

Controls whether the following instructions at EL3 require a target capability with ObjectType set to 1:

BLRR, BLRS (capability), BRR, BRS (capability), RETR, RETS (capability).

Value	Meaning
`0b0`	Branch-and-link instructions which generate a capability in C30 do not seal the capability. The specified instructions do not require a target capability with ObjectType set to 1.
`0b1`	Branch-and-link instructions which generate a capability in C30 seal the generated capability with ObjectType set to 1. The specified instructions require a target capability with ObjectType set to 1.

This field resets to an architecturally UNKNOWN value.

PERMVCT, bit [6]

Permits access to CNTVCT_EL0 without PCC System permission at EL3

Value	Meaning
`0b0`	Access to CNTVCT_EL0 at EL3 requires PCC System permission
`0b1`	This field has no effect

This field resets to an architecturally UNKNOWN value.

C64E, bit [5]

Capability mode on exception entry to EL3

Value	Meaning
`0b0`	On exception entry PSTATE.C64 is set to 0.
`0b1`	On exception entry PSTATE.C64 is set to 1.

This field resets to `0b0`.

ADRDPB, bit [4]

ADRDP instruction base register selection at EL3

Value	Meaning
`0b0`	ADRDP uses DDC as a base register
`0b1`	ADRDP uses C28 as a base register

This field resets to an architecturally UNKNOWN value.

PCCBO, bit [3]

PCC base offset enable for A64 instructions writing PC or generating a PC derived 64-bit value at EL3

Value	Meaning
`0b0`	Accesses do not add PCC base to the address written to PC, and do not subtract PCC base from the address read from PCC.
`0b1`	Accesses add PCC base to the address written to PC, and subtract PCC base from the address read from PCC.

Note: this affects the following instructions:

- BR Xn
- RET Xn
- BL imm (the value written to LR)
- BLR Xn (both the Xn and LR values)
- ADR(P) Xd, label

This field resets to an architecturally UNKNOWN value.

DDCBO, bit [2]

DDC base offset enable for accesses using a 64-bit base register at EL3

Value	Meaning
`0b0`	Accesses do not add or subtract DDC base from the accessed address.
`0b1`	Accesses add or subtract DDC base from the accessed address, depending on the instruction.

This field resets to an architecturally UNKNOWN value.

Bit [1]

Reserved, RES0.

TGEN0, bit [0]

Tag generation bit for TTBR0_EL3 based accesses

Value	Meaning
`0b0`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b11.
`0b1`	Generates a fault when loading a valid capability from memory where the Block and Page descriptor LC field is 0b10.

This field resets to an architecturally UNKNOWN value.

Accessing the CCTLR_EL3

Read using name CCTLR_EL3

The assembler syntax is:

```
MRS <Xt>, CCTLR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
```

```
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CCTLR_EL3;
```

Write using name CCTLR_EL3

The assembler syntax is:

```
MSR CCTLR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CCTLR_EL3 = X[t];
```

3.2.5 CDBGDTR_EL0, Capability Debug Data Transfer Register, half-duplex

The CDBGDTR_EL0 characteristics are:

Purpose

Transfers 129 bits of data between the PE and an external debugger. Can transfer both ways using only a single register.

Attributes

CDBGDTR_EL0 is a 129-bit register.

Configuration

AArch64 System register CDBGDTR_EL0[63:0] is architecturally mapped to AArch64 System register DBGDTR_EL0[63:0].

AArch64 System register CDBGDTR_EL0[128] is architecturally mapped to External register EDSCR2[0].

AArch64 System register CDBGDTR_EL0[127:96] is architecturally mapped to External register DBGDTR2B[31:0].

AArch64 System register CDBGDTR_EL0[95:64] is architecturally mapped to External register DBGDTR2A[31:0].

AArch64 System register CDBGDTR_EL0[63:32] is architecturally mapped to AArch32 System register DBGDTRRXint[31:0]when written.

AArch64 System register CDBGDTR_EL0[63:32] is architecturally mapped to External register DBGDTRRX_EL0[31:0]when written.

AArch64 System register CDBGDTR_EL0[63:32] is architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0]when written.

AArch64 System register CDBGDTR_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRTXint[31:0]when written.

AArch64 System register CDBGDTR_EL0[31:0] is architecturally mapped to External register DBGDTRTX_EL0[31:0]when written.

AArch64 System register CDBGDTR_EL0[31:0] is architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0]when written.

AArch64 System register CDBGDTR_EL0[63:32] is architecturally mapped to AArch32 System register DBGDTRTXint[31:0]when read.

AArch64 System register CDBGDTR_EL0[63:32] is architecturally mapped to External register DBGDTRTX_EL0[31:0]when read.

AArch64 System register CDBGDTR_EL0[63:32] is architecturally mapped to AArch64 System register DBGDTRTX_EL0[31:0]when read.

AArch64 System register CDBGDTR_EL0[31:0] is architecturally mapped to AArch32 System register DBGDTRRXint[31:0]when read.

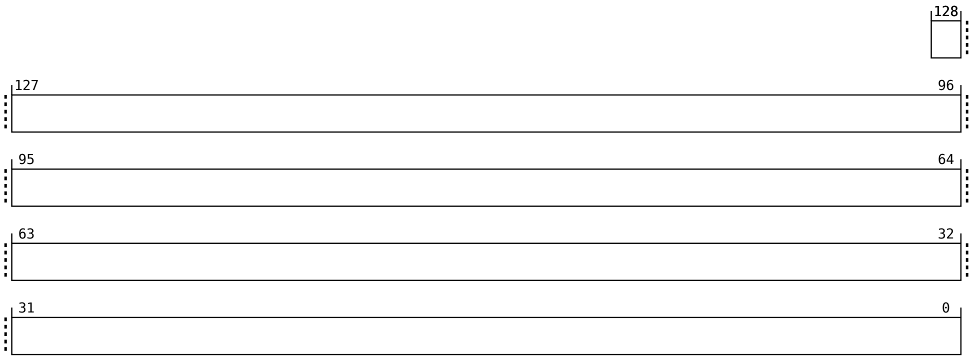
AArch64 System register CDBGDTR_EL0[31:0] is architecturally mapped to External register DBGDTRRX_EL0[31:0]when read.

AArch64 System register CDBGDTR_EL0[31:0] is architecturally mapped to AArch64 System register DBGDTRRX_EL0[31:0]when read.

This register is present only when Morello is implemented. Otherwise, direct accesses to CDBGDTR_EL0 are UNDEFINED.

Field descriptions

The CDBGDTR_EL0 bit assignments are:



Bits [128:0]

Writes to this register set:

- EDSCR2.DTRTAG to bit[128] of this field
- DTR2B to bits[127:96] of this field
- DTR2A to bits[95:64] of this field
- DTRRX to bits[63:32] of this field
- DTRTX to bits[31:0] of this field
- TXfull to 1

If RXfull is set to 1, reads of this register return:

- EDSCR2.DTRTAG in bit[128] of this field
- DTR2B in bits[127:96] of this field
- DTR2A in bits[95:64] of this field
- DTRTX in bits[63:32] of this field
- DTRRX in bits[31:0] of this field

If RXfull is set to 0, reads of this register return an UNKNOWN value.

After the read, RXfull is cleared to 0.

Accessing the CDBGDTR_EL0

Read using name CDBGDTR_EL0

The assembler syntax is:

```
MRS <Ct>, CDBGDTR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

Accessibility:

```
if !Halted() then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDBGDTR_EL0;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDBGDTR_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDBGDTR_EL0;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDBGDTR_EL0;
```

Write using name CDBGDTR_EL0

The assembler syntax is:

```
MSR CDBGDTR_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b10	0b011	0b0000	0b0100	0b000

Accessibility:

```
if !Halted() then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDBGDTR_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDBGDTR_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDBGDTR_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDBGDTR_EL0 = C[t];
```

3.2.6 CDLR_EL0, Capability Debug Link Register

The CDLR_EL0 characteristics are:

Purpose

In Debug state, holds the capability to restart from.

Attributes

CDLR_EL0 is a 129-bit register.

Configuration

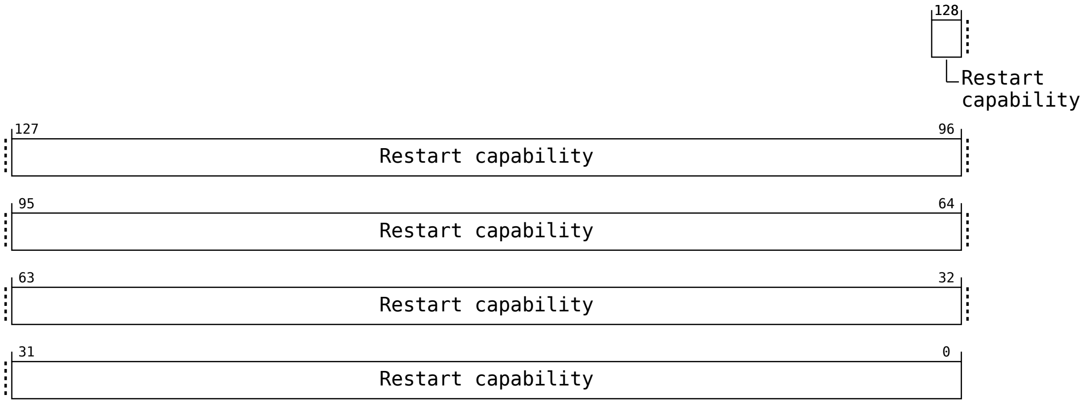
AArch64 System register CDLR_EL0[31:0] is architecturally mapped to AArch32 System register DLR[31:0].

AArch64 System register CDLR_EL0[63:0] is architecturally mapped to AArch64 System register DLR_EL0[63:0].

This register is present only when Morello is implemented. Otherwise, direct accesses to CDLR_EL0 are UNDEFINED.

Field descriptions

The CDLR_EL0 bit assignments are:



Bits [128:0]

Restart capability.

Accessing the CDLR_EL0

Read using name CDLR_EL0

The assembler syntax is:

```
MRS <Ct>, CDLR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

Accessibility:

```
if !Halted() then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDLR_EL0;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDLR_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDLR_EL0;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CDLR_EL0;
```

Write using name CDLR_EL0

The assembler syntax is:

```
MSR CDLR_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b001

Accessibility:

```
if !Halted() then
    UNDEFINED;
elsif PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDLR_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDLR_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDLR_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CDLR_EL0 = C[t];
```

3.2.7 CHCR_EL2, Capability Hypervisor Configuration Register

The CHCR_EL2 characteristics are:

Purpose

Provides control over privileged access to capabilities

Attributes

CHCR_EL2 is a 64-bit register.

Configuration

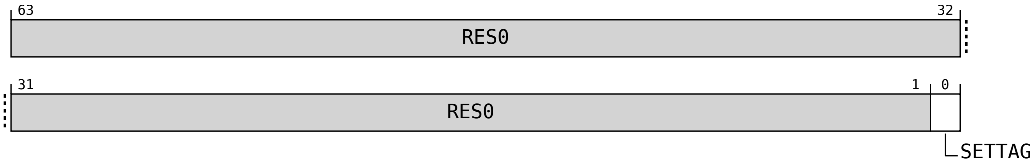
If EL2 is not implemented, this register is RES0 from EL3.

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

This register is present only when Morello is implemented. Otherwise, direct accesses to CHCR_EL2 are UNDEFINED.

Field descriptions

The CHCR_EL2 bit assignments are:



Bits [63:1]

Reserved, RES0.

SETTAG, bit [0]

Access to privileged capability creating instructions, SCTAG and STCT.

Value	Meaning
`0b0`	No effect.
`0b1`	Privileged capability creating instructions clear the tag if executed at EL1.

This field resets to an architecturally UNKNOWN value.

Accessing the CHCR_EL2

Read using name CHCR_EL2

The assembler syntax is:

```
MRS <Xt>, CHCR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CHCR_EL2;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CHCR_EL2;
```

Write using name CHCR_EL2

The assembler syntax is:

```
MSR CHCR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0001	0b0010	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CHCR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
```

```
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CHCR_EL2 = X[t];
```

3.2.8 CID_EL0, Compartment ID Register

The CID_EL0 characteristics are:

Purpose

Provides a number that can be used to separate out different context numbers with each Exception level.

Attributes

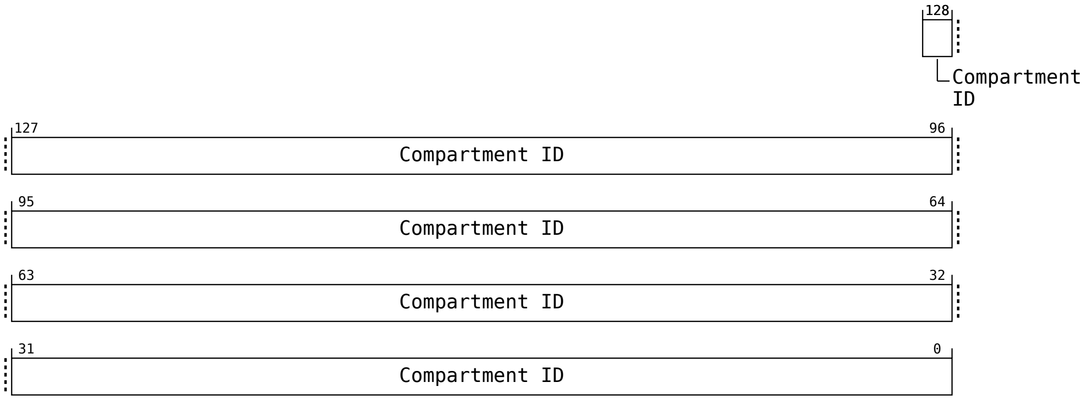
CID_EL0 is a 129-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to CID_EL0 are UNDEFINED.

Field descriptions

The CID_EL0 bit assignments are:



Bits [128:0]

Compartment ID

This field resets to an architecturally UNKNOWN value.

Accessing the CID_EL0

Read using name CID_EL0

The assembler syntax is:

```
MRS <Ct>, CID_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

Accessibility:

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CID_EL0;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CID_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CID_EL0;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CID_EL0;
```

Write using name CID_EL0

The assembler syntax is:

```
MSR CID_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b111

Accessibility:

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CID_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CID_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CID_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CID_EL0 = C[t];
```

3.2.9 CNTVCT_EL0, Counter-timer Virtual Count register

The CNTVCT_EL0 characteristics are:

Purpose

Holds the 64-bit virtual count value. The virtual count value is equal to the physical count value minus the virtual offset visible in CNTVOFF_EL2.

Attributes

CNTVCT_EL0 is a 64-bit register.

Configuration

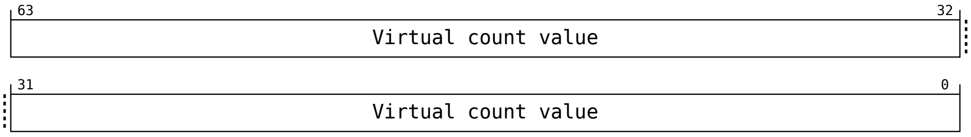
The value of this register is the same as the value of CNTPCT_EL0 in the following conditions:

- When EL2 is not implemented.
- When EL2 is implemented, HCR_EL2.E2H is 1, and this register is read from EL2.
- When EL2 is implemented and enabled in the current Security state, HCR_EL2.{E2H, TGE} is {1, 1}, and this register is read from EL0 or EL2.

AArch64 System register CNTVCT_EL0[63:0] is architecturally mapped to AArch32 System register CNTVCT[63:0].

Field descriptions

The CNTVCT_EL0 bit assignments are:



Bits [63:0]

Virtual count value.

Accessing the CNTVCT_EL0

Read using name CNTVCT_EL0

The assembler syntax is:

```
MRS <Xt>, CNTVCT_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1110	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && CCTLR_EL0.PERMVCT == '0' && !CapIsSystemAccessEnabled() &&
        ↪!Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
```

```
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CNTKCTL_EL1.EL0VCTEN == '0' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL1, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CNTHCTL_EL2.EL0VCTEN == '0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return CNTVCT_EL0;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && CCTLR_EL1.PERMVCT == '0' && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CNTVCT_EL0;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && CCTLR_EL2.PERMVCT == '0' && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CNTVCT_EL0;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && CCTLR_EL3.PERMVCT == '0' && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CNTVCT_EL0;
```

3.2.10 CPACR_EL1, Architectural Feature Access Control Register

The CPACR_EL1 characteristics are:

Purpose

Controls access to trace, SVE, Advanced SIMD and floating-point, and the Morello architecture.

Attributes

CPACR_EL1 is a 64-bit register.

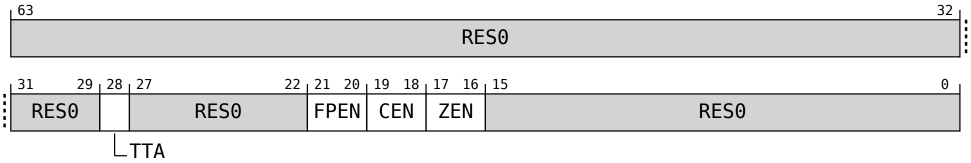
Configuration

When HCR_EL2.{E2H, TGE} == {1, 1}, the fields in this register have no effect on execution at EL0 and EL1. In this case, the controls provided by CPTR_EL2 are used.

AArch64 System register CPACR_EL1[31:0] is architecturally mapped to AArch32 System register CPACR[31:0].

Field descriptions

The CPACR_EL1 bit assignments are:



Bits [63:29]

Reserved, RES0.

TTA, bit [28]

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

- In AArch64 state, accesses to trace registers are trapped, reported using EC syndrome value 0x18.
- In AArch32 state, MRC and MCR accesses to trace registers are trapped, reported using EC syndrome value 0x05.
- In AArch32 state, MRRC and MCRR accesses to trace registers are trapped, reported using EC syndrome value 0x0C.

Value	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	This control causes EL0 and EL1 System register accesses to all implemented trace registers to be trapped.

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.TTA is 1.
- The Armv8-A architecture does not provide traps on trace register accesses through the optional

memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not implemented, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

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

- In AArch64 state, accesses to FPCR, FPSR, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers. See x'The SIMD and floating-point registers, V0-V31'.
- FPSCR, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. See x'Advanced SIMD and floating-point System registers'.

Value	Meaning
`0b00`	This control causes any instructions at EL0 or EL1 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN.
`0b01`	This control causes any instructions at EL0 that use the registers associated with SVE, Advanced SIMD and floating- point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN, but does not cause any instruction at EL1 to be trapped.
`0b10`	This control causes any instructions at EL0 or EL1 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPACR_EL1.ZEN.
`0b11`	This control does not cause any instructions to be trapped.

Writes to MVFR0, MVFR1 and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPACR_EL1.FPEN is not 0b11.

This field resets to an architecturally UNKNOWN value.

CEN, bits [19:18]

When Morello is implemented:

Traps Morello instructions and instructions that access Morello System registers at EL0 and EL1 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

Value	Meaning
0b00	This control causes these instructions executed at EL0 or EL1 to be trapped.
0b01	This control causes these instructions executed at EL0 to be trapped, but does not cause any instructions at EL1 to be trapped.
0b10	This control causes these instructions executed at EL0 or EL1 to be trapped.
0b11	This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ZEN, bits [17:16]

When SVE is implemented:

Traps SVE instructions and instructions that access SVE System registers at EL0 and EL1 to EL1, or to EL2 when it is implemented and enabled for the current Security state and HCR_EL2.TGE is 1.

Value	Meaning
0b00	This control causes these instructions executed at EL0 or EL1 to be trapped.
0b01	This control causes these instructions executed at EL0 to be trapped, but does not cause any instruction at EL1 to be trapped.
0b10	This control causes these instructions executed at EL0 or EL1 to be trapped.
0b11	This control does not cause any instruction to be trapped.

If xSVEis not implemented, this field is RES0.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [15:0]

Reserved, RES0.

Accessing the CPACR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic CPACR_EL1 or CPACR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name CPACR_EL1

The assembler syntax is:

```
MRS <Xt>, CPACR_EL1
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return CPTR_EL2;
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CPACR_EL1;
```

Write using name CPACR_EL1

The assembler syntax is:

```
MSR CPACR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        CPTR_EL2 = X[t];
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPACR_EL1 = X[t];
```

Read using name CPACR_EL12

The assembler syntax is:

```
MRS <Xt>, CPACR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return CPACR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return CPACR_EL1;
    else
        UNDEFINED;
```

Write using name CPACR_EL12

The assembler syntax is:

```
MSR CPACR_EL12, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0001	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            CPACR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            CPACR_EL1 = X[t];
    else
        UNDEFINED;
```

3.2.11 CPTR_EL2, Architectural Feature Trap Register (EL2)

The CPTR_EL2 characteristics are:

Purpose

Controls:

- Trapping to EL2 of access to CPACR, CPACR_EL1, trace functionality, SVE, Advanced SIMD and floating-point functionality, and to the Morello architecture.
- EL2 access to trace functionality, SVE, Advanced SIMD and floating-point functionality, and to the Morello architecture.

Attributes

CPTR_EL2 is a 64-bit register.

Configuration

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

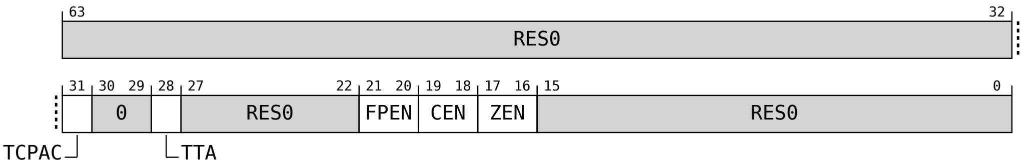
This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register CPTR_EL2[31:0] is architecturally mapped to AArch32 System register HCPTR[31:0].

Field descriptions

The CPTR_EL2 bit assignments are:

When ARMv8.1-VHE is implemented and HCR_EL2.E2H == 1:



Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

When HCR_EL2.TGE is 0, traps EL1 accesses to CPACR_EL1 reported using EC syndrome value 0x18, and accesses to CPACR reported using EC syndrome value 0x03, to EL2 when EL2 is enabled in the current Security state.

Value	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL1 accesses to CPACR_EL1 and CPACR are trapped to EL2 when EL2 is enabled in the current Security state.

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

CPACR_EL1 and CPACR are not accessible at EL0.

This field resets to an architecturally UNKNOWN value.

Bit [30:29]

Reserved, RES0.

TTA, bit [28]

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

- In AArch64 state, accesses to trace registers with op0=2, op1=1 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1, are trapped to EL2, reported using EC syndrome value 0x05.
- In AArch32 state, MRRC or MCRR accesses to trace registers with cpnum=14, opc1=1, are trapped to EL2, reported using EC syndrome value 0x0C.

Value	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt at EL0, EL1 or EL2, to execute a System register access to an implemented trace register is trapped to EL2 when EL2 is enabled in the current Security state, unless HCR_EL2.TGE is 0 and it is trapped by CPACR.NSTRCDIS or CPACR_EL1.TTA. When HCR_EL2.TGE is 1, any attempt at EL0 or EL2 to execute a System register access to an implemented trace register is trapped to EL2 when EL2 is enabled in the current Security state.

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [27:22]

Reserved, RES0.

FPEN, bits [21:20]

Traps EL0, EL2 and, when HCR_EL2.TGE is 0, EL1 accesses to the SVE, Advanced SIMD and floating-point registers to EL2 when EL2 is enabled in the current Security state, from both Execution states.

Value	Meaning
0b00	This control causes any instructions at EL0, EL1, or EL2 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, subject to the exception prioritization rules, unless they are trapped by CPTR_EL2.ZEN.
0b01	When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes instructions at EL0 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, unless they are trapped by CPTR_EL2.ZEN, but does not cause any instruction at EL2 to be trapped.
0b10	This control causes any instructions at EL0, EL1, or EL2 that use the registers associated with SVE, Advanced SIMD and floating-point execution to be trapped, subject to the exception prioritization rules, unless they are trapped by CPTR_EL2.ZEN.
0b11	This control does not cause any instructions to be trapped.

Writes to MVFR0, MVFR1, and MVFR2 from EL1 or higher are CONSTRAINED UNPREDICTABLE and whether these accesses can be trapped by this control depends on implemented CONSTRAINED UNPREDICTABLE behavior.

- Attempts to write to the FPSID count as use of the registers for accesses from EL1 or higher.
- Accesses from EL0 to FPSID, MVFR0, MVFR1, MVFR2, and FPEXC are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.FPEN is not 0b11.

This field resets to an architecturally UNKNOWN value.

CEN, bits [19:18]

When Morello is implemented:

Traps execution at EL2, EL1, and EL0 of Morello instructions or instructions that access Morello System registers to EL2 when EL2 is enabled in the current Security state.

Value	Meaning
0b00	This control causes execution at EL2, EL1, and EL0 of Morello instructions to be trapped, subject to the exception prioritization rules.
0b01	When HCR_EL2.TGE is 0, this control does not cause any instructions to be trapped. When HCR_EL2.TGE is 1, this control causes these instructions executed at EL0 to be trapped, but does not cause any instructions at EL2 to be trapped.
0b10	This control causes execution at EL2, EL1, and EL0 of these instructions to be trapped, subject to the exception prioritization rules.
0b11	This control does not cause any instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

ZEN, bits [17:16]

When SVE is implemented:

Traps execution at EL2, EL1, and EL0 of SVE instructions or instructions that access SVE System registers to EL2 when EL2 is enabled in the current Security state.

Value	Meaning
`0b00`	This control causes execution at EL2, EL1, and EL0 of these instructions to be trapped, subject to the exception prioritization rules.
`0b01`	When HCR_EL2.TGE is 0, this control does not cause any instruction to be trapped. When HCR_EL2.TGE is 1, this control causes these instructions executed at EL0 to be trapped, but does not cause any instruction at EL2 to be trapped.
`0b10`	This control causes execution at EL2, EL1, and EL0 of these instructions to be trapped, subject to the exception prioritization rules.
`0b11`	This control does not cause any instruction to be trapped.

This field resets to an architecturally UNKNOWN value.

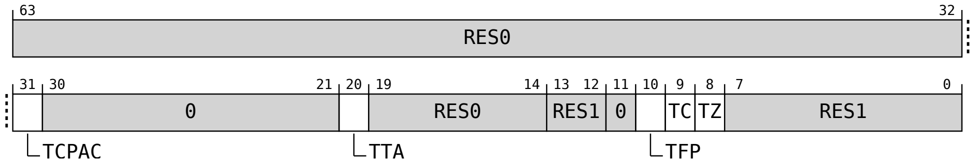
Otherwise:

RES0

Bits [15:0]

Reserved, RES0.

Otherwise:



This format applies in all Armv8.0 implementations.

Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

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

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

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

When HCR_EL2.TGE is 1, this control does not cause any instructions to be trapped.

CPACR_EL1 and CPACR are not accessible at EL0.

This field resets to an architecturally UNKNOWN value.

Bit [30:21]

Reserved, RES0.

TTA, bit [20]

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

- In AArch64 state, accesses to trace registers with op0=2, op1=1 are trapped to EL2, reported using EC syndrome value 0x18.
- In AArch32 state, MRC or MCR accesses to trace registers with cpnum=14, opc1=1 are trapped to EL2, reported using EC syndrome value 0x05.
- In AArch32 state, MRRC or MCRR accesses to trace registers with cpnum=14, opc1=1 are trapped to EL2, reported using EC syndrome value 0x0C.

Value	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt at EL0, EL1, or EL2, to execute a System register access to an implemented trace register is trapped to EL2 when EL2 is enabled in the current Security state, unless it is trapped by CPACR.TRCDIS or CPACR_EL1.TTA.

- The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than an exception that would be generated because the value of CPTR_EL2.TTA is 1.
- EL2 does not provide traps on trace register accesses through the optional memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, any side-effects that are normally associated with the access do not occur before the exception is taken.

If System register access to the trace functionality is not supported, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bits [19:14]

Reserved, RES0.

Bits [13:12]

Reserved, RES1.

Bit [11]

Reserved, RES0.

TFP, bit [10]

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

- In AArch64 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x07:
  - FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers. See x'The SIMD and floating-point registers, V0-V31'.
- In AArch32 state, accesses to the following registers are trapped to EL2, reported using EC syndrome value 0x07:
  - MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. See x'Advanced SIMD and floating-point System registers'. For the purposes of this trap, the architecture defines a VMSR access to FPSID from EL1 or higher as an access to a SIMD and floating point register. Otherwise, permitted VMSR accesses to FPSID are ignored.

Value	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any attempt at EL0, EL1 or EL2, to execute an instruction that uses the registers associated with SVE, Advanced SIMD and floating-point execution is trapped to EL2 when EL2 is enabled in the current Security state, subject to the exception prioritization rules, unless it is trapped by CPTR_EL2.TZ.

FPEXC32_EL2 is not accessible from EL0 using AArch64.

FPSID, MRFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

This field resets to an architecturally UNKNOWN value.

TC, bit [9]

When Morello is implemented:

Traps execution at EL2, EL1, or EL0 of Morello instructions and instructions that access Morello System registers to EL2 when EL2 is enabled in the current Security state.

Value	Meaning
`0b0`	Does not cause Morello instructions to be trapped.
`0b1`	Causes Morello instructions to be trapped.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

TZ, bit [8]

When SVE is implemented:

Traps execution at EL2, EL1, or EL0 of SVE instructions and instructions that access SVE System registers to EL2 when EL2 is enabled in the current Security state.

Value	Meaning
`0b0`	This control does not cause any instruction to be trapped.
`0b1`	This control causes these instructions to be trapped, subject to the exception prioritization rules.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES1

Bits [7:0]

Reserved, RES1.

Accessing the CPTR_EL2

Read using name CPTR_EL2

The assembler syntax is:

```
MRS <Xt>, CPTR_EL2
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CPTR_EL2;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```

```
        return CPTR_EL2;
```

Write using name CPTR_EL2

The assembler syntax is:

`MSR CPTR_EL2, <Xt>`

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPTR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPTR_EL2 = X[t];
```

Read using name CPACR_EL1

The assembler syntax is:

`MRS <Xt>, CPACR_EL1`

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return CPTR_EL2;
    else
        return CPACR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CPACR_EL1;
```

Write using name CPACR_EL1

The assembler syntax is:

```
MSR CPACR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && CPTR_EL2.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.TCPAC == '1' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        CPTR_EL2 = X[t];
    else
        CPACR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPACR_EL1 = X[t];
```

3.2.12 CPTR_EL3, Architectural Feature Trap Register (EL3)

The CPTR_EL3 characteristics are:

Purpose

Controls:

- Trapping to EL3 of access to CPACR_EL1, CPTR_EL2, trace functionality, SVE, Advanced SIMD and floating-point functionality, and to the Morello architecture.
- EL3 access to trace functionality, SVE, Advanced SIMD and floating-point functionality, and to the Morello architecture.

Attributes

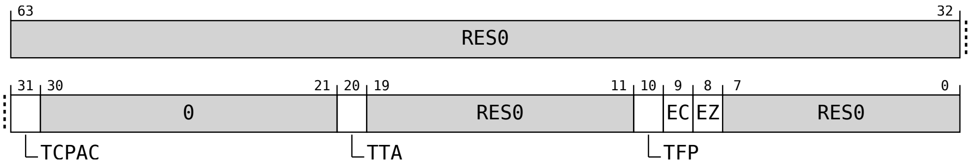
CPTR_EL3 is a 64-bit register.

Configuration

This register is present only when HaveEL(EL3). Otherwise, direct accesses to CPTR_EL3 are UNDEFINED.

Field descriptions

The CPTR_EL3 bit assignments are:



Bits [63:32]

Reserved, RES0.

TCPAC, bit [31]

Traps all of the following to EL3, from both Security states and both Execution states.

- EL2 accesses to CPTR_EL2, reported using EC syndrome value 0x18, or HCPTR, reported using EC syndrome value 0x03.
- EL2 and EL1 accesses to CPACR_EL1 reported using EC syndrome value 0x18, or CPACR reported using EC syndrome value 0x03.

When CPTR_EL3.TCPAC is:

Value	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	EL2 accesses to the CPTR_EL2 or HCPTR, and EL2 and EL1 accesses to the CPACR_EL1 or CPACR, are trapped to EL3, unless they are trapped by CPTR_EL2.TCPAC.

This field resets to an architecturally UNKNOWN value.

Bit [30:21]

Reserved, RES0.

TTA, bit [20]

Traps System register accesses. Accesses to the trace registers, from all Exception levels, both Security states, and both Execution states are trapped to EL3 as follows:

- In AArch64 state, Trace registers with op0=2, op1=1, are trapped to EL3 and reported using EC syndrome value 0x18.
- In AArch32 state, accesses using MCR or MRC to the Trace registers with cpnum=14 and opc1=1 are reported using EC syndrome value 0x05.
- In AArch32 state, accesses using MCRR or MRRC to the Trace registers with cpnum=14 and opc1=1 are reported using EC syndrome value 0x0C.

Value	Meaning
`0b0`	This control does not cause any instructions to be trapped.
`0b1`	Any System register access to the trace registers is trapped to EL3, subject to the exception prioritization rules, unless it is trapped by CPACR.TRCDIS, CPACR_EL1.TTA or CPTR_EL2.TTA.

If System register access to trace functionality is not supported, this bit is RES0.

The ETMv4 architecture does not permit EL0 to access the trace registers. If the Armv8-A architecture is implemented with an ETMv4 implementation, EL0 accesses to the trace registers are UNDEFINED, and any resulting exception is higher priority than this trap exception.

EL3 does not provide traps on trace register accesses through the Memory-mapped interface.

System register accesses to the trace registers can have side-effects. When a System register access is trapped, no side-effects occur before the exception is taken, see x'Traps on instructions'.

This field resets to an architecturally UNKNOWN value.

Bits [19:11]

Reserved, RES0.

TFP, bit [10]

Traps all accesses to SVE, Advanced SIMD and floating-point functionality, from all Exception levels, both Security states, and both Execution states, to EL3. Defined values are:

This includes the following registers, all reported using EC syndrome value 0x07:

- FPCR, FPSR, FPEXC32_EL2, any of the SIMD and floating-point registers V0-V31, including their views as D0-D31 registers or S0-31 registers. See x'The SIMD and floating-point registers, V0-V31'.
- MVFR0, MVFR1, MVFR2, FPSCR, FPEXC, and any of the SIMD and floating-point registers Q0-15, including their views as D0-D31 registers or S0-31 registers. See x'Advanced SIMD and floating-point System registers'.

Permitted VMSR accesses to FPSID are ignored, but for the purposes of this trap the architecture define a VMSR access to the FPSID from EL1 or higher as an access to a SIMD and floating-point register.

Value	Meaning
0b0	This control does not cause any instructions to be trapped.
0b1	Any attempt at any Exception level to execute an instruction that uses the registers associated with SVE, Advanced SIMD and floating-point is trapped to EL3, subject to the exception prioritization rules, unless it is trapped by CPTR_EL3.EZ.

FPEXC32_EL2 is not accessible from EL0 using AArch64.

FPSID, MRFR0, MVFR1, and FPEXC are not accessible from EL0 using AArch32.

This field resets to an architecturally UNKNOWN value.

EC, bit [9]

When Morello is implemented:

Traps all accesses to the Morello architecture and registers from all Exception levels, and both Security states, to EL3.

Value	Meaning
0b0	This control causes these instructions executed at any Exception level to be trapped, subject to the exception prioritization rules.
0b1	This control does not cause any instructions to be trapped.

This field resets to 0b0.

Otherwise:

RES0

EZ, bit [8]

When SVE is implemented:

Traps all accesses to SVE functionality and registers from all Exception levels, and both Security states, to EL3.

Value	Meaning
0b0	This control causes these instructions executed at any Exception level to be trapped, subject to the exception prioritization rules.
0b1	This control does not cause any instruction to be trapped.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [7:0]

Reserved, RES0.

Accessing the CPTR_EL3

Read using name CPTR_EL3

The assembler syntax is:

```
MRS <Xt>, CPTR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return CPTR_EL3;
```

Write using name CPTR_EL3

The assembler syntax is:

```
MSR CPTR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0001	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        CPTR_EL3 = X[t];
```

3.2.13 CSCR_EL3, Capability Secure Configuration Register

The CSCR_EL3 characteristics are:

Purpose

Provides control over privileged access to capabilities

Attributes

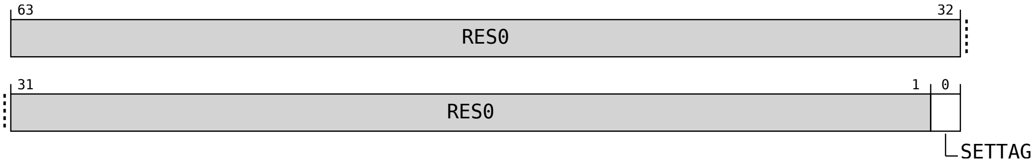
CSCR_EL3 is a 64-bit register.

Configuration

This register is present only when Morello is implemented and HaveEL(EL3). Otherwise, direct accesses to CSCR_EL3 are UNDEFINED.

Field descriptions

The CSCR_EL3 bit assignments are:



Bits [63:1]

Reserved, RES0.

SETTAG, bit [0]

Access to privileged capability creating instructions, SCTAG and STCT.

Value	Meaning
`0b0`	No effect.
`0b1`	Privileged capability creating instructions clear the tag if executed at EL2 or EL1.

This field resets to an architecturally UNKNOWN value.

Accessing the CSCR_EL3

Read using name CSCR_EL3

The assembler syntax is:

```
MRS <Xt>, CSCR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return CSCR_EL3;
```

Write using name CSCR_EL3

The assembler syntax is:

```
MSR CSCR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0001	0b0010	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        CSCR_EL3 = X[t];
```

3.2.14 DBGDTR2A, Debug Data Transfer Register 2A

The DBGDTR2A characteristics are:

Purpose

Allows external debuggers to access capability state within PE. Transfers lower 32 bits of the upper half of capabilities. It is a component of the Debug Communications Channel.

Attributes

DBGDTR2A is a 32-bit register.

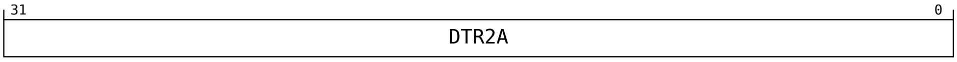
Configuration

External register DBGDTR2A[31:0] is architecturally mapped to AArch64 System register CDBGDTR_EL0[95:64].

This register is present only when Morello is implemented. Otherwise, direct accesses to DBGDTR2A are RES0.

Field descriptions

The DBGDTR2A bit assignments are:



Bits [31:0]

Data transfer register for bits 95:64 of capability tranfers.

On a cold reset, this field resets to an UNKNOWN value.

Accessing the DBGDTR2A

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTR2A can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x040	DBGDTR2A

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() access to this register is **RO**.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() access to this register is **RW**.
- Otherwise access to this register returns an ERROR.

3.2.15 DBGDTR2B, Debug Data Transfer Register 2B

The DBGDTR2B characteristics are:

Purpose

Allows external debuggers to access capability state within PE. Transfers higher 32 bits of the upper half of capabilities. It is a component of the Debug Communications Channel.

Attributes

DBGDTR2B is a 32-bit register.

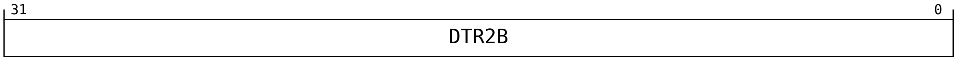
Configuration

External register DBGDTR2B[31:0] is architecturally mapped to AArch64 System register CDBGDTR_EL0[127:96].

This register is present only when Morello is implemented. Otherwise, direct accesses to DBGDTR2B are RES0.

Field descriptions

The DBGDTR2B bit assignments are:



Bits [31:0]

Data transfer register for bits 127:96 of capability tranfers.

On a cold reset, this field resets to an UNKNOWN value.

Accessing the DBGDTR2B

If EDSCR.ITE == 0 when the PE exits Debug state on receiving a Restart request trigger event, the behavior of any operation issued by a DTR access in memory access mode that has not completed execution is CONSTRAINED UNPREDICTABLE, and must do one of the following:

- It must complete execution in Debug state before the PE executes the restart sequence.
- It must complete execution in Non-debug state before the PE executes the restart sequence.
- It must be abandoned. This means that the instruction does not execute. Any registers or memory accessed by the instruction are left in an UNKNOWN state.

DBGDTR2B can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x044	DBGDTR2B

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() access to this register is **RO**.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() access to this register is **RW**.
- Otherwise access to this register returns an ERROR.

3.2.16 DDC_EL0, Default Data Capability (EL0)

The DDC_EL0 characteristics are:

Purpose

Holds the default data capability associated with EL0 when the PE is in Executive.

Attributes

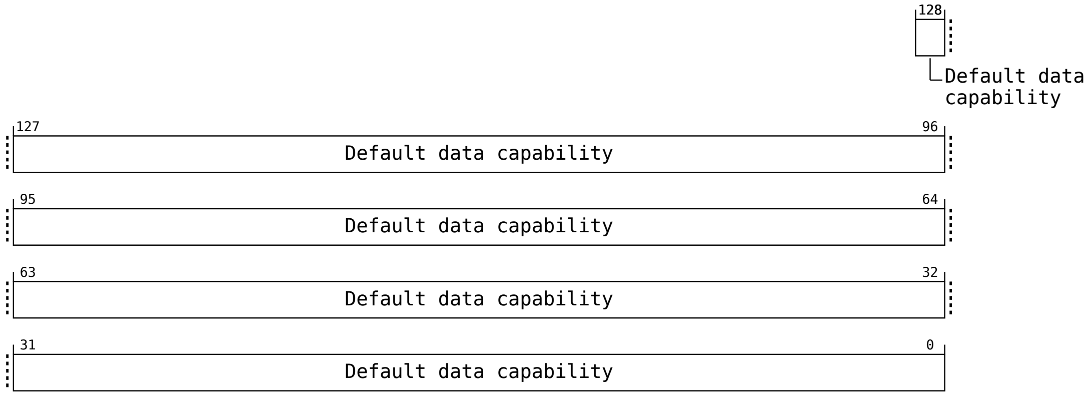
DDC_EL0 is a 129-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to DDC_EL0 are UNDEFINED.

Field descriptions

The DDC_EL0 bit assignments are:



Bits [128:0]

Default data capability.

This field resets to `0x1FFFFC0000001000500000000000000000`.

Accessing the DDC_EL0

Read using name DDC_EL0

The assembler syntax is:

```
MRS <Ct>, DDC_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
```

```
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return DDC_EL0;
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return DDC_EL0;
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return DDC_EL0;
```

Write using name DDC_EL0

The assembler syntax is:

```
MSR DDC_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        DDC_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        DDC_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        DDC_EL0 = C[t];
```

Read using name DDC

The assembler syntax is:

```
MRS <Ct>, DDC
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    return RDDC_EL0;
elsif PSTATE.SP == '0' then
    return DDC_EL0;
elsif PSTATE.EL == EL0 then
    return DDC_EL0;
elsif PSTATE.EL == EL1 then
    return DDC_EL1;
elsif PSTATE.EL == EL2 then
    return DDC_EL2;
elsif PSTATE.EL == EL3 then
    return DDC_EL3;
```

Write using name DDC

The assembler syntax is:

```
MSR DDC, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    RDDC_EL0 = C[t];
elsif PSTATE.SP == '0' then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL0 then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    DDC_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    DDC_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    DDC_EL3 = C[t];
```

3.2.17 DDC_EL1, Default Data Capability (EL1)

The DDC_EL1 characteristics are:

Purpose

Holds the default data capability associated with EL1 when the PE is in Executive.

Attributes

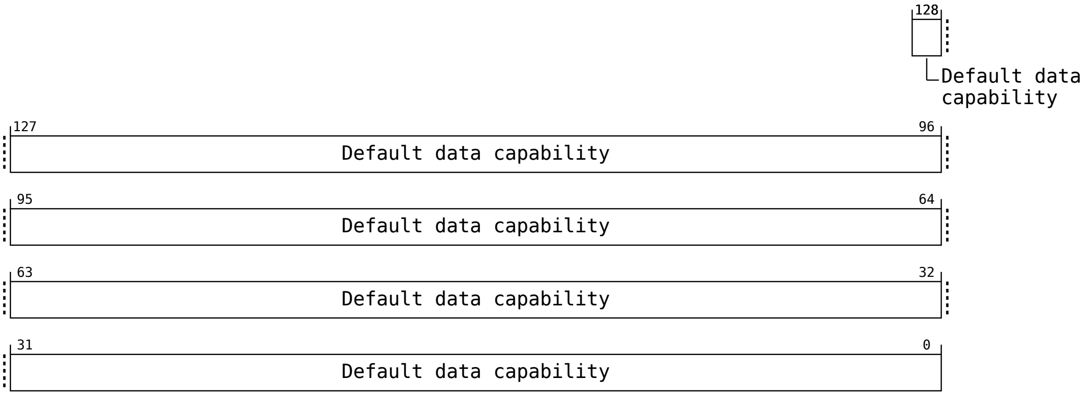
DDC_EL1 is a 129-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to DDC_EL1 are UNDEFINED.

Field descriptions

The DDC_EL1 bit assignments are:



Bits [128:0]

Default data capability.

This field resets to `0x1FFFFC000000100050000000000000000`.

Accessing the DDC_EL1

Read using name DDC_EL1

The assembler syntax is:

```
MRS <Ct>, DDC_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b001

Accessibility:

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

```
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return DDC_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return DDC_EL1;
```

Write using name DDC_EL1

The assembler syntax is:

```
MSR DDC_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        DDC_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        DDC_EL1 = C[t];
```

Read using name DDC

The assembler syntax is:

```
MRS <Ct>, DDC
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') &&
    ↪CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' &&
    ↪CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' &&
    ↪CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' &&
    ↪CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    return RDDC_EL0;
elsif PSTATE.SP == '0' then
    return DDC_EL0;
elsif PSTATE.EL == EL0 then
    return DDC_EL0;
elsif PSTATE.EL == EL1 then
    return DDC_EL1;
elsif PSTATE.EL == EL2 then
    return DDC_EL2;
elsif PSTATE.EL == EL3 then
    return DDC_EL3;
```

Write using name DDC

The assembler syntax is:

```
MSR DDC, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') &&
    ↪CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' &&
    ↪CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' &&
    ↪CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' &&
    ↪CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    RDDC_EL0 = C[t];
elsif PSTATE.SP == '0' then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL0 then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    DDC_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    DDC_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    DDC_EL3 = C[t];
```

3.2.18 DDC_EL2, Default Data Capability (EL2)

The DDC_EL2 characteristics are:

Purpose

Holds the default data capability associated with EL2 when the PE is in Executive.

Attributes

DDC_EL2 is a 129-bit register.

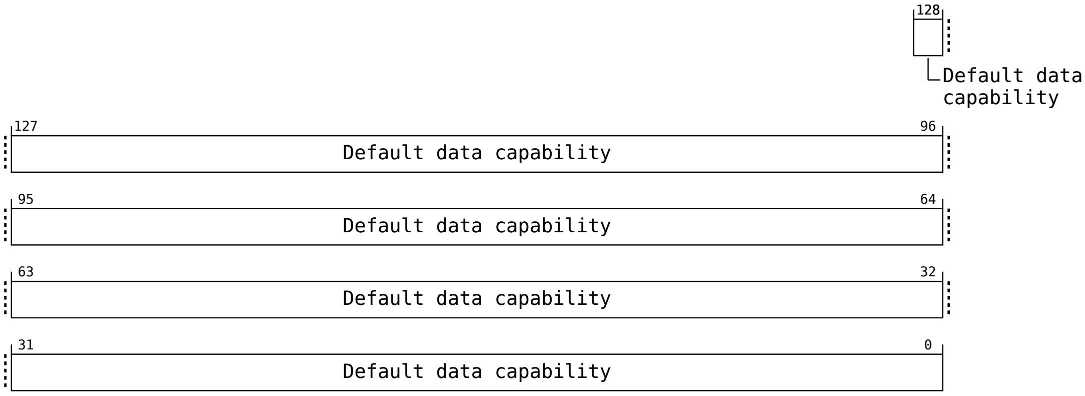
Configuration

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

This register is present only when Morello is implemented. Otherwise, direct accesses to DDC_EL2 are UNDEFINED.

Field descriptions

The DDC_EL2 bit assignments are:



Bits [128:0]

Default data capability.

This field resets to `0x1FFFFC000000100050000000000000000`.

Accessing the DDC_EL2

Read using name DDC_EL2

The assembler syntax is:

```
MRS <Ct>, DDC_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
```

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return DDC_EL2;
```

Write using name DDC_EL2

The assembler syntax is:

```
MSR DDC_EL2, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        DDC_EL2 = C[t];
```

Read using name DDC

The assembler syntax is:

```
MRS <Ct>, DDC
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') &&
    ↪CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
```

```
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    return RDDC_EL0;
elsif PSTATE.SP == '0' then
    return DDC_EL0;
elsif PSTATE.EL == EL0 then
    return DDC_EL0;
elsif PSTATE.EL == EL1 then
    return DDC_EL1;
elsif PSTATE.EL == EL2 then
    return DDC_EL2;
elsif PSTATE.EL == EL3 then
    return DDC_EL3;
```

Write using name DDC

The assembler syntax is:

`MSR DDC, <Ct>`

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    RDDC_EL0 = C[t];
elsif PSTATE.SP == '0' then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL0 then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    DDC_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    DDC_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    DDC_EL3 = C[t];
```

3.2.19 DDC_EL3, Default Data Capability (EL3)

The DDC_EL3 characteristics are:

Purpose

Holds the default data capability associated with EL3 when the PE is in Executive.

Attributes

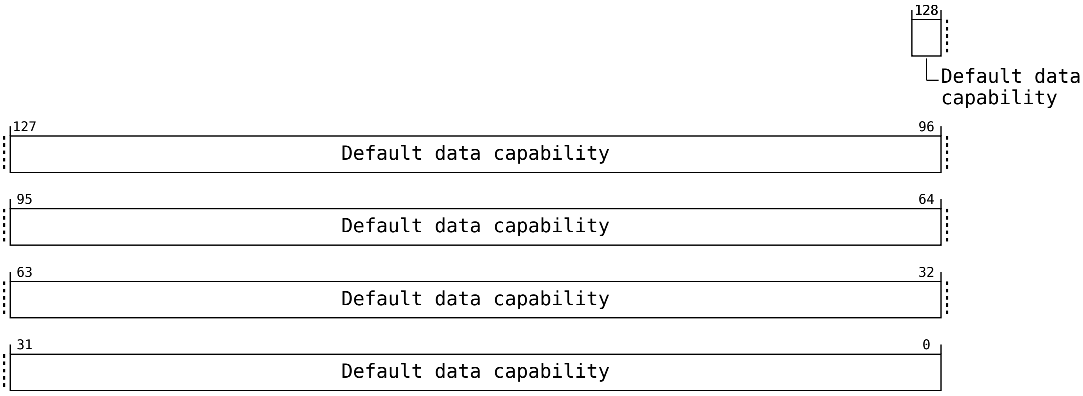
DDC_EL3 is a 129-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to DDC_EL3 are UNDEFINED.

Field descriptions

The DDC_EL3 bit assignments are:



Bits [128:0]

Default data capability.

This field resets to `0x1FFFFC000000100050000000000000000`.

Accessing the DDC_EL3

Read using name DDC

The assembler syntax is:

```
MRS <Ct>, DDC
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') &&
    ↪CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    return RDDC_EL0;
elsif PSTATE.SP == '0' then
    return DDC_EL0;
elsif PSTATE.EL == EL0 then
    return DDC_EL0;
elsif PSTATE.EL == EL1 then
    return DDC_EL1;
elsif PSTATE.EL == EL2 then
    return DDC_EL2;
elsif PSTATE.EL == EL3 then
    return DDC_EL3;
```

Write using name DDC

The assembler syntax is:

```
MSR DDC, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    RDDC_EL0 = C[t];
elsif PSTATE.SP == '0' then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL0 then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    DDC_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    DDC_EL2 = C[t];
elsif PSTATE.EL == EL3 then
```

```
    DDC_EL3 = C[t];
```

3.2.20 DSPSR_EL0, Debug Saved Program Status Register

The DSPSR_EL0 characteristics are:

Purpose

Holds the saved process state for Debug state. On entering Debug state, PSTATE information is written to this register. On exiting Debug state, values are copied from this register to PSTATE.

Attributes

DSPSR_EL0 is a 64-bit register.

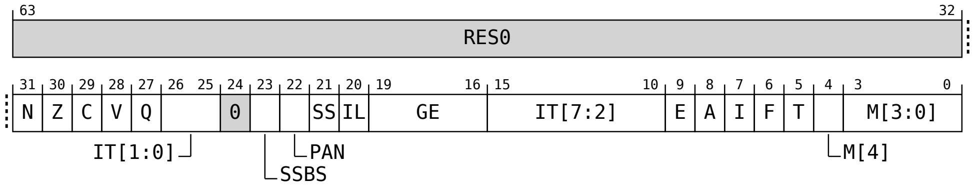
Configuration

AArch64 System register DSPSR_EL0[31:0] is architecturally mapped to AArch32 System register DSPSR[31:0].

Field descriptions

The DSPSR_EL0 bit assignments are:

When exiting Debug state to AArch32 state:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Copied to PSTATE.N on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Copied to PSTATE.Z on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Copied to PSTATE.C on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Copied to PSTATE.V on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Copied to PSTATE.Q on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IT[1:0], bits [26:25]

If-Then. Copied to PSTATE.IT[1:0] on exiting Debug state.

On exiting Debug state DSPSR_EL0.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

Bit [24]

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Copied to PSTATE.SSBS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Copied to PSTATE.PAN on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Copied to PSTATE.SS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Copied to PSTATE.IL on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Copied to PSTATE.GE on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Copied to PSTATE.IT[7:2] on exiting Debug state.

DSPSR_EL0.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Copied to PSTATE.E on exiting Debug state.

If the implementation does not support big-endian operation, DSPSR_EL0.E is RES0. If the implementation does not support little-endian operation, DSPSR_EL0.E is RES1. On exiting Debug state, if the implementation does not support big-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, DSPSR_EL0.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Copied to PSTATE.A on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Copied to PSTATE.I on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Copied to PSTATE.F on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Copied to PSTATE.T on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Copied to PSTATE.nRW on exiting Debug state.

Value	Meaning
`0b1`	AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch32 Mode. Copied to PSTATE.M[3:0] on exiting Debug state.

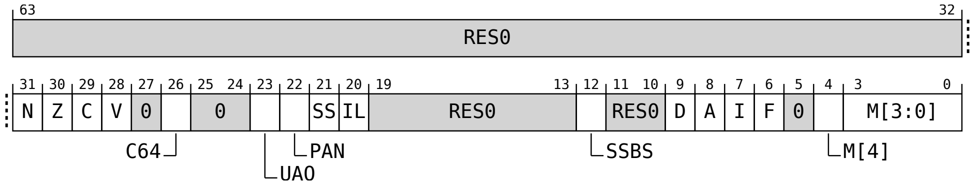
Value	Meaning
`0b0000`	User.
`0b0001`	FIQ.
`0b0010`	IRQ.
`0b0011`	Supervisor.
`0b0110`	Monitor.
`0b0111`	Abort.
`0b1010`	Hyp.

Value	Meaning
`0b1011`	Undefined.
`0b1111`	System.

Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in x'Illegal return events from AArch64 state'.

This field resets to an architecturally UNKNOWN value.

When entering Debug state from AArch64 state and exiting Debug state to AArch64 state:



Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on entering Debug state, and copied to PSTATE.N on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on entering Debug state, and copied to PSTATE.Z on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on entering Debug state, and copied to PSTATE.C on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on entering Debug state, and copied to PSTATE.V on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Bit [27]

Reserved, RES0.

C64, bit [26]

When Morello is implemented:

Current instruction set state. Set to the value of PSTATE.C64 on entering Debug state, and copied to PSTATE.C64 on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [25:24]

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on entering Debug state, and copied to PSTATE.UAO on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on entering Debug state, and copied to PSTATE.PAN on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on entering Debug state, and conditionally copied to PSTATE.SS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on entering Debug state, and copied to PSTATE.IL on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on entering Debug state, and copied to PSTATE.SSBS on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [11:10]

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on entering Debug state, and copied to PSTATE.D on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on entering Debug state, and copied to PSTATE.A on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on entering Debug state, and copied to PSTATE.I on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on entering Debug state, and copied to PSTATE.F on exiting Debug state.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0b0, the value of PSTATE.nRW, on entering Debug state from AArch64 state, and copied to PSTATE.nRW on exiting Debug state.

Value	Meaning
`0b0`	AArch64 execution state.

If AArch32 is not supported at any Exception level, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

Value	Meaning
`0b0000`	EL0t.
`0b0100`	EL1t.

Value	Meaning
`0b0101`	EL1h.
`0b1000`	EL2t.
`0b1001`	EL2h.
`0b1100`	EL3t.
`0b1101`	EL3h.

Other values are reserved. If DSPSR_EL0.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, exiting Debug state is an illegal return event, as described in x'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on entering Debug state and copied to PSTATE.EL on exiting Debug state.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on entering Debug state and copied to PSTATE.SP on exiting Debug state

This field resets to an architecturally UNKNOWN value.

Accessing the DSPSR_EL0

Read using name DSPSR_EL0

The assembler syntax is:

```
MRS <Xt>, DSPSR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

Accessibility:

```
if !Halted() then
    UNDEFINED;
else
    return DSPSR_EL0;
```

Write using name DSPSR_EL0

The assembler syntax is:

```
MSR DSPSR_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0101	0b000

Accessibility:

```
if !Halted() then
    UNDEFINED;
else
    DSPSR_EL0 = X[t];
```

3.2.21 EDSCR2, External Debug Status and Control Register 2

The EDSCR2 characteristics are:

Purpose

Extended control register for the debug implementation

Attributes

EDSCR2 is a 32-bit register.

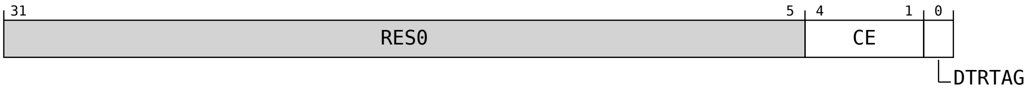
Configuration

External register EDSCR2[0] is architecturally mapped to AArch64 System register CDBGDTR_EL0[128].

This register is present only when Morello is implemented. Otherwise, direct accesses to EDSCR2 are RES0.

Field descriptions

The EDSCR2 bit assignments are:



Bits [31:5]

Reserved, RES0.

CE, bits [4:1]

Access to Morello Feature status. In Debug state, each bit gives the current access to the Morello architecture extension at each Exception level as controlled by CPTR_ELx and CPACR_EL1:

Value	Meaning
`0b1111`	All Exception levels have access to the Morello architecture extension or the PE is in Non-debug state.
`0b1110`	The PE is in Debug state. EL0 does not have access to the Morello architecture extension. All other Exception levels have access to the Morello architecture extension.
`0b1100`	The PE is in Debug state. EL0 and EL1 do not have access to the Morello architecture extension. All other Exception levels have access to the Morello architecture extension.
`0b1000`	The PE is in Debug state. EL3 has access to the Morello architecture extension. All other Exception levels do not have access to the Morello architecture extension.
`0b0000`	The PE is in Debug state. No Exception level has access to the Morello architecture extension.

In Non-debug state, this field is RAO.

Access to this field is **RO**.

DTRTAG, bit [0]

Capability data transfer register tag.

On a cold reset, this field resets to an UNKNOWN value.

Accessing the EDSCR2

Access to EDSCR2 is only possible externally

EDSCR2 can be accessed through the external debug interface:

Component	Offset	Instance
Debug	0x048	EDSCR2

This interface is accessible as follows:

- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and SoftwareLockStatus() access to this register is **RO**.
- When IsCorePowered(), !DoubleLockStatus(), !OSLockStatus() and !SoftwareLockStatus() access to this register is **RW**.
- Otherwise access to this register returns an ERROR.

3.2.22 ELR_EL1, Exception Link Register (EL1)

The ELR_EL1 characteristics are:

Purpose

When taking an exception to EL1, holds the address to return to.

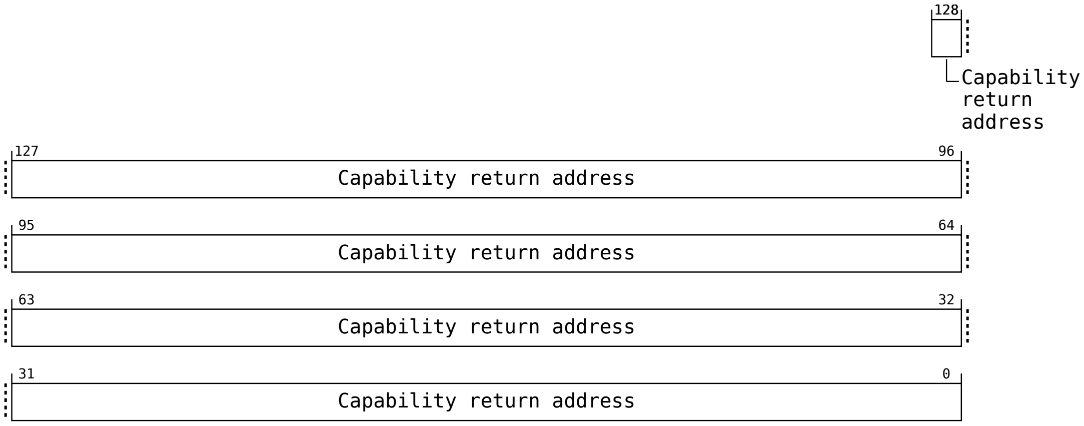
Attributes

ELR_EL1 is a 129-bit register.

Field descriptions

The ELR_EL1 bit assignments are:

When Morello is implemented and Capability access at EL1 is not trapped:


128
Capability return address
127
96
Capability return address
95
64
Capability return address
63
32
Capability return address
31
0
Capability return address

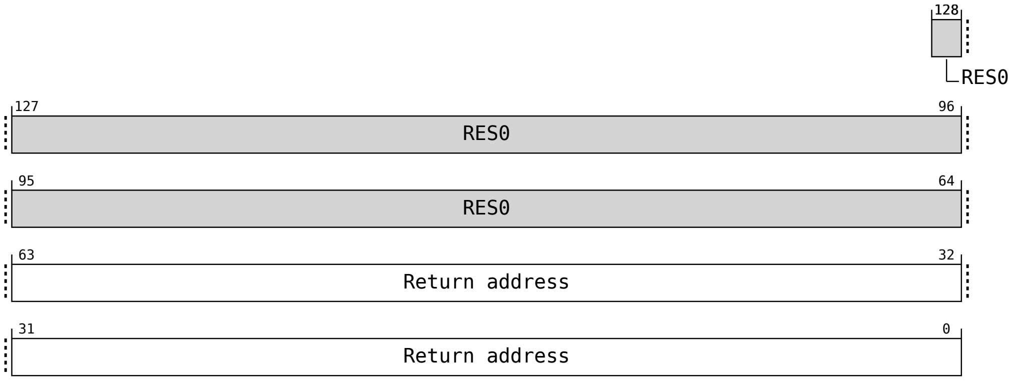
Bits [128:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

When Morello is implemented and Capability access at EL1 is trapped:



Bits [128:64]

Reserved, RES0.

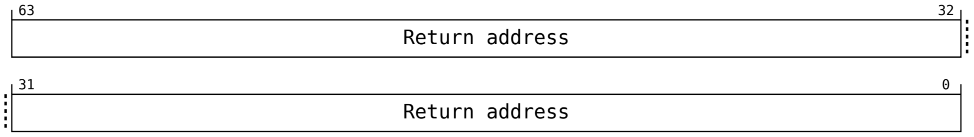
Bits [63:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Return address.

An exception return from EL1 using AArch64 makes ELR_EL1 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the ELR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic ELR_EL1 or ELR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name ELR_EL1

The assembler syntax is:

```
MRS <Xt>, ELR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return ELR_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ELR_EL2<63:0>;
    else
        return ELR_EL1<63:0>;
elsif PSTATE.EL == EL3 then
    return ELR_EL1<63:0>;
```

Write using name ELR_EL1

The assembler syntax is:

```
MSR ELR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    ELR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL2 = ZeroExtend(X[t]);
    else
        ELR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    ELR_EL1 = ZeroExtend(X[t]);
```

Read using name ELR_EL12

The assembler syntax is:

```
MRS <Xt>, ELR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ELR_EL1<63:0>;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return ELR_EL1<63:0>;
    else
        UNDEFINED;
```

Write using name ELR_EL12

The assembler syntax is:

```
MSR ELR_EL12, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL1 = ZeroExtend(X[t]);
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        ELR_EL1 = ZeroExtend(X[t]);
    else
        UNDEFINED;
```

Read using name CELR_EL1

The assembler syntax is:

```
MRS <Ct>, CELR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
```

```
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return ELR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        return ELR_EL2;
    else
        return ELR_EL1;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return ELR_EL1;
```

Write using name CELR_EL1

The assembler syntax is:

```
MSR CELR_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        ELR_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        ELR_EL2 = C[t];
    else
        ELR_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        ELR_EL1 = C[t];
```

Read using name CELR_EL12

The assembler syntax is:

```
MRS <Ct>, CELR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            return ELR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            return ELR_EL1;
    else
        UNDEFINED;
```

Write using name CELR_EL12

The assembler syntax is:

```
MSR CELR_EL12, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            ELR_EL1 = C[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            ELR_EL1 = C[t];
    else
        UNDEFINED;
```

3.2.23 ELR_EL2, Exception Link Register (EL2)

The ELR_EL2 characteristics are:

Purpose

When taking an exception to EL2, holds the address to return to.

Attributes

ELR_EL2 is a 129-bit register.

Configuration

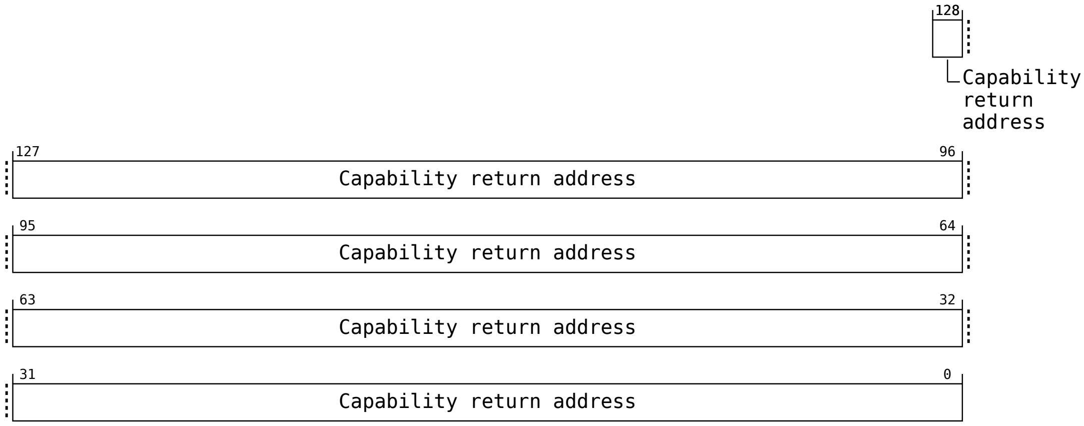
This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register ELR_EL2[31:0] is architecturally mapped to AArch32 System register ELR_hyp[31:0].

Field descriptions

The ELR_EL2 bit assignments are:

When Morello is implemented and Capability access at EL2 is not trapped:


128
Capability return address
127
96
Capability return address
95
64
Capability return address
63
32
Capability return address
31
0
Capability return address

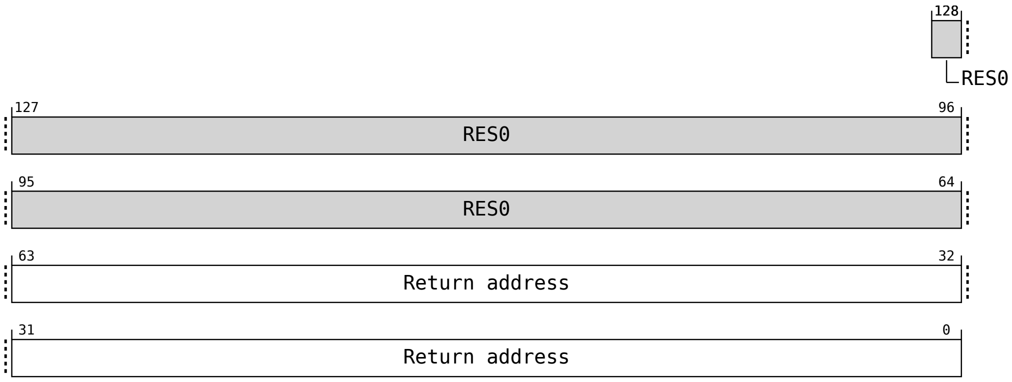
Bits [128:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

When Morello is implemented and Capability access at EL2 is trapped:



Bits [128:64]

Reserved, RES0.

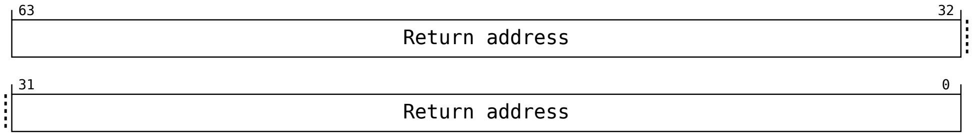
Bits [63:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Return address.

An exception return from EL2 using AArch64 makes ELR_EL2 become UNKNOWN.

When EL2 is in AArch32 Execution state and an exception is taken from EL0, EL1, or EL2 to EL3 and AArch64 execution, the upper 32-bits of ELR_EL2 are either set to 0 or hold the same value that they did before AArch32 execution. Which option is adopted is determined by an implementation, and might vary dynamically within an implementation. Correspondingly software must regard the value as being an UNKNOWN choice between the two values.

This field resets to an architecturally UNKNOWN value.

Accessing the ELR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic ELR_EL2 or ELR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name ELR_EL2

The assembler syntax is:

```
MRS <Xt>, ELR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return ELR_EL2<63:0>;
elsif PSTATE.EL == EL3 then
    return ELR_EL2<63:0>;
```

Write using name ELR_EL2

The assembler syntax is:

```
MSR ELR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    ELR_EL2 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    ELR_EL2 = ZeroExtend(X[t]);
```

Read using name ELR_EL1

The assembler syntax is:

```
MRS <Xt>, ELR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

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

```
    return ELR_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return ELR_EL2<63:0>;
    else
        return ELR_EL1<63:0>;
elsif PSTATE.EL == EL3 then
    return ELR_EL1<63:0>;
```

Write using name ELR_EL1

The assembler syntax is:

```
MSR ELR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    ELR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        ELR_EL2 = ZeroExtend(X[t]);
    else
        ELR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    ELR_EL1 = ZeroExtend(X[t]);
```

Read using name CELR_EL2

The assembler syntax is:

```
MRS <Ct>, CELR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return ELR_EL2;
```

```
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return ELR_EL2;
```

Write using name CELR_EL2

The assembler syntax is:

```
MSR CELR_EL2, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        ELR_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        ELR_EL2 = C[t];
```

Read using name CELR_EL1

The assembler syntax is:

```
MRS <Ct>, CELR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return ELR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        return ELR_EL2;
    else
        return ELR_EL1;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return ELR_EL1;
```

Write using name CELR_EL1

The assembler syntax is:

```
MSR CELR_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        ELR_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        ELR_EL2 = C[t];
    else
        ELR_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        ELR_EL1 = C[t];
```

3.2.24 ELR_EL3, Exception Link Register (EL3)

The ELR_EL3 characteristics are:

Purpose

When taking an exception to EL3, holds the address to return to.

Attributes

ELR_EL3 is a 129-bit register.

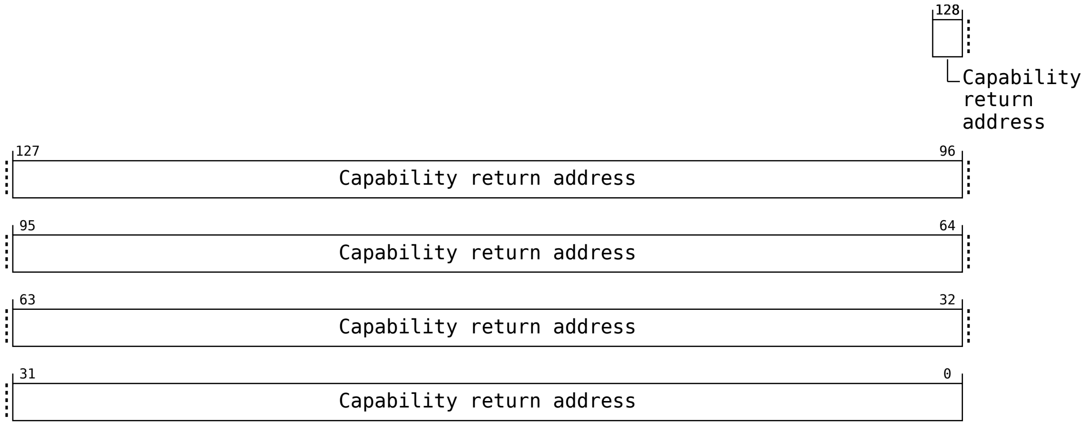
Configuration

This register is present only when HaveEL(EL3). Otherwise, direct accesses to ELR_EL3 are UNDEFINED.

Field descriptions

The ELR_EL3 bit assignments are:

When Morello is implemented and Capability access at EL3 is not trapped:


128
Capability return address
127
96
Capability return address
95
64
Capability return address
63
32
Capability return address
31
0
Capability return address

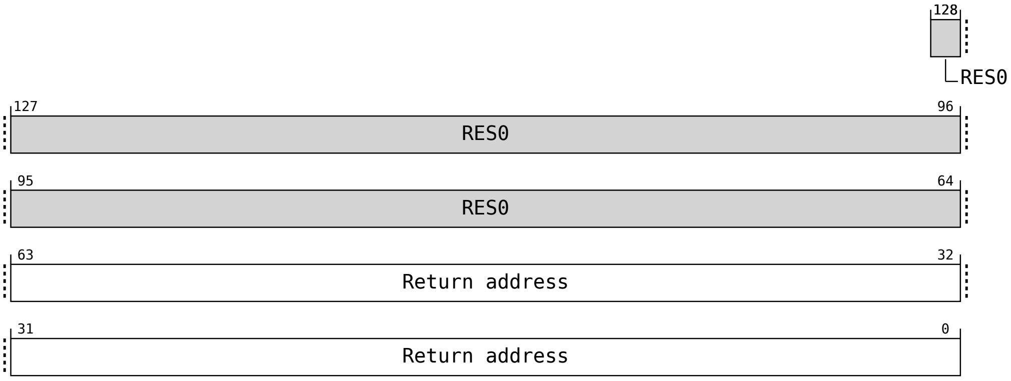
Bits [128:0]

Return address.

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

When Morello is implemented and Capability access at EL3 is trapped:



Bits [128:64]

Reserved, RES0.

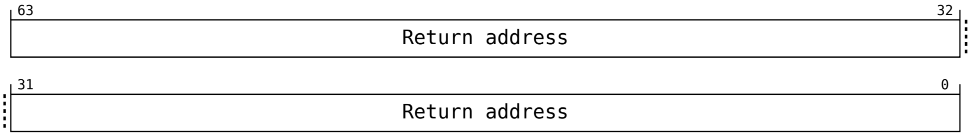
Bits [63:0]

Return address.

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Return address.

An exception return from EL3 using AArch64 makes ELR_EL3 become UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Accessing the ELR_EL3

Read using name ELR_EL3

The assembler syntax is:

```
MRS <Xt>, ELR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return ELR_EL3<63:0>;
```

Write using name ELR_EL3

The assembler syntax is:

```
MSR ELR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    ELR_EL3 = ZeroExtend(X[t]);
```

Read using name CELR_EL3

The assembler syntax is:

```
MRS <Ct>, CELR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return ELR_EL3;
```

Write using name CELR_EL3

The assembler syntax is:

```
MSR CELR_EL3, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        ELR_EL3 = C[t];
```

3.2.25 ESR_EL1, Exception Syndrome Register (EL1)

The ESR_EL1 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL1.

Attributes

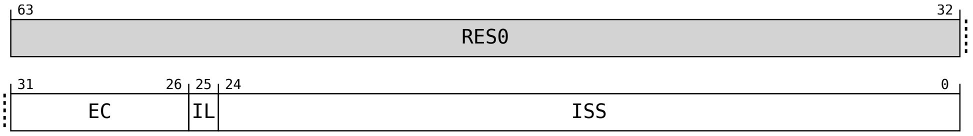
ESR_EL1 is a 64-bit register.

Configuration

AArch64 System register ESR_EL1[31:0] is architecturally mapped to AArch32 System register DFSR[31:0].

Field descriptions

The ESR_EL1 bit assignments are:



ESR_EL1 is made UNKNOWN as a result of an exception return from EL1.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL1, the value of ESR_EL1 is UNKNOWN. The value written to ESR_EL1 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

Value	Meaning	Link	Applies
0b000000	Unknown reason.	ISS - exceptions with an unknown reason	
0b000001	Trapped WFI or WFE instruction execution. Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.	ISS - an exception from a WFI or WFE instruction	
0b000011	Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC 0b000000.	ISS - an exception from an MCR or MRC access	

Value	Meaning	Link	Applies
`0b000100`	Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC 0b000000.	ISS - an exception from an MCRR or MRRC access	
`0b000101`	Trapped MCR or MRC access with (coproc==0b1110).	ISS - an exception from an MCR or MRC access	
`0b000110`	Trapped LDC or STC access. The only architected uses of these instruction are: • An STC to write data to memory from DBGDTRRXint. • An LDC to read data from memory to DBGDTRTXint.	ISS - an exception from an LDC or STC instruction	
`0b000111`	Access to SVE, Advanced SIMD, or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000 as described in ’EC encodings when routing exceptions to EL2’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section D1.10.4.	ISS - an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP	
`0b001100`	Trapped MRRC access with (coproc==0b1110).	ISS - an exception from an MCRR or MRRC access	
`0b001110`	Illegal Execution state.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	
`0b010001`	SVC instruction execution in AArch32 state. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.	ISS - an exception from HVC or SVC instruction execution	
`0b010101`	SVC instruction execution in AArch64 state.	ISS - an exception from HVC or SVC instruction execution	

Value	Meaning	Link	Applies
0b011000	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC 0b000000, 0b000001, 0b000111 or 0b101010. If xARMv8.0-CSV2 is implemented, also Cache Speculation Variant exceptions. If xARMv8.2-EVT is implemented, also traps for EL1 and EL0 Cache controls. This includes all instructions that cause exceptions that are part of the encoding space defined in ’System instruction class encoding overview’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section C5.2.2, except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS - an exception from MSR, MRS, or System instruction execution in AArch64 state	
0b011001	Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC 0b000000. This EC is defined only if xSVEis implemented.	ISS - an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ	
0b100000	Instruction Abort from a lower Exception level, that might be using AArch32 or AArch64. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from an Instruction Abort	
0b100001	Instruction Abort taken without a change in Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from an Instruction Abort	
0b100010	PC alignment fault exception.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	

Value	Meaning	Link	Applies
0b100100	Data Abort from a lower Exception level, that might be using AArch32 or AArch64. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from a Data Abort	
0b100101	Data Abort taken without a change in Exception level. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from a Data Abort	
0b100110	SP alignment fault exception.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	
0b101000	Trapped floating-point exception taken from AArch32 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS - an exception from a trapped floating-point exception	
0b101001	Access to the Morello architecture trapped as a result of CPACR_EL1.CEN, CPTR_EL2.CEN, CPTR_EL2.TC, or CPTR_EL3.EC.	ISS - an exception from an access to the Morello architecture	When Morello is implemented
0b101010	Trapped capability MSR or MRS instruction execution. This EC value is valid if Morello architecture is implemented, otherwise it is reserved. Used for trapped accesses to capability System registers via MSR or MRS instructions.	ISS - an exception from capability MSR or MRS instruction execution	When Morello is implemented
0b101100	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS - an exception from a trapped floating-point exception	

Value	Meaning	Link	Applies
0b101111	SError interrupt.	ISS - an SError interrupt	
0b110000	Breakpoint exception from a lower Exception level, that might be using AArch32 or AArch64.	ISS - an exception from a Breakpoint or Vector Catch debug exception	
0b110001	Breakpoint exception taken without a change in Exception level.	ISS - an exception from a Breakpoint or Vector Catch debug exception	
0b110010	Software Step exception from a lower Exception level, that might be using AArch32 or AArch64.	ISS - an exception from a Software Step exception	
0b110011	Software Step exception taken without a change in Exception level.	ISS - an exception from a Software Step exception	
0b110100	Watchpoint exception from a lower Exception level, that might be using AArch32 or AArch64.	ISS - an exception from a Watchpoint exception	
0b110101	Watchpoint exception taken without a change in Exception level.	ISS - an exception from a Watchpoint exception	
0b111000	BKPT instruction execution in AArch32 state.	ISS - an exception from execution of a Breakpoint instruction	
0b111100	BRK instruction execution in AArch64 state. This is reported in ESR_EL3 only if a BRK instruction is executed.	ISS - an exception from execution of a Breakpoint instruction	

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in ’Reserved values in System and memory-mapped registers and translation table entries’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section K1.1.11.

This field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

Value	Meaning
0b0	16-bit instruction trapped.

Value	Meaning
0b1	32-bit instruction trapped. This value is also used when the exception is one of the following: • An SError interrupt. • An Instruction Abort exception. • A PC alignment fault exception. • An SP alignment fault exception. • A Data Abort exception for which the value of the ISV bit is 0. • An Illegal Execution state exception. • Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning: – 0b0: 16-bit T32 BKPT instruction. – 0b1: 32-bit A32 BKPT instruction or A64 BRK instruction. • An exception reported using EC value 0b000000.

This field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

For an exception taken from AArch32 state, see x'Mapping of the general-purpose registers between the Execution states'.

If the AArch32 register descriptor is 0b1111, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value 0b11111.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value 0b11111.

When the EC field is 0b000000, indicating an exception with an unknown reason, the ISS field is not valid, RES0.

exceptions with an unknown reason

Bits [24:0]

Reserved, RES0.

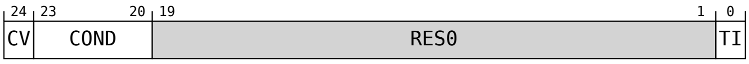
When an exception is reported using this EC code the IL field is set to 1.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:

  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon. See x'Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32'.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of 0b000111.
- When SVE is not implemented, attempted execution of:
  - An SVE instruction.
  - An MSR or MRS instruction to access ZCR_EL1, ZCR_EL2, or ZCR_EL3.

an exception from a WFI or WFE instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

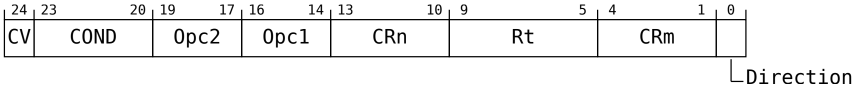
Value	Meaning
`0b0`	WFI trapped.
`0b1`	WFE trapped.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

an exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
0b0	Write to System register space. MCR instruction.
0b1	Read from System register space. MRC or VMRS instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000011:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32

state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- For information on other traps using EC value 0b000011, see x'Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32'.
- If xARMv8.6-FGT is implemented, MCR or MRC access to some registers at EL0, trapped to EL2. [endif]

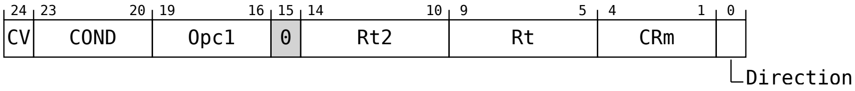
The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000101:

- CPACR_EL1.TTA for accesses to trace registers, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.
- HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDA, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001000:

- HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

an exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
`0b0`	Write to System register space. MCRR instruction.
`0b1`	Read from System register space. MRRC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000100:

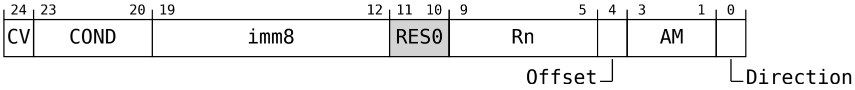
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.

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

- CPACR_EL1.TTA for accesses to trace registers using MCR or MRC instructions, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.

- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDOSA, for traps to powerdown debug registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.

an exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is

set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Value	Meaning
0b0	Subtract offset.
0b1	Add offset.

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

This field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

Value	Meaning
0b000	Immediate unindexed.
0b001	Immediate post-indexed.
0b010	Immediate offset.
0b011	Immediate pre-indexed.
0b100	For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
0b110	For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in x'Reserved values in System and memory-mapped registers and translation table entries'.

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

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

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

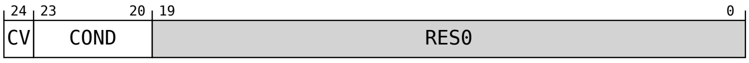
Value	Meaning
`0b0`	Write to memory. STC instruction.
`0b1`	Read from memory. LDC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000110:

- MDSCR_EL1.TDCC, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.
- MDCR_EL2.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- MDCR_EL3.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.

an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP



The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include either SVE or support for floating-point and Advanced SIMD, the exception is reported using the EC value 0b000000.

CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

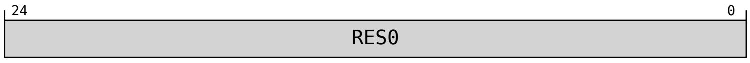
Bits [19:0]

Reserved, RES0.

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

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ



Bits [24:0]

When SVE is implemented:

Reserved, RES0.

Otherwise:

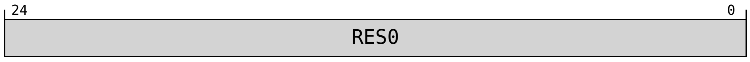
RES0

The accesses covered by this trap include:

- Execution of SVE instructions.
- Accesses to the SVE system registers, ZCR_ELx and ID_AA64ZFR0_EL1.

For an implementation that does not include SVE, the exception is reported using the EC value 0b000000.

an exception from an Illegal Execution state, or a PC or SP alignment fault



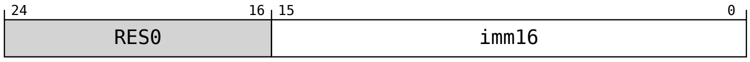
Bits [24:0]

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about these exceptions see x‘The Illegal Execution state exception’ and x‘PC alignment checking’.

x‘SP alignment checking’ describes the configuration settings for generating SP alignment fault exceptions.

an exception from HVC or SVC instruction execution



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

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

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

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

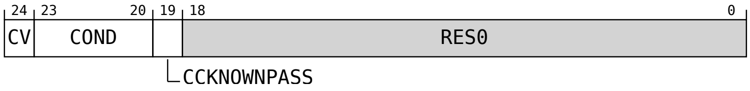
This field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see x‘SVC’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile and x‘HVC’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

For A64 instructions, see x‘SVC’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile and x‘HVC’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from SMC instruction execution in AArch32 state



For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

Value	Meaning
`0b0`	The instruction was unconditional, or was conditional and passed its condition code check.
`0b1`	The instruction was conditional, and might have failed its condition code check.

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

This field resets to an architecturally UNKNOWN value.

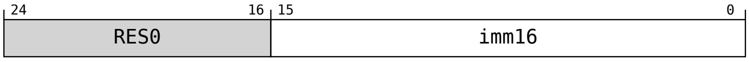
Bits [18:0]

Reserved, RES0.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

x'System calls' describes the case where these exceptions are trapped to EL3.

an exception from SMC instruction execution in AArch64 state



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

This field resets to an architecturally UNKNOWN value.

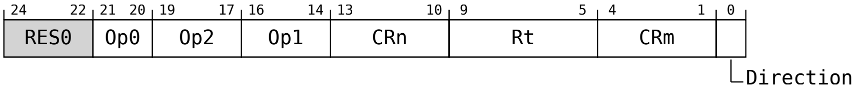
The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

x'System calls' describes the case where these exceptions are trapped to EL3.

an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

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

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
`0b0`	Write access, including MSR instructions.
`0b1`	Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

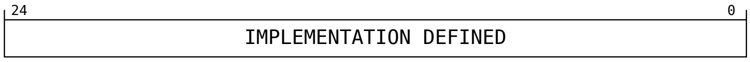
For exceptions caused by System instructions, see x'System instructions' subsection of 'Branches, exception generating and System instructions' for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.

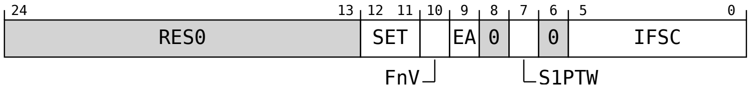
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the filter trace control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If xARMv8.2-EVT is implemented, HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4} and HCR2.{TTLBIS, TICAB, TOCU, TID4} control traps for EL1 and EL0 Cache controls that use this EC value.

an IMPLEMENTATION DEFINED exception to EL3



IMPLEMENTATION DEFINED, bits [24:0] IMPLEMENTATION DEFINED

an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and IFSC is 0b010000, describes the state of the PE after taking the Instruction Abort exception. The possible values of this field are:

Value	Meaning
0b00	Recoverable error (UER).
0b10	Uncontainable error (UC).
0b11	Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the IFSC field is not 0b010000.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

Value	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is 0b010000. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Value	Meaning
`0b0`	Fault not on a stage 2 translation for a stage 1 translation table walk.
`0b1`	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

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

Value	Meaning
`0b000000`	Address size fault, level 0 of translation or translation table base register
`0b000001`	Address size fault, level 1
`0b000010`	Address size fault, level 2
`0b000011`	Address size fault, level 3
`0b000100`	Translation fault, level 0
`0b000101`	Translation fault, level 1
`0b000110`	Translation fault, level 2
`0b000111`	Translation fault, level 3
`0b001001`	Access flag fault, level 1
`0b001010`	Access flag fault, level 2
`0b001011`	Access flag fault, level 3
`0b001101`	Permission fault, level 1
`0b001110`	Permission fault, level 2
`0b001111`	Permission fault, level 3
`0b010000`	Synchronous External abort, not on translation table walk
`0b010100`	Synchronous External abort, on translation table walk, level 0

Value	Meaning
`0b010101`	Synchronous External abort, on translation table walk, level 1
`0b010110`	Synchronous External abort, on translation table walk, level 2
`0b010111`	Synchronous External abort, on translation table walk, level 3
`0b011000`	Synchronous parity or ECC error on memory access, not on translation table walk
`0b011100`	Synchronous parity or ECC error on memory access on translation table walk, level 0
`0b011101`	Synchronous parity or ECC error on memory access on translation table walk, level 1
`0b011110`	Synchronous parity or ECC error on memory access on translation table walk, level 2
`0b011111`	Synchronous parity or ECC error on memory access on translation table walk, level 3
`0b101000`	Capability tag fault.
`0b101001`	Capability sealed fault.
`0b101010`	Capability bound fault.
`0b101011`	Capability permission fault.
`0b110000`	TLB conflict abort
`0b110001`	Unsupported atomic hardware update fault, if the implementation includes xARMv8.1-TTHM. Otherwise reserved.

All other values are reserved.

When the RAS Extension is implemented, 0b011000, 0b011100, 0b011101, 0b011110, and 0b011111, are reserved.

Armv8.2 requires the implementation of the RAS Extension.

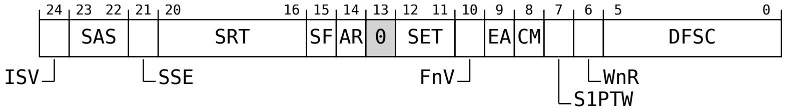
For more information about the lookup level associated with a fault, see x‘The level associated with MMU faults’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

an exception from a Data Abort



ISV, bit [24]

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

Value	Meaning
`0b0`	No valid instruction syndrome. ISS[23:14] are RES0.
`0b1`	ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with 0b11111, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive, excluding those with writeback and excluding accesses of a capability.
- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When the RAS Extension is not implemented, the value of ISV on a synchronous External abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

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

Value	Meaning
`0b00`	Byte
`0b01`	Halfword
`0b10`	Word
`0b11`	Doubleword

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

Value	Meaning
`0b0`	Sign-extension not required.

Value	Meaning
0b1	Data item must be sign-extended.

For all other operations this bit is 0.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SRT, bits [20:16]

Syndrome Register transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction. If the exception was taken from an Exception level that is using AArch32 then this is the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SF, bit [15]

Width of the register accessed by the instruction is Sixty-Four. When ISV is 1, the possible values of this bit are:

Value	Meaning
0b0	Instruction loads/stores a 32-bit wide register.
0b1	Instruction loads/stores a 64-bit wide register.

This field specifies the register width identified by the instruction, not the Execution state.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

Value	Meaning
0b0	Instruction did not have acquire/release semantics.
0b1	Instruction did have acquire/release semantics.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and DFSC is 0b010000, describes the state of the PE after taking the Data Abort exception. The possible values of this field are:

Value	Meaning
`0b00`	Recoverable error (UER).
`0b10`	Uncontainable error (UC).
`0b11`	Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010000.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

Value	Meaning
`0b0`	FAR is valid.
`0b1`	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

Value	Meaning
`0b0`	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.

Value	Meaning
`0b1`	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Value	Meaning
`0b0`	Fault not on a stage 2 translation for a stage 1 translation table walk.
`0b1`	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

Value	Meaning
`0b0`	Abort caused by an instruction reading from a memory location.
`0b1`	Abort caused by an instruction writing to a memory location.

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

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

For Page table LC or SC permission violation faults from an atomic instruction that both reads and writes a valid capability from a memory location, this bit is set to 1 if a write of a valid capability from the memory location would have generated the fault which is being reported, otherwise it is set to 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

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

Value	Meaning
`0b000000`	Address size fault, level 0 of translation or translation table base register.
`0b000001`	Address size fault, level 1.
`0b000010`	Address size fault, level 2.
`0b000011`	Address size fault, level 3.
`0b000100`	Translation fault, level 0.
`0b000101`	Translation fault, level 1.
`0b000110`	Translation fault, level 2.
`0b000111`	Translation fault, level 3.
`0b001001`	Access flag fault, level 1.
`0b001010`	Access flag fault, level 2.
`0b001011`	Access flag fault, level 3.
`0b001101`	Permission fault, level 1.
`0b001110`	Permission fault, level 2.
`0b001111`	Permission fault, level 3.
`0b010000`	Synchronous External abort, not on translation table walk.
`0b010001`	Synchronous Tag Check fail
`0b010100`	Synchronous External abort, on translation table walk, level 0.
`0b010101`	Synchronous External abort, on translation table walk, level 1.
`0b010110`	Synchronous External abort, on translation table walk, level 2.
`0b010111`	Synchronous External abort, on translation table walk, level 3.
`0b011000`	Synchronous parity or ECC error on memory access, not on translation table walk.
`0b011100`	Synchronous parity or ECC error on memory access on translation table walk, level 0.
`0b011101`	Synchronous parity or ECC error on memory access on translation table walk, level 1.
`0b011110`	Synchronous parity or ECC error on memory access on translation table walk, level 2.
`0b011111`	Synchronous parity or ECC error on memory access on translation table walk, level 3.
`0b100001`	Alignment fault.
`0b101000`	Capability tag fault.
`0b101001`	Capability sealed fault.
`0b101010`	Capability bound fault.
`0b101011`	Capability permission fault.
`0b101100`	Page table LC or SC permission violation fault.

Value	Meaning
0b110000	TLB conflict abort.
0b110001	Unsupported atomic hardware update fault, if the implementation includes xARMv8.1-TTHM. Otherwise reserved.
0b110100	IMPLEMENTATION DEFINED fault (Lockdown).
0b110101	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).
0b110110	Unsupported LDCT or SDCT to Device or Non-cacheable.
0b111101	Section Domain Fault, used only for faults reported in the PAR_EL1.
0b111110	Page Domain Fault, used only for faults reported in the PAR_EL1.

All other values are reserved.

When the RAS Extension is implemented, 0b011000, 0b011100, 0b011101, 0b011110, and 0b011111, are reserved.

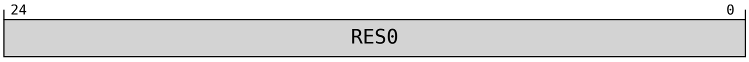
For more information about the lookup level associated with a fault, see x'The level associated with MMU faults' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

an exception from an access to the Morello architecture

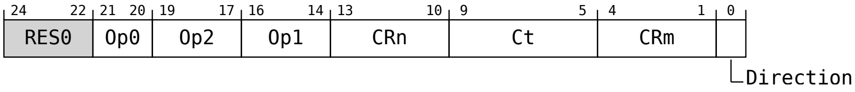


Bits [24:0]

Reserved, RES0.

In an implementation that supports Morello architecture, from an Exception level using AArch64, the CPACR_EL1.CEN, CPTR_EL2.{CEN, DC} and CPTR_EL3.EC bits control whether Morello instructions and accesses to Morello System registers are trapped.

an exception from capability MSR or MRS instruction execution



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Ct, bits [9:5]

The Ct value from the issued instruction, the capability register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

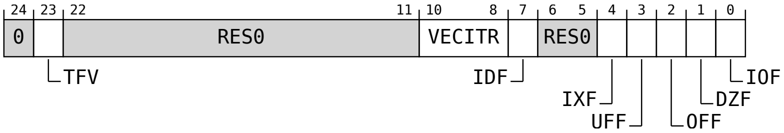
Direction, bit [0]

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

Value	Meaning
`0b0`	Write access, including MSR instructions.
`0b1`	Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions. The possible values of this bit are:

Value	Meaning
`0b0`	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.

Value	Meaning
`0b1`	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information see x'Floating- point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating point exception from a vector instruction.

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from a vector instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

This field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Input denormal floating-point exception has not occurred.
`0b1`	Input denormal floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Inexact floating-point exception has not occurred.
`0b1`	Inexact floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Underflow floating-point exception has not occurred.
`0b1`	Underflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Overflow floating-point exception has not occurred.
`0b1`	Overflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Divide by Zero floating-point exception has not occurred.
`0b1`	Divide by Zero floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

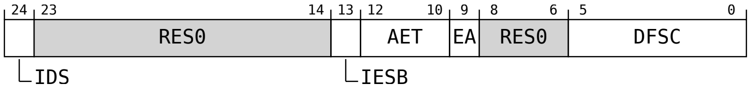
Value	Meaning
`0b0`	Invalid Operation floating-point exception has not occurred.
`0b1`	Invalid Operation floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

an SError interrupt



IDS, bit [24]

IMPLEMENTATION DEFINED syndrome. Possible values of this bit are:

Value	Meaning
`0b0`	Bits[23:0] of the ISS field holds the fields described in this encoding. If the RAS Extension is not implemented, this means that bits[23:0] of the ISS field are RES0.
`0b1`	Bits[23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.

This field was previously called ISV.

This field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

When ARMv8.2-IESB is implemented:

Implicit error synchronization event.

Value	Meaning
`0b0`	The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.
`0b1`	The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

This field is RES0 if the value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension and xARMv8.2-IESB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

AET, bits [12:10]

Asynchronous Error Type.

When the RAS Extension is implemented and DFSC is 0b010001, describes the state of the PE after taking the SError interrupt exception. The possible values of this field are:

Value	Meaning
0b000	Uncontainable error (UC).
0b001	Unrecoverable error (UEU).
0b010	Restartable error (UEO).
0b011	Recoverable error (UER).
0b110	Corrected error (CE).

All other values are reserved.

If multiple errors are taken as a single SError interrupt exception, the overall state of the PE is reported. For example, if both a Recoverable and Unrecoverable error occurred, the state is Unrecoverable.

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. When the RAS Extension is implemented, this bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

Data Fault Status Code. When the RAS Extension is implemented, possible values of this field are:

Value	Meaning
0b000000	Uncategorized.
0b010001	Asynchronous SError interrupt.

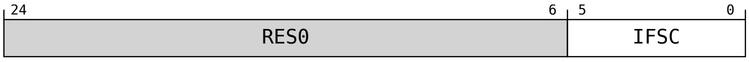
All other values are reserved.

If the RAS Extension is not implemented, this field is RES0.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

an exception from a Breakpoint or Vector Catch debug exception



Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

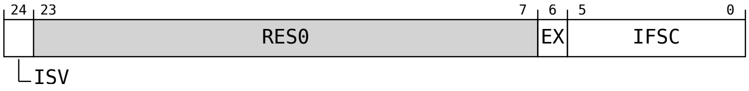
Instruction Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions:

- For exceptions from AArch64, see x'Breakpoint exceptions'.
- For exceptions from AArch32, see x'Breakpoint exceptions' and x'Vector Catch exceptions'.

an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

Value	Meaning
`0b0`	EX bit is RES0.
`0b1`	EX bit is valid.

See the EX bit description for more information.

This field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

Value	Meaning
`0b0`	An instruction other than a Load- Exclusive instruction was stepped.
`0b1`	A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

This field resets to an architecturally UNKNOWN value.

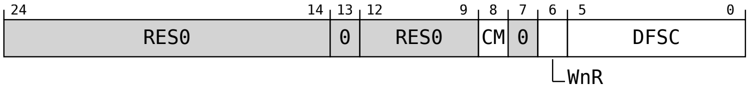
IFSC, bits [5:0]

Instruction Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x'Software Step exceptions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile,.

an exception from a Watchpoint exception



Bits [24:14]

Reserved, RES0.

Bit [13]

Reserved, RES0.

Bits [12:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance or address translation instruction:

Value	Meaning
`0b0`	The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
`0b1`	The Watchpoint exception was generated by either the execution of a cache maintenance instruction or by a synchronous Watchpoint exception on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

Value	Meaning
`0b0`	Watchpoint exception caused by an instruction reading from a memory location.

Value	Meaning
`0b1`	Watchpoint exception caused by an instruction writing to a memory location.

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

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

This field resets to an architecturally UNKNOWN value.

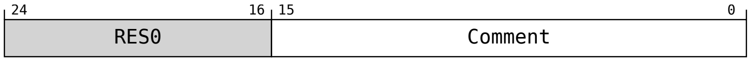
DFSC, bits [5:0]

Data Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x'Watchpoint exceptions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from execution of a Breakpoint instruction



Bits [24:16]

Reserved, RES0.

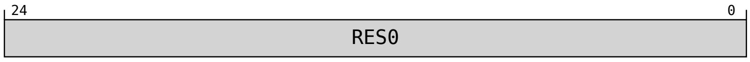
Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary. For the AArch32 BKPT instructions, the comment field is described as the immediate field.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x'Breakpoint instruction exceptions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0



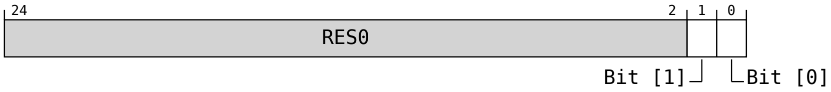
Bits [24:0]

Reserved, RES0.

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

an exception from a Pointer Authentication instruction authentication failure



Bits [24:2]

Reserved, RES0.

Bit [1], bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

Value	Meaning
`0b0`	Instruction Key.
`0b1`	Data Key.

This field resets to an architecturally UNKNOWN value.

Bit [0], bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

Value	Meaning
`0b0`	A key.
`0b1`	B key.

This field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:

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

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a translation fault when the address is accessed:

- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing the ESR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic ESR_EL1 or

ESR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name ESR_EL1

The assembler syntax is:

```
MRS <Xt>, ESR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ESR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return ESR_EL2;
    else
        return ESR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ESR_EL1;
```

Write using name ESR_EL1

The assembler syntax is:

```
MSR ESR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
```

```
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        ESR_EL2 = X[t];
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ESR_EL1 = X[t];
```

Read using name ESR_EL12

The assembler syntax is:

```
MRS <Xt>, ESR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return ESR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return ESR_EL1;
    else
        UNDEFINED;
```

Write using name ESR_EL12

The assembler syntax is:

```
MSR ESR_EL12, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            ESR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            ESR_EL1 = X[t];
    else
        UNDEFINED;
```

3.2.26 ESR_EL2, Exception Syndrome Register (EL2)

The ESR_EL2 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL2.

Attributes

ESR_EL2 is a 64-bit register.

Configuration

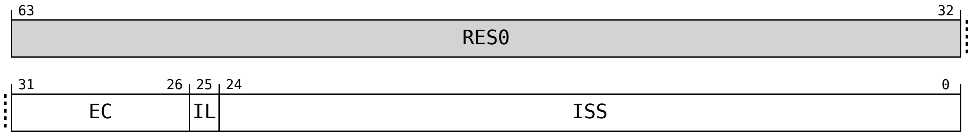
If EL2 is not implemented, this register is RES0 from EL3.

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

AArch64 System register ESR_EL2[31:0] is architecturally mapped to AArch32 System register HSR[31:0].

Field descriptions

The ESR_EL2 bit assignments are:



ESR_EL2 is made UNKNOWN as a result of an exception return from EL2.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL2, the value of ESR_EL2 is UNKNOWN. The value written to ESR_EL2 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

Value	Meaning	Link	Applies
`0b000000`	Unknown reason.	ISS - exceptions with an unknown reason	
`0b000001`	Trapped WFI or WFE instruction execution. Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.	ISS - an exception from a WFI or WFE instruction	

Value	Meaning	Link	Applies
`0b000011`	Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC 0b000000.	ISS - an exception from an MCR or MRC access	
`0b000100`	Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC 0b000000.	ISS - an exception from an MCRR or MRRC access	
`0b000101`	Trapped MCR or MRC access with (coproc==0b1110).	ISS - an exception from an MCR or MRC access	
`0b000110`	Trapped LDC or STC access. The only architected uses of these instruction are: • An STC to write data to memory from DBGDTRRXint. • An LDC to read data from memory to DBGDTRTXint.	ISS - an exception from an LDC or STC instruction	
`0b000111`	Access to SVE, Advanced SIMD, or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000 as described in 'EC encodings when routing exceptions to EL2' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section D1.10.4.	ISS - an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP	
`0b001000`	Trapped VMRS access, from ID group trap, that is not reported using EC 0b000111.	ISS - an exception from an MCR or MRC access	
`0b001100`	Trapped MRRC access with (coproc==0b1110).	ISS - an exception from an MCRR or MRRC access	
`0b001110`	Illegal Execution state.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	
`0b010001`	SVC instruction execution in AArch32 state. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TGE is 1.	ISS - an exception from HVC or SVC instruction execution	
`0b010010`	HVC instruction execution in AArch32 state, when HVC is not disabled.	ISS - an exception from HVC or SVC instruction execution	

Value	Meaning	Link	Applies
`0b010011`	SMC instruction execution in AArch32 state, when SMC is not disabled. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.	ISS - an exception from SMC instruction execution in AArch32 state	
`0b010101`	SVC instruction execution in AArch64 state.	ISS - an exception from HVC or SVC instruction execution	
`0b010110`	HVC instruction execution in AArch64 state, when HVC is not disabled.	ISS - an exception from HVC or SVC instruction execution	
`0b010111`	SMC instruction execution in AArch64 state, when SMC is not disabled. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.	ISS - an exception from SMC instruction execution in AArch64 state	
`0b011000`	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC 0b000000, 0b000001, 0b000111 or 0b101010. If xARMv8.0-CSV2 is implemented, also Cache Speculation Variant exceptions. If xARMv8.2-EVT is implemented, also traps for EL1 and EL0 Cache controls. This includes all instructions that cause exceptions that are part of the encoding space defined in 'System instruction class encoding overview' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section C5.2.2, except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS - an exception from MSR, MRS, or System instruction execution in AArch64 state	
`0b011001`	Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC 0b000000. This EC is defined only if xSVEis implemented.	ISS - an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ	
`0b100000`	Instruction Abort from a lower Exception level, that might be using AArch32 or AArch64. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from an Instruction Abort	

Value	Meaning	Link	Applies
`0b100001`	Instruction Abort taken without a change in Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from an Instruction Abort	
`0b100010`	PC alignment fault exception.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	
`0b100100`	Data Abort from a lower Exception level, excluding Data Aborts taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. These Data Aborts might be generated from Exception levels using AArch32 or AArch64. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from a Data Abort	
`0b100101`	Data Abort without a change in Exception level, or Data Aborts taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from a Data Abort	
`0b100110`	SP alignment fault exception.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	
`0b101000`	Trapped floating-point exception taken from AArch32 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS - an exception from a trapped floating-point exception	

Value	Meaning	Link	Applies
`0b101001`	Access to the Morello architecture trapped as a result of CPACR_EL1.CEN, CPTR_EL2.CEN, CPTR_EL2.TC, or CPTR_EL3.EC.	ISS - an exception from an access to the Morello architecture	When Morello is implemented
`0b101010`	Trapped capability MSR or MRS instruction execution. This EC value is valid if Morello architecture is implemented, otherwise it is reserved. Used for trapped accesses to capability System registers via MSR or MRS instructions.	ISS - an exception from capability MSR or MRS instruction execution	When Morello is implemented
`0b101100`	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS - an exception from a trapped floating-point exception	
`0b101111`	SError interrupt.	ISS - an SError interrupt	
`0b110000`	Breakpoint exception from a lower Exception level, that might be using AArch32 or AArch64.	ISS - an exception from a Breakpoint or Vector Catch debug exception	
`0b110001`	Breakpoint exception taken without a change in Exception level.	ISS - an exception from a Breakpoint or Vector Catch debug exception	
`0b110010`	Software Step exception from a lower Exception level, that might be using AArch32 or AArch64.	ISS - an exception from a Software Step exception	
`0b110011`	Software Step exception taken without a change in Exception level.	ISS - an exception from a Software Step exception	
`0b110100`	Watchpoint from a lower Exception level, excluding Watchpoint Exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support. These Watchpoint Exceptions might be generated from Exception levels using AArch32 or AArch64	ISS - an exception from a Watchpoint exception	
`0b110101`	Watchpoint exceptions without a change in Exception level, or Watchpoint exceptions taken to EL2 as a result of accesses generated associated with VNCR_EL2 as part of nested virtualization support.	ISS - an exception from a Watchpoint exception	
`0b111000`	BKPT instruction execution in AArch32 state.	ISS - an exception from execution of a Breakpoint instruction	

Value	Meaning	Link	Applies
0b111010	Vector Catch exception from AArch32 state. The only case where a Vector Catch exception is taken to an Exception level that is using AArch64 is when the exception is routed to EL2 and EL2 is using AArch64.	ISS - an exception from a Breakpoint or Vector Catch debug exception	
0b111100	BRK instruction execution in AArch64 state. This is reported in ESR_EL3 only if a BRK instruction is executed.	ISS - an exception from execution of a Breakpoint instruction	

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in 'Reserved values in System and memory-mapped registers and translation table entries' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section K1.1.11.

This field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

Value	Meaning
0b0	16-bit instruction trapped.
0b1	32-bit instruction trapped. This value is also used when the exception is one of the following: • An SError interrupt. • An Instruction Abort exception. • A PC alignment fault exception. • An SP alignment fault exception. • A Data Abort exception for which the value of the ISV bit is 0. • An Illegal Execution state exception. • Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning: – 0b0: 16-bit T32 BKPT instruction. – 0b1: 32-bit A32 BKPT instruction or A64 BRK instruction. • An exception reported using EC value 0b000000.

This field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

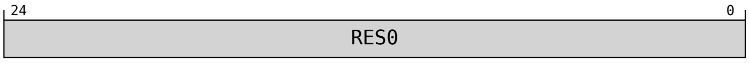
For an exception taken from AArch32 state, see x'Mapping of the general-purpose registers between the Execution states'.

If the AArch32 register descriptor is 0b1111, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value 0b11111.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value 0b11111.

When the EC field is 0b000000, indicating an exception with an unknown reason, the ISS field is not valid, RES0.

exceptions with an unknown reason



Bits [24:0]

Reserved, RES0.

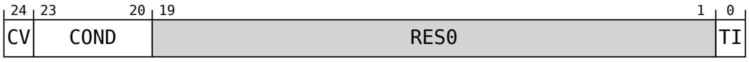
When an exception is reported using this EC code the IL field is set to 1.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.

- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon. See x'Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32'.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of 0b000111.
- When SVE is not implemented, attempted execution of:
  - An SVE instruction.
  - An MSR or MRS instruction to access ZCR_EL1, ZCR_EL2, or ZCR_EL3.

an exception from a WFI or WFE instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.

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

This field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

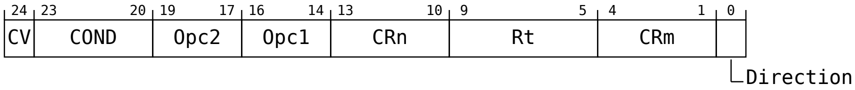
Value	Meaning
0b0	WFI trapped.
0b1	WFE trapped.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

an exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
`0b0`	Write to System register space. MCR instruction.
`0b1`	Read from System register space. MRC or VMRS instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000011:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- For information on other traps using EC value 0b000011, see x'Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32'.
- If xARMv8.6-FGT is implemented, MCR or MRC access to some registers at EL0, trapped to EL2. [endif]

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000101:

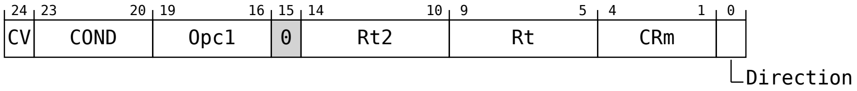
- CPACR_EL1.TTA for accesses to trace registers, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.

- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.
- HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDA, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001000:

- HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

an exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
0b0	Write to System register space. MCRR instruction.
0b1	Read from System register space. MRRC instruction.

This field resets to an architecturally UNKNOWN value.

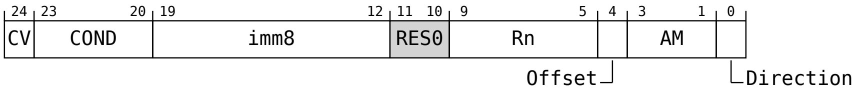
The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000100:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.

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

- CPACR_EL1.TTA for accesses to trace registers using MCR or MRC instructions, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDOSA, for traps to powerdown debug registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.

an exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x‘Mapping of the general-purpose registers between the Execution states’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Value	Meaning
0b0	Subtract offset.
0b1	Add offset.

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

This field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

Value	Meaning
0b000	Immediate unindexed.
0b001	Immediate post-indexed.
0b010	Immediate offset.
0b011	Immediate pre-indexed.
0b100	For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
0b110	For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in x'Reserved values in System and memory-mapped registers and translation table entries'.

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

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

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
0b0	Write to memory. STC instruction.
0b1	Read from memory. LDC instruction.

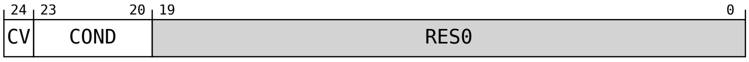
This field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000110:

- MDSCR_EL1.TDCC, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.

- MDCR_EL2.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- MDCR_EL3.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.

an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP



The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include either SVE or support for floating-point and Advanced SIMD, the exception is reported using the EC value 0b000000.

CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is

set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

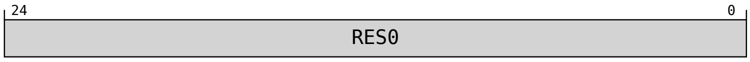
Bits [19:0]

Reserved, RES0.

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

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ



Bits [24:0]

When SVE is implemented:

Reserved, RES0.

Otherwise:

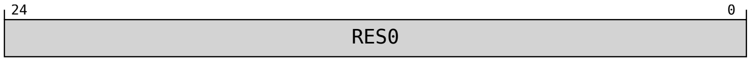
RES0

The accesses covered by this trap include:

- Execution of SVE instructions.
- Accesses to the SVE system registers, ZCR_ELx and ID_AA64ZFR0_EL1.

For an implementation that does not include SVE, the exception is reported using the EC value 0b000000.

an exception from an Illegal Execution state, or a PC or SP alignment fault



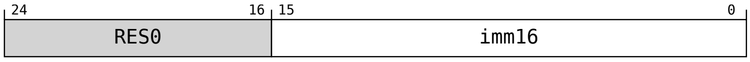
Bits [24:0]

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about these exceptions see x'The Illegal Execution state exception' and x'PC alignment checking'.

x'SP alignment checking' describes the configuration settings for generating SP alignment fault exceptions.

an exception from HVC or SVC instruction execution



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

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

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

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

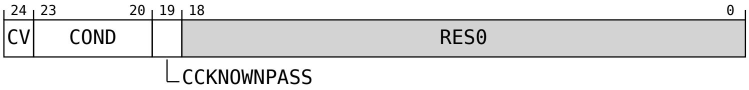
This field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see x'SVC' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile and x'HVC' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

For A64 instructions, see x'SVC' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile and x'HVC' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from SMC instruction execution in AArch32 state



For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

Value	Meaning
`0b0`	The instruction was unconditional, or was conditional and passed its condition code check.
`0b1`	The instruction was conditional, and might have failed its condition code check.

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

This field resets to an architecturally UNKNOWN value.

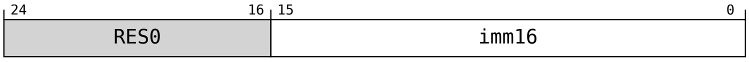
Bits [18:0]

Reserved, RES0.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

x'System calls' describes the case where these exceptions are trapped to EL3.

an exception from SMC instruction execution in AArch64 state



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

This field resets to an architecturally UNKNOWN value.

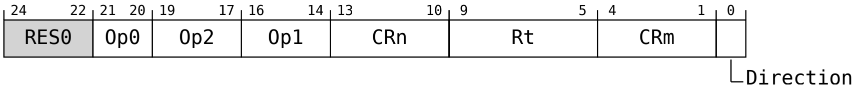
The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

x'System calls' describes the case where these exceptions are trapped to EL3.

an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

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

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
`0b0`	Write access, including MSR instructions.

Value	Meaning
`0b1`	Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

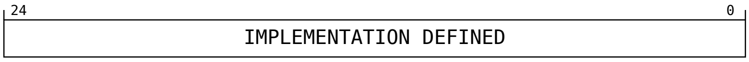
For exceptions caused by System instructions, see x'System instructions' subsection of 'Branches, exception generating and System instructions' for the encoding values returned by an instruction.

The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.

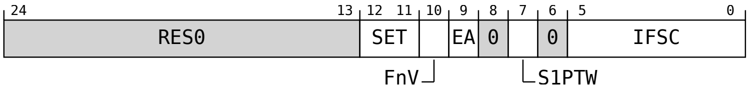
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the filter trace control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If xARMv8.2-EVT is implemented, HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4} and HCR2.{TTLBIS, TICAB, TOCU, TID4} control traps for EL1 and EL0 Cache controls that use this EC value.

an IMPLEMENTATION DEFINED exception to EL3



IMPLEMENTATION DEFINED, bits [24:0] IMPLEMENTATION DEFINED

an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and IFSC is 0b010000, describes the state of the PE after taking the Instruction Abort exception. The possible values of this field are:

Value	Meaning
`0b00`	Recoverable error (UER).
`0b10`	Uncontainable error (UC).
`0b11`	Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the IFSC field is not 0b010000.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

Value	Meaning
`0b0`	FAR is valid.
`0b1`	FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is 0b010000. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Value	Meaning
`0b0`	Fault not on a stage 2 translation for a stage 1 translation table walk.

Value	Meaning
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

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

Value	Meaning
0b000000	Address size fault, level 0 of translation or translation table base register
0b000001	Address size fault, level 1
0b000010	Address size fault, level 2
0b000011	Address size fault, level 3
0b000100	Translation fault, level 0
0b000101	Translation fault, level 1
0b000110	Translation fault, level 2
0b000111	Translation fault, level 3
0b001001	Access flag fault, level 1
0b001010	Access flag fault, level 2
0b001011	Access flag fault, level 3
0b001101	Permission fault, level 1
0b001110	Permission fault, level 2
0b001111	Permission fault, level 3
0b010000	Synchronous External abort, not on translation table walk
0b010100	Synchronous External abort, on translation table walk, level 0
0b010101	Synchronous External abort, on translation table walk, level 1
0b010110	Synchronous External abort, on translation table walk, level 2
0b010111	Synchronous External abort, on translation table walk, level 3
0b011000	Synchronous parity or ECC error on memory access, not on translation table walk
0b011100	Synchronous parity or ECC error on memory access on translation table walk, level 0
0b011101	Synchronous parity or ECC error on memory access on translation table walk, level 1

Value	Meaning
0b011110	Synchronous parity or ECC error on memory access on translation table walk, level 2
0b011111	Synchronous parity or ECC error on memory access on translation table walk, level 3
0b101000	Capability tag fault.
0b101001	Capability sealed fault.
0b101010	Capability bound fault.
0b101011	Capability permission fault.
0b110000	TLB conflict abort
0b110001	Unsupported atomic hardware update fault, if the implementation includes xARMv8.1-TTHM. Otherwise reserved.

All other values are reserved.

When the RAS Extension is implemented, 0b011000, 0b011100, 0b011101, 0b011110, and 0b011111, are reserved.

Armv8.2 requires the implementation of the RAS Extension.

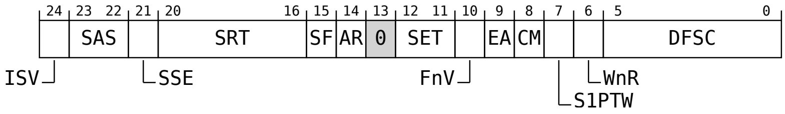
For more information about the lookup level associated with a fault, see x'The level associated with MMU faults' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

an exception from a Data Abort



ISV, bit [24]

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

Value	Meaning
0b0	No valid instruction syndrome. ISS[23:14] are RES0.
0b1	ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with 0b11111, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive, excluding those with writeback and excluding accesses of a capability.
- AArch32 instructions where the instruction:

- Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
- Is not performing register writeback.
- Is not using R15 as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When the RAS Extension is not implemented, the value of ISV on a synchronous External abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

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

Value	Meaning
`0b00`	Byte
`0b01`	Halfword
`0b10`	Word
`0b11`	Doubleword

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

Value	Meaning
`0b0`	Sign-extension not required.
`0b1`	Data item must be sign-extended.

For all other operations this bit is 0.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SRT, bits [20:16]

Syndrome Register transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction. If the exception was taken from an Exception level that is using AArch32 then this is the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SF, bit [15]

Width of the register accessed by the instruction is Sixty-Four. When ISV is 1, the possible values of this bit are:

Value	Meaning
`0b0`	Instruction loads/stores a 32-bit wide register.
`0b1`	Instruction loads/stores a 64-bit wide register.

This field specifies the register width identified by the instruction, not the Execution state.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

Value	Meaning
`0b0`	Instruction did not have acquire/release semantics.
`0b1`	Instruction did have acquire/release semantics.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and DFSC is 0b010000, describes the state of the PE after taking the Data Abort exception. The possible values of this field are:

Value	Meaning
`0b00`	Recoverable error (UER).
`0b10`	Uncontainable error (UC).
`0b11`	Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010000.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

Value	Meaning
`0b0`	FAR is valid.
`0b1`	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

Value	Meaning
`0b0`	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
`0b1`	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Value	Meaning
`0b0`	Fault not on a stage 2 translation for a stage 1 translation table walk.

Value	Meaning
`0b1`	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

Value	Meaning
`0b0`	Abort caused by an instruction reading from a memory location.
`0b1`	Abort caused by an instruction writing to a memory location.

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

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

For Page table LC or SC permission violation faults from an atomic instruction that both reads and writes a valid capability from a memory location, this bit is set to 1 if a write of a valid capability from the memory location would have generated the fault which is being reported, otherwise it is set to 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

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

Value	Meaning
`0b000000`	Address size fault, level 0 of translation or translation table base register.
`0b000001`	Address size fault, level 1.
`0b000010`	Address size fault, level 2.
`0b000011`	Address size fault, level 3.
`0b000100`	Translation fault, level 0.
`0b000101`	Translation fault, level 1.
`0b000110`	Translation fault, level 2.
`0b000111`	Translation fault, level 3.

Value	Meaning
`0b001001`	Access flag fault, level 1.
`0b001010`	Access flag fault, level 2.
`0b001011`	Access flag fault, level 3.
`0b001101`	Permission fault, level 1.
`0b001110`	Permission fault, level 2.
`0b001111`	Permission fault, level 3.
`0b010000`	Synchronous External abort, not on translation table walk.
`0b010001`	Synchronous Tag Check fail
`0b010100`	Synchronous External abort, on translation table walk, level 0.
`0b010101`	Synchronous External abort, on translation table walk, level 1.
`0b010110`	Synchronous External abort, on translation table walk, level 2.
`0b010111`	Synchronous External abort, on translation table walk, level 3.
`0b011000`	Synchronous parity or ECC error on memory access, not on translation table walk.
`0b011100`	Synchronous parity or ECC error on memory access on translation table walk, level 0.
`0b011101`	Synchronous parity or ECC error on memory access on translation table walk, level 1.
`0b011110`	Synchronous parity or ECC error on memory access on translation table walk, level 2.
`0b011111`	Synchronous parity or ECC error on memory access on translation table walk, level 3.
`0b100001`	Alignment fault.
`0b101000`	Capability tag fault.
`0b101001`	Capability sealed fault.
`0b101010`	Capability bound fault.
`0b101011`	Capability permission fault.
`0b101100`	Page table LC or SC permission violation fault.
`0b110000`	TLB conflict abort.
`0b110001`	Unsupported atomic hardware update fault, if the implementation includes xARMv8.1-TTHM. Otherwise reserved.
`0b110100`	IMPLEMENTATION DEFINED fault (Lockdown).
`0b110101`	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).
`0b110110`	Unsupported LDCT or SDCT to Device or Non-cacheable.
`0b111101`	Section Domain Fault, used only for faults reported in the PAR_EL1.
`0b111110`	Page Domain Fault, used only for faults reported in the PAR_EL1.

All other values are reserved.

When the RAS Extension is implemented, 0b011000, 0b011100, 0b011101, 0b011110, and 0b011111, are reserved.

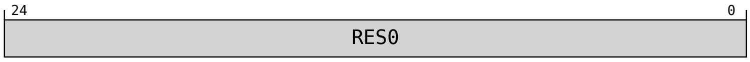
For more information about the lookup level associated with a fault, see x'The level associated with MMU faults' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

an exception from an access to the Morello architecture

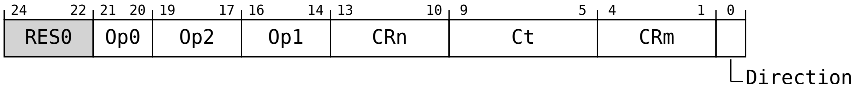


Bits [24:0]

Reserved, RES0.

In an implementation that supports Morello architecture, from an Exception level using AArch64, the CPACR_EL1.CEN, CPTR_EL2.{CEN, DC} and CPTR_EL3.EC bits control whether Morello instructions and accesses to Morello System registers are trapped.

an exception from capability MSR or MRS instruction execution



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Ct, bits [9:5]

The Ct value from the issued instruction, the capability register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

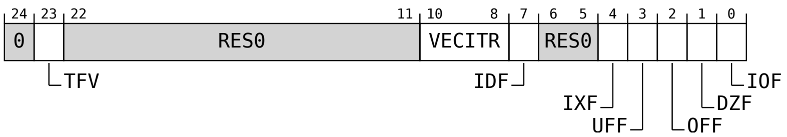
Direction, bit [0]

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

Value	Meaning
`0b0`	Write access, including MSR instructions.
`0b1`	Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions. The possible values of this bit are:

Value	Meaning
`0b0`	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
`0b1`	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information see x'Floating- point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating point exception from a vector instruction.

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from a vector instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

This field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Input denormal floating-point exception has not occurred.
`0b1`	Input denormal floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Inexact floating-point exception has not occurred.
`0b1`	Inexact floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Underflow floating-point exception has not occurred.
`0b1`	Underflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Overflow floating-point exception has not occurred.
`0b1`	Overflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Divide by Zero floating-point exception has not occurred.
`0b1`	Divide by Zero floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

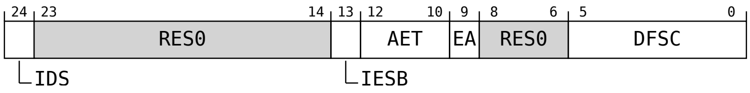
Value	Meaning
`0b0`	Invalid Operation floating-point exception has not occurred.
`0b1`	Invalid Operation floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

an SError interrupt



IDS, bit [24]

IMPLEMENTATION DEFINED syndrome. Possible values of this bit are:

Value	Meaning
`0b0`	Bits[23:0] of the ISS field holds the fields described in this encoding. If the RAS Extension is not implemented, this means that bits[23:0] of the ISS field are RES0.
`0b1`	Bits[23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.

This field was previously called ISV.

This field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

When ARMv8.2-IESB is implemented:

Implicit error synchronization event.

Value	Meaning
`0b0`	The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.
`0b1`	The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

This field is RES0 if the value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension and xARMv8.2-IESB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

AET, bits [12:10]

Asynchronous Error Type.

When the RAS Extension is implemented and DFSC is 0b010001, describes the state of the PE after taking the SError interrupt exception. The possible values of this field are:

Value	Meaning
`0b000`	Uncontainable error (UC).
`0b001`	Unrecoverable error (UEU).
`0b010`	Restartable error (UEO).
`0b011`	Recoverable error (UER).
`0b110`	Corrected error (CE).

All other values are reserved.

If multiple errors are taken as a single SError interrupt exception, the overall state of the PE is reported. For example, if both a Recoverable and Unrecoverable error occurred, the state is Unrecoverable.

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. When the RAS Extension is implemented, this bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

Data Fault Status Code. When the RAS Extension is implemented, possible values of this field are:

Value	Meaning
`0b000000`	Uncategorized.
`0b010001`	Asynchronous SError interrupt.

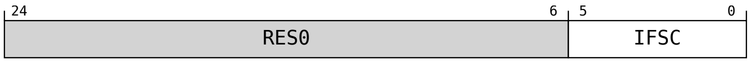
All other values are reserved.

If the RAS Extension is not implemented, this field is RES0.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

an exception from a Breakpoint or Vector Catch debug exception



Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

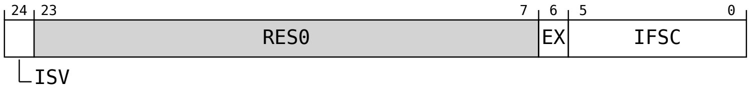
Instruction Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions:

- For exceptions from AArch64, see x'Breakpoint exceptions'.
- For exceptions from AArch32, see x'Breakpoint exceptions' and x'Vector Catch exceptions'.

an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

Value	Meaning
`0b0`	EX bit is RES0.
`0b1`	EX bit is valid.

See the EX bit description for more information.

This field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

Value	Meaning
`0b0`	An instruction other than a Load- Exclusive instruction was stepped.
`0b1`	A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

This field resets to an architecturally UNKNOWN value.

IFSC, bits [5:0]

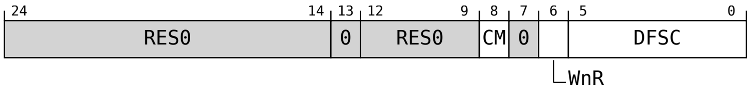
Instruction Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x'Software Step exceptions' in the Arm® Architecture

Reference Manual, Armv8, for Armv8-A architecture profile,.

an exception from a Watchpoint exception



Bits [24:14]

Reserved, RES0.

Bit [13]

Reserved, RES0.

Bits [12:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance or address translation instruction:

Value	Meaning
0b0	The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
0b1	The Watchpoint exception was generated by either the execution of a cache maintenance instruction or by a synchronous Watchpoint exception on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

Value	Meaning
0b0	Watchpoint exception caused by an instruction reading from a memory location.
0b1	Watchpoint exception caused by an instruction writing to a memory location.

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

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

This field resets to an architecturally UNKNOWN value.

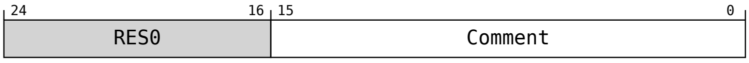
DFSC, bits [5:0]

Data Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x'Watchpoint exceptions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from execution of a Breakpoint instruction



Bits [24:16]

Reserved, RES0.

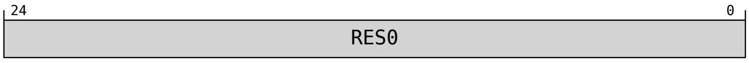
Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary. For the AArch32 BKPT instructions, the comment field is described as the immediate field.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x'Breakpoint instruction exceptions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0



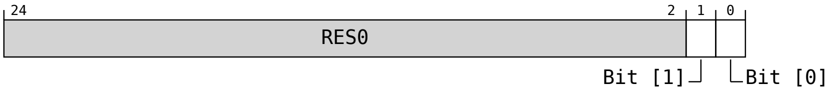
Bits [24:0]

Reserved, RES0.

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

an exception from a Pointer Authentication instruction authentication failure



Bits [24:2]

Reserved, RES0.

Bit [1], bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

Value	Meaning
`0b0`	Instruction Key.
`0b1`	Data Key.

This field resets to an architecturally UNKNOWN value.

Bit [0], bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

Value	Meaning
`0b0`	A key.
`0b1`	B key.

This field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:

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

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a translation fault when the address is accessed:

- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing the ESR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic ESR_EL2 or ESR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name ESR_EL2

The assembler syntax is:

```
MRS <Xt>, ESR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
```

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ESR_EL2;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ESR_EL2;
```

Write using name ESR_EL2

The assembler syntax is:

```
MSR ESR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ESR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ESR_EL2 = X[t];
```

Read using name ESR_EL1

The assembler syntax is:

```
MRS <Xt>, ESR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ESR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return ESR_EL2;
    else
        return ESR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ESR_EL1;
```

Write using name ESR_EL1

The assembler syntax is:

```
MSR ESR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        ESR_EL2 = X[t];
    else
        ESR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```

```
        ESR_EL1 = X[t];
```

3.2.27 ESR_EL3, Exception Syndrome Register (EL3)

The ESR_EL3 characteristics are:

Purpose

Holds syndrome information for an exception taken to EL3.

Attributes

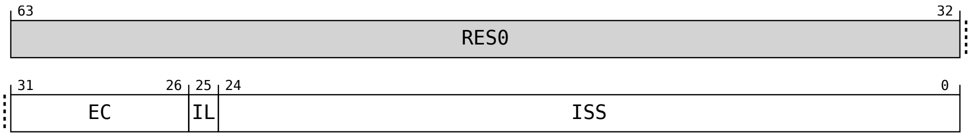
ESR_EL3 is a 64-bit register.

Configuration

This register is present only when HaveEL(EL3). Otherwise, direct accesses to ESR_EL3 are UNDEFINED.

Field descriptions

The ESR_EL3 bit assignments are:



ESR_EL3 is made UNKNOWN as a result of an exception return from EL3.

When an UNPREDICTABLE instruction is treated as UNDEFINED, and the exception is taken to EL3, the value of ESR_EL3 is UNKNOWN. The value written to ESR_EL3 must be consistent with a value that could be created as a result of an exception from the same Exception level that generated the exception as a result of a situation that is not UNPREDICTABLE at that Exception level, in order to avoid the possibility of a privilege violation.

Bits [63:32]

Reserved, RES0.

EC, bits [31:26]

Exception Class. Indicates the reason for the exception that this register holds information about.

For each EC value, the table references a subsection that gives information about:

- The cause of the exception, for example the configuration required to enable the trap.
- The encoding of the associated ISS.

Possible values of the EC field are:

Value	Meaning	Link	Applies
`0b000000`	Unknown reason.	ISS - exceptions with an unknown reason	
`0b000001`	Trapped WFI or WFE instruction execution. Conditional WFE and WFI instructions that fail their condition code check do not cause an exception.	ISS - an exception from a WFI or WFE instruction	
`0b000011`	Trapped MCR or MRC access with (coproc==0b1111) that is not reported using EC 0b000000.	ISS - an exception from an MCR or MRC access	

Value	Meaning	Link	Applies
`0b000100`	Trapped MCRR or MRRC access with (coproc==0b1111) that is not reported using EC 0b000000.	ISS - an exception from an MCRR or MRRC access	
`0b000101`	Trapped MCR or MRC access with (coproc==0b1110).	ISS - an exception from an MCR or MRC access	
`0b000110`	Trapped LDC or STC access. The only architected uses of these instruction are: • An STC to write data to memory from DBGDTRRXint. • An LDC to read data from memory to DBGDTRTXint.	ISS - an exception from an LDC or STC instruction	
`0b000111`	Access to SVE, Advanced SIMD, or floating-point functionality trapped by CPACR_EL1.FPEN, CPTR_EL2.FPEN, CPTR_EL2.TFP, or CPTR_EL3.TFP control. Excludes exceptions resulting from CPACR_EL1 when the value of HCR_EL2.TGE is 1, or because SVE or Advanced SIMD and floating-point are not implemented. These are reported with EC value 0b000000 as described in ’EC encodings when routing exceptions to EL2’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section D1.10.4.	ISS - an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP	
`0b001100`	Trapped MRRC access with (coproc==0b1110).	ISS - an exception from an MCRR or MRRC access	
`0b001110`	Illegal Execution state.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	
`0b010011`	SMC instruction execution in AArch32 state, when SMC is not disabled. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.	ISS - an exception from SMC instruction execution in AArch32 state	
`0b010101`	SVC instruction execution in AArch64 state.	ISS - an exception from HVC or SVC instruction execution	
`0b010110`	HVC instruction execution in AArch64 state, when HVC is not disabled.	ISS - an exception from HVC or SVC instruction execution	
`0b010111`	SMC instruction execution in AArch64 state, when SMC is not disabled. This is reported in ESR_EL2 only when the exception is generated because the value of HCR_EL2.TSC is 1.	ISS - an exception from SMC instruction execution in AArch64 state	

Value	Meaning	Link	Applies
0b011000	Trapped MSR, MRS or System instruction execution in AArch64 state, that is not reported using EC 0b000000, 0b000001, 0b000111 or 0b101010. If xARMv8.0-CSV2 is implemented, also Cache Speculation Variant exceptions. This includes all instructions that cause exceptions that are part of the encoding space defined in 'System instruction class encoding overview' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section C5.2.2, except for those exceptions reported using EC values 0b000000, 0b000001, or 0b000111.	ISS - an exception from MSR, MRS, or System instruction execution in AArch64 state	
0b011001	Access to SVE functionality trapped as a result of CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ, that is not reported using EC 0b000000. This EC is defined only if xSVEis implemented.	ISS - an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ	
0b011111	IMPLEMENTATION DEFINED exception to EL3.	ISS - an IMPLEMENTATION DEFINED exception to EL3	
0b100000	Instruction Abort from a lower Exception level, that might be using AArch32 or AArch64. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from an Instruction Abort	
0b100001	Instruction Abort taken without a change in Exception level. Used for MMU faults generated by instruction accesses and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from an Instruction Abort	
0b100010	PC alignment fault exception.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	

Value	Meaning	Link	Applies
`0b100100`	Data Abort from a lower Exception level, that might be using AArch32 or AArch64. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from a Data Abort	
`0b100101`	Data Abort taken without a change in Exception level. Used for MMU faults generated by data accesses, alignment faults other than those caused by Stack Pointer misalignment, and synchronous External aborts, including synchronous parity or ECC errors. Not used for debug related exceptions.	ISS - an exception from a Data Abort	
`0b100110`	SP alignment fault exception.	ISS - an exception from an Illegal Execution state, or a PC or SP alignment fault	
`0b101001`	Access to the Morello architecture trapped as a result of CPACR_EL1.CEN, CPTR_EL2.CEN, CPTR_EL2.TC, or CPTR_EL3.EC.	ISS - an exception from an access to the Morello architecture	When Morello is implemented
`0b101010`	Trapped capability MSR or MRS instruction execution. This EC value is valid if Morello architecture is implemented, otherwise it is reserved. Used for trapped accesses to capability System registers via MSR or MRS instructions.	ISS - an exception from capability MSR or MRS instruction execution	When Morello is implemented
`0b101100`	Trapped floating-point exception taken from AArch64 state. This EC value is valid if the implementation supports trapping of floating-point exceptions, otherwise it is reserved. Whether a floating-point implementation supports trapping of floating-point exceptions is IMPLEMENTATION DEFINED.	ISS - an exception from a trapped floating-point exception	
`0b101111`	SError interrupt.	ISS - an SError interrupt	
`0b111100`	BRK instruction execution in AArch64 state. This is reported in ESR_EL3 only if a BRK instruction is executed.	ISS - an exception from execution of a Breakpoint instruction	

All other EC values are reserved by Arm, and:

- Unused values in the range 0b000000 - 0b101100 (0x00 - 0x2C) are reserved for future use for synchronous exceptions.
- Unused values in the range 0b101101 - 0b111111 (0x2D - 0x3F) are reserved for future use, and might be used for synchronous or asynchronous exceptions.

The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in 'Reserved values in System and memory-mapped registers and translation table entries' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile, section K1.1.11.

This field resets to an architecturally UNKNOWN value.

IL, bit [25]

Instruction Length for synchronous exceptions. Possible values of this bit are:

Value	Meaning
`0b0`	16-bit instruction trapped.
`0b1`	32-bit instruction trapped. This value is also used when the exception is one of the following: • An SError interrupt. • An Instruction Abort exception. • A PC alignment fault exception. • An SP alignment fault exception. • A Data Abort exception for which the value of the ISV bit is 0. • An Illegal Execution state exception. • Any debug exception except for Breakpoint instruction exceptions. For Breakpoint instruction exceptions, this bit has its standard meaning: – 0b0: 16-bit T32 BKPT instruction. – 0b1: 32-bit A32 BKPT instruction or A64 BRK instruction. • An exception reported using EC value 0b000000.

This field resets to an architecturally UNKNOWN value.

ISS, bits [24:0]

Instruction Specific Syndrome. Architecturally, this field can be defined independently for each defined Exception class. However, in practice, some ISS encodings are used for more than one Exception class.

Typically, an ISS encoding has a number of subfields. When an ISS subfield holds a register number, the value returned in that field is the AArch64 view of the register number.

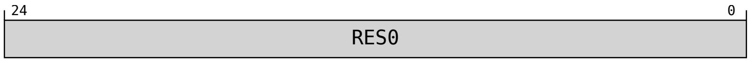
For an exception taken from AArch32 state, see x'Mapping of the general-purpose registers between the Execution states'.

If the AArch32 register descriptor is 0b1111, then:

- If the instruction that generated the exception was not UNPREDICTABLE, the field takes the value 0b11111.
- If the instruction that generated the exception was UNPREDICTABLE, the field takes an UNKNOWN value that must be either:
  - The AArch64 view of the register number of a register that might have been used at the Exception level from which the exception was taken.
  - The value 0b11111.

When the EC field is 0b000000, indicating an exception with an unknown reason, the ISS field is not valid, RES0.

exceptions with an unknown reason



Bits [24:0]

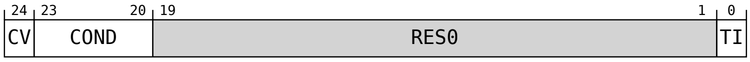
Reserved, RES0.

When an exception is reported using this EC code the IL field is set to 1.

This EC code is used for all exceptions that are not covered by any other EC value. This includes exceptions that are generated in the following situations:

- The attempted execution of an instruction bit pattern that has no allocated instruction or that is not accessible at the current Exception level and Security state, including:
  - A read access using a System register pattern that is not allocated for reads or that does not permit reads at the current Exception level and Security state.
  - A write access using a System register pattern that is not allocated for writes or that does not permit writes at the current Exception level and Security state.
  - Instruction encodings that are unallocated.
  - Instruction encodings for instructions that are not implemented in the implementation.
- In Debug state, the attempted execution of an instruction bit pattern that is not accessible in Debug state.
- In Non-debug state, the attempted execution of an instruction bit pattern that is not accessible in Non-debug state.
- In AArch32 state, attempted execution of a short vector floating-point instruction.
- In an implementation that does not include Advanced SIMD and floating-point functionality, an attempted access to Advanced SIMD or floating-point functionality under conditions where that access would be permitted if that functionality was present. This includes the attempted execution of an Advanced SIMD or floating-point instruction, and attempted accesses to Advanced SIMD and floating-point System registers.
- An exception generated because of the value of one of the SCTLR_EL1.{ITD, SED, CP15BEN} control bits.
- Attempted execution of:
  - An HVC instruction when disabled by HCR_EL2.HCD or SCR_EL3.HCE.
  - An SMC instruction when disabled by SCR_EL3.SMD.
  - An HLT instruction when disabled by EDSCR.HDE.
- Attempted execution of an MSR or MRS instruction to access SP_EL0 when the value of SPSel.SP is 0.
- Attempted execution, in Debug state, of:
  - A DCPS1 instruction when the value of HCR_EL2.TGE is 1 and EL2 is disabled or not implemented in the current Security state.
  - A DCPS2 instruction from EL1 or EL0 when EL2 is disabled or not implemented in the current Security state.
  - A DCPS3 instruction when the value of EDSCR.SDD is 1, or when EL3 is not implemented.
- When EL3 is using AArch64, attempted execution from Secure EL1 of an SRS instruction using R13_mon. See x'Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32'.
- In Debug state when the value of EDSCR.SDD is 1, the attempted execution at EL2, EL1, or EL0 of an instruction that is configured to trap to EL3.
- In AArch32 state, the attempted execution of an MRS (banked register) or an MSR (banked register) instruction to SPSR_mon, SP_mon, or LR_mon.
- An exception that is taken to EL2 because the value of HCR_EL2.TGE is 1 that, if the value of HCR_EL2.TGE was 0 would have been reported with an ESR_ELx.EC value of 0b000111.
- When SVE is not implemented, attempted execution of:
  - An SVE instruction.
  - An MSR or MRS instruction to access ZCR_EL1, ZCR_EL2, or ZCR_EL3.

an exception from a WFI or WFE instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Bits [19:1]

Reserved, RES0.

TI, bit [0]

Trapped instruction. Possible values of this bit are:

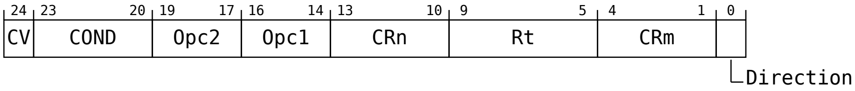
Value	Meaning
`0b0`	WFI trapped.
`0b1`	WFE trapped.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating this exception:

- SCTLR_EL1.{nTWE, nTWI}.
- HCR_EL2.{TWE, TWI}.
- SCR_EL3.{TWE, TWI}.

an exception from an MCR or MRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.

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

This field resets to an architecturally UNKNOWN value.

Opc2, bits [19:17]

The Opc2 value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Opc1, bits [16:14]

The Opc1 value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x‘Mapping of the general-purpose registers between the Execution states’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

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

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
`0b0`	Write to System register space. MCR instruction.
`0b1`	Read from System register space. MRC or VMRS instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000011:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.

- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers from EL0 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU} for execution of cache maintenance instructions at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register at EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1 or CPACR using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CNTHCTL_EL2.EL1PCEN, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers from EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL3.TCPAC, for accesses to CPACR from EL1 and EL2, and accesses to HCPTR from EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCR or MRC access (coproc == 0b1111) trapped to EL3.
- For information on other traps using EC value 0b000011, see x'Traps to EL3 of Secure monitor functionality from Secure EL1 using AArch32'.
- If xARMv8.6-FGT is implemented, MCR or MRC access to some registers at EL0, trapped to EL2. [endif]

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000101:

- CPACR_EL1.TTA for accesses to trace registers, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers at EL0 and EL1 using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL1 or EL2.
- HCR_EL2.TID0, for accesses to the JIDR register in the ID group 0 at EL0 and EL1 using AArch32, MRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDA, for accesses to other debug registers, using AArch32 state, MCR or MRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers using AArch32, MCR or MRC access

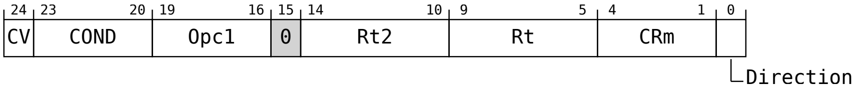
(coproc == 0b1110) trapped to EL3.

- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCR or MRC access (coproc == 0b1110) trapped to EL3.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b001000:

- HCR_EL2.TID0, for accesses to the FPSID register in ID group 0 at EL1 using AArch32 state, VMRS access trapped to EL2.
- HCR_EL2.TID3, for accesses to registers in ID group 3 including MVFR0, MVFR1 and MVFR2, VMRS access trapped to EL2.

an exception from an MCRR or MRRC access



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is

set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

Opc1, bits [19:16]

The Opc1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bit [15]

Reserved, RES0.

Rt2, bits [14:10]

The Rt2 value from the issued instruction, the second general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

The Rt value from the issued instruction, the first general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
`0b0`	Write to System register space. MCRR instruction.
`0b1`	Read from System register space. MRRC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe configuration settings for generating exceptions that are reported using EC value 0b000100:

- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN}, for accesses to the Generic Timer Registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- PMUSERENR_EL0.{CR, EN}, for accesses to Performance Monitor registers from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- AMUSERENR_EL0.{EN}, for accesses to Activity Monitors registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers from EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- HSTR_EL2.T<n>, for accesses to System registers using AArch32 state, MCRR or MRRC access (coproc

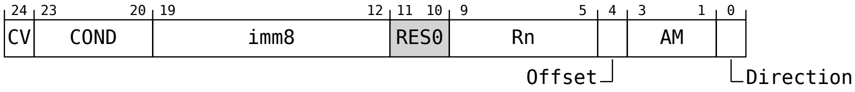
== 0b1111) trapped to EL2.

- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers registers AMEVCNTR0<n> and AMEVCNTR1<n> from EL0 and EL1 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL2.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers from EL0, EL1 and EL2 using AArch32 state, MCRR or MRRC access (coproc == 0b1111) trapped to EL3.

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

- CPACR_EL1.TTA for accesses to trace registers using MCR or MRC instructions, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers DBGDSAR and DBGDRAR at EL0 using AArch32 state, MCRR or MRRC access (coproc == 0b1110) trapped to EL1 or EL2.
- CPTR_EL2.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers DBGDRAR and AArch-DBGDSAR using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL2.
- CPTR_EL3.TTA, for accesses to trace registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDOSA, for traps to powerdown debug registers using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.
- MDCR_EL3.TDA, for accesses to other debug registers, using AArch32, MCRR or MRRC access (coproc == 0b1110) trapped to EL3.

an exception from an LDC or STC instruction



CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

imm8, bits [19:12]

The immediate value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Bits [11:10]

Reserved, RES0.

Rn, bits [9:5]

The Rn value from the issued instruction, the general-purpose register used for the transfer. The reported value gives the AArch64 view of the register. See x‘Mapping of the general-purpose registers between the Execution states’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

This field is valid only when AM[2] is 0, indicating an immediate form of the LDC or STC instruction. When AM[2] is 1, indicating a literal form of the LDC or STC instruction, this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

Offset, bit [4]

Indicates whether the offset is added or subtracted:

Value	Meaning
`0b0`	Subtract offset.
`0b1`	Add offset.

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

This field resets to an architecturally UNKNOWN value.

AM, bits [3:1]

Addressing mode. The permitted values of this field are:

Value	Meaning
0b000	Immediate unindexed.
0b001	Immediate post-indexed.
0b010	Immediate offset.
0b011	Immediate pre-indexed.
0b100	For a trapped STC instruction or a trapped T32 LDC instruction this encoding is reserved.
0b110	For a trapped STC instruction, this encoding is reserved.

The values 0b101 and 0b111 are reserved. The effect of programming this field to a reserved value is that behavior is CONSTRAINED UNPREDICTABLE, as described in x'Reserved values in System and memory-mapped registers and translation table entries'.

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

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

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

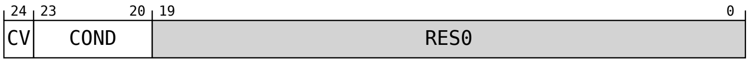
Value	Meaning
0b0	Write to memory. STC instruction.
0b1	Read from memory. LDC instruction.

This field resets to an architecturally UNKNOWN value.

The following fields describe the configuration settings for the traps that are reported using EC value 0b000110:

- MDSCR_EL1.TDCC, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint trapped to EL1 or EL2.
- MDCR_EL2.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL2.
- MDCR_EL3.TDA, for accesses using AArch32 state, LDC access to DBGDTRTXint or STC access to DBGDTRRXint MCR or MRC access trapped to EL3.

an exception from an access to SVE, Advanced SIMD or floating-point functionality, resulting from CPACR_EL1.FPEN, CPTR_EL2.FPEN or CPTR_ELx.TFP



The accesses covered by this trap include:

- Execution of SVE or Advanced SIMD and floating-point instructions.
- Accesses to the Advanced SIMD and floating-point System registers.

For an implementation that does not include either SVE or support for floating-point and Advanced SIMD, the exception is reported using the EC value 0b000000.

CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
`0b0`	The COND field is not valid.
`0b1`	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field resets to an architecturally UNKNOWN value.

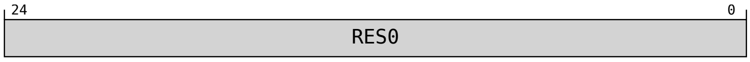
Bits [19:0]

Reserved, RES0.

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

- CPACR_EL1.FPEN, for accesses to SIMD and floating-point registers trapped to EL1.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL2.
- CPTR_EL2.TFP, for accesses to SIMD and floating-point registers trapped to EL3.

an exception from an access to SVE functionality, resulting from CPACR_EL1.ZEN, CPTR_EL2.ZEN, CPTR_EL2.TZ, or CPTR_EL3.EZ



Bits [24:0]

When SVE is implemented:

Reserved, RES0.

Otherwise:

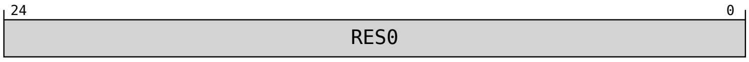
RES0

The accesses covered by this trap include:

- Execution of SVE instructions.
- Accesses to the SVE system registers, ZCR_ELx and ID_AA64ZFR0_EL1.

For an implementation that does not include SVE, the exception is reported using the EC value 0b000000.

an exception from an Illegal Execution state, or a PC or SP alignment fault



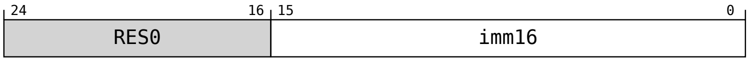
Bits [24:0]

Reserved, RES0.

There are no configuration settings for generating Illegal Execution state exceptions and PC alignment fault exceptions. For more information about these exceptions see x'The Illegal Execution state exception' and x'PC alignment checking'.

x'SP alignment checking' describes the configuration settings for generating SP alignment fault exceptions.

an exception from HVC or SVC instruction execution



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

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

For an HVC instruction, and for an A64 SVC instruction, this is the value of the imm16 field of the issued instruction.

For an A32 or T32 SVC instruction:

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

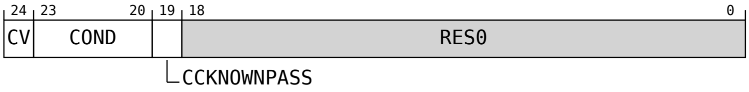
This field resets to an architecturally UNKNOWN value.

In AArch32 state, the HVC instruction is unconditional, and a conditional SVC instruction generates an exception only if it passes its condition code check. Therefore, the syndrome information for these exceptions does not require conditionality information.

For T32 and A32 instructions, see x'SVC' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile and x'HVC' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

For A64 instructions, see x‘SVC’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile and x‘HVC’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from SMC instruction execution in AArch32 state



For an SMC instruction that completes normally and generates an exception that is taken to EL3, the ISS encoding is RES0.

For an SMC instruction that is trapped to EL2 from EL1 because HCR_EL2.TSC is 1, the ISS encoding is as shown in the diagram.

CV, bit [24]

Condition code valid. Possible values of this bit are:

Value	Meaning
0b0	The COND field is not valid.
0b1	The COND field is valid.

For exceptions taken from AArch64, CV is set to 1.

For exceptions taken from AArch32:

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

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

COND, bits [23:20]

The condition code for the trapped instruction. This field is valid only for exceptions taken from AArch32, and only when the value of CV is 1.

For exceptions taken from AArch64, this field is set to 0b1110.

For exceptions taken from AArch32:

- When an A32 instruction is trapped, CV is set to 1 and:
  - If the instruction is conditional, COND is set to the condition code field value from the instruction.
  - If the instruction is unconditional, COND is set to 0b1110.
- A conditional A32 instruction that is known to pass its condition code check can be presented either:
  - With COND set to 0b1110, the value for unconditional.
  - With the COND value held in the instruction.
- When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
  - CV is set to 0 and COND is set to an UNKNOWN value. Software must examine the SPSR.IT field to determine the condition, if any, of the T32 instruction.
  - CV is set to 1 and COND is set to the condition code for the condition that applied to the instruction.
- For an implementation that, for both A32 and T32 instructions, takes an exception on a trapped conditional instruction only if the instruction passes its condition code check, these definitions mean that when CV is set to 1 it is IMPLEMENTATION DEFINED whether the COND field is set to 0b1110, or to the value of any condition that applied to the instruction.

This field is only valid if CCKNOWNPASS is 1, otherwise it is RES0.

This field resets to an architecturally UNKNOWN value.

CCKNOWNPASS, bit [19]

Indicates whether the instruction might have failed its condition code check.

Value	Meaning
`0b0`	The instruction was unconditional, or was conditional and passed its condition code check.
`0b1`	The instruction was conditional, and might have failed its condition code check.

In an implementation in which an SMC instruction that fails it code check is not trapped, this field can always return the value 0.

This field resets to an architecturally UNKNOWN value.

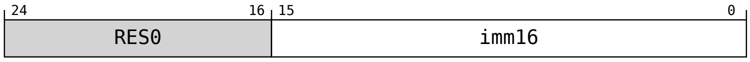
Bits [18:0]

Reserved, RES0.

HCR_EL2.TSC describes the configuration settings for trapping SMC instructions to EL2.

x'System calls' describes the case where these exceptions are trapped to EL3.

an exception from SMC instruction execution in AArch64 state



Bits [24:16]

Reserved, RES0.

imm16, bits [15:0]

The value of the immediate field from the issued SMC instruction.

This field resets to an architecturally UNKNOWN value.

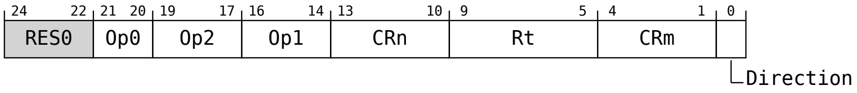
The value of ISS[24:0] described here is used both:

- When an SMC instruction is trapped from EL1 modes.
- When an SMC instruction is not trapped, so completes normally and generates an exception that is taken to EL3.

HCR_EL2.TSC describes the configuration settings for trapping SMC from EL1 modes.

x'System calls' describes the case where these exceptions are trapped to EL3.

an exception from MSR, MRS, or System instruction execution in AArch64 state



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Rt, bits [9:5]

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

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Direction, bit [0]

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

Value	Meaning
`0b0`	Write access, including MSR instructions.
`0b1`	Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

For exceptions caused by System instructions, see x‘System instructions’ subsection of ‘Branches, exception generating and System instructions’ for the encoding values returned by an instruction.

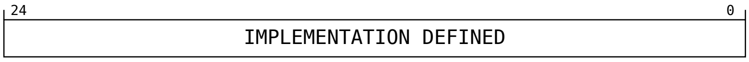
The following fields describe configuration settings for generating the exception that is reported using EC value 0b011000:

- SCTLR_EL1.UCI, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UCT, for accesses to CTR_EL0 using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.DZE, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- SCTLR_EL1.UMA, for accesses to the PSTATE interrupt masks using AArch64 state, MSR or MRS access trapped to EL1 or EL2.

- CPACR_EL1.TTA, for accesses to the trace registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- MDSCR_EL1.TDCC, for accesses to the Debug Communications Channel (DCC) registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- CNTKCTL_EL1.{EL0PTEN, EL0VTEN, EL0PCTEN, EL0VCTEN} accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- PMUSERENR_EL0.{ER, CR, SW, EN}, for accesses to the Performance Monitor registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- AMUSERENR_EL0.EN, for accesses to Activity Monitors registers using AArch64 state, MSR or MRS access trapped to EL1 or EL2.
- HCR_EL2.{TRVM, TVM}, for accesses to virtual memory control registers using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TDZ, for execution of DC ZVA instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TTLB, for execution of TLB maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TSW, TPC, TPU}, for execution of cache maintenance instructions using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TACR, for accesses to the Auxiliary Control Register, ACTLR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.TIDCP, for accesses to lockdown, DMA, and TCM operations using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{TID1, TID2, TID3}, for accesses to ID group 1, ID group 2 or ID group 3 registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TCPAC, for accesses to CPACR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TTRF, for accesses to the trace filter register, TRFCR_EL1, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDRA, for accesses to Debug ROM registers, using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDOSA, for accesses to powerdown debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- CNTHCTL_EL2.{EL1PCEN, EL1PCTEN}, for accesses to the Generic Timer registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.TDA, for accesses to debug registers using AArch64 state, MSR or MRS access trapped to EL2.
- MDCR_EL2.{TPM, TPMCR}, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL2.
- CPTR_EL2.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.APK, for accesses to Pointer authentication key registers. using AArch64 state, MSR or MRS access trapped to EL2.
- HCR_EL2.{NV, NV1}, for Nested virtualization register access, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.AT, for execution of AT S1E* instructions, using AArch64 state, MSR or MRS access, trapped to EL2.
- HCR_EL2.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access, trapped to EL2.
- SCR_EL3.APK, for accesses to Pointer authentication key registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.ST, for accesses to the Counter-timer Physical Secure timer registers, using AArch64 state, MSR or MRS access trapped to EL3.
- SCR_EL3.{TERR, FIEN}, for accesses to RAS registers, using AArch64 state, MSR or MRS access trapped to EL3.

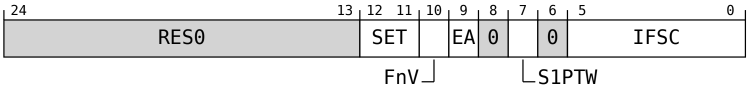
- CPTR_EL3.TCPAC, for accesses to CPTR_EL2 and CPACR_EL1 using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TTA, for accesses to the trace registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TTRF, for accesses to the filter trace control registers, TRFCR_EL1 and TRFCR_EL2, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDA, for accesses to debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TDOSA, for accesses to powerdown debug registers, using AArch64 state, MSR or MRS access trapped to EL3.
- MDCR_EL3.TPM, for accesses to Performance Monitor registers, using AArch64 state, MSR or MRS access trapped to EL3.
- CPTR_EL3.TAM, for accesses to Activity Monitors registers, using AArch64 state, MSR or MRS access, trapped to EL3.
- If xARMv8.2-EVT is implemented, HCR_EL2.{TTLBOS, TTLBIS, TICAB, TOCU, TID4} and HCR2.{TTLBIS, TICAB, TOCU, TID4} control traps for EL1 and EL0 Cache controls that use this EC value.

an IMPLEMENTATION DEFINED exception to EL3



IMPLEMENTATION DEFINED, bits [24:0] IMPLEMENTATION DEFINED

an exception from an Instruction Abort



Bits [24:13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and IFSC is 0b010000, describes the state of the PE after taking the Instruction Abort exception. The possible values of this field are:

Value	Meaning
0b00	Recoverable error (UER).
0b10	Uncontainable error (UC).
0b11	Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the IFSC field is not 0b010000.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

Value	Meaning
0b0	FAR is valid.
0b1	FAR is not valid, and holds an UNKNOWN value.

This field is only valid if the IFSC code is 0b010000. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

Bit [8]

Reserved, RES0.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Value	Meaning
0b0	Fault not on a stage 2 translation for a stage 1 translation table walk.
0b1	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

Bit [6]

Reserved, RES0.

IFSC, bits [5:0]

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

Value	Meaning
0b000000	Address size fault, level 0 of translation or translation table base register
0b000001	Address size fault, level 1
0b000010	Address size fault, level 2
0b000011	Address size fault, level 3

Value	Meaning
`0b000100`	Translation fault, level 0
`0b000101`	Translation fault, level 1
`0b000110`	Translation fault, level 2
`0b000111`	Translation fault, level 3
`0b001001`	Access flag fault, level 1
`0b001010`	Access flag fault, level 2
`0b001011`	Access flag fault, level 3
`0b001101`	Permission fault, level 1
`0b001110`	Permission fault, level 2
`0b001111`	Permission fault, level 3
`0b010000`	Synchronous External abort, not on translation table walk
`0b010100`	Synchronous External abort, on translation table walk, level 0
`0b010101`	Synchronous External abort, on translation table walk, level 1
`0b010110`	Synchronous External abort, on translation table walk, level 2
`0b010111`	Synchronous External abort, on translation table walk, level 3
`0b011000`	Synchronous parity or ECC error on memory access, not on translation table walk
`0b011100`	Synchronous parity or ECC error on memory access on translation table walk, level 0
`0b011101`	Synchronous parity or ECC error on memory access on translation table walk, level 1
`0b011110`	Synchronous parity or ECC error on memory access on translation table walk, level 2
`0b011111`	Synchronous parity or ECC error on memory access on translation table walk, level 3
`0b101000`	Capability tag fault.
`0b101001`	Capability sealed fault.
`0b101010`	Capability bound fault.
`0b101011`	Capability permission fault.
`0b110000`	TLB conflict abort
`0b110001`	Unsupported atomic hardware update fault, if the implementation includes xARMv8.1-TTHM. Otherwise reserved.

All other values are reserved.

When the RAS Extension is implemented, 0b011000, 0b011100, 0b011101, 0b011110, and 0b011111, are reserved.

Armv8.2 requires the implementation of the RAS Extension.

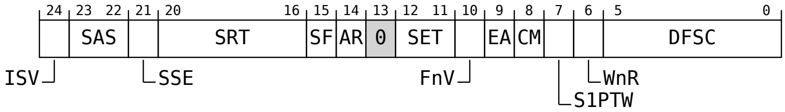
For more information about the lookup level associated with a fault, see x'The level associated with MMU faults' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

an exception from a Data Abort



ISV, bit [24]

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

Value	Meaning
`0b0`	No valid instruction syndrome. ISS[23:14] are RES0.
`0b1`	ISS[23:14] hold a valid instruction syndrome.

This bit is 0 for all faults reported in ESR_EL2 except the following stage 2 aborts:

- AArch64 loads and stores of a single general-purpose register (including the register specified with 0b11111, including those with Acquire/Release semantics, but excluding Load Exclusive or Store Exclusive, excluding those with writeback and excluding accesses of a capability.
- AArch32 instructions where the instruction:
  - Is an LDR, LDA, LDRT, LDRSH, LDRSHT, LDRH, LDAH, LDRHT, LDRSB, LDRSBT, LDRB, LDAB, LDRBT, STR, STL, STRT, STRH, STLH, STRHT, STRB, STLB, or STRBT instruction.
  - Is not performing register writeback.
  - Is not using R15 as a source or destination register.

For these cases, ISV is UNKNOWN if the exception was generated in Debug state in memory access mode, and otherwise indicates whether ISS[23:14] hold a valid syndrome.

ISV is 0 for all faults reported in ESR_EL1 or ESR_EL3.

When the RAS Extension is implemented, ISV is 0 for any synchronous External abort.

For ISS reporting, a stage 2 abort on a stage 1 translation table walk does not return a valid instruction syndrome, and therefore ISV is 0 for these aborts.

When the RAS Extension is not implemented, the value of ISV on a synchronous External abort on a stage 2 translation table walk is IMPLEMENTATION DEFINED.

This field resets to an architecturally UNKNOWN value.

SAS, bits [23:22]

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

Value	Meaning
`0b00`	Byte
`0b01`	Halfword

Value	Meaning
`0b10`	Word
`0b11`	Doubleword

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SSE, bit [21]

Syndrome Sign Extend. When ISV is 1, for a byte, halfword, or word load operation, indicates whether the data item must be sign extended. For these cases, the possible values of this bit are:

Value	Meaning
`0b0`	Sign-extension not required.
`0b1`	Data item must be sign-extended.

For all other operations this bit is 0.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SRT, bits [20:16]

Syndrome Register transfer. When ISV is 1, the register number of the Rt operand of the faulting instruction. If the exception was taken from an Exception level that is using AArch32 then this is the AArch64 view of the register. See x'Mapping of the general-purpose registers between the Execution states' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

SF, bit [15]

Width of the register accessed by the instruction is Sixty-Four. When ISV is 1, the possible values of this bit are:

Value	Meaning
`0b0`	Instruction loads/stores a 32-bit wide register.
`0b1`	Instruction loads/stores a 64-bit wide register.

This field specifies the register width identified by the instruction, not the Execution state.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

AR, bit [14]

Acquire/Release. When ISV is 1, the possible values of this bit are:

Value	Meaning
`0b0`	Instruction did not have acquire/release semantics.
`0b1`	Instruction did have acquire/release semantics.

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

This field is RES0 when the value of ISV is 0.

This field resets to an architecturally UNKNOWN value.

Bit [13]

Reserved, RES0.

SET, bits [12:11]

Synchronous Error Type. When the RAS Extension is implemented and DFSC is 0b010000, describes the state of the PE after taking the Data Abort exception. The possible values of this field are:

Value	Meaning
`0b00`	Recoverable error (UER).
`0b10`	Uncontainable error (UC).
`0b11`	Restartable error (UEO) or Corrected error (CE).

All other values are reserved.

Software can use this information to determine what recovery might be possible. Taking a synchronous External Abort exception might result in an unrecoverable PE state.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010000.

This field resets to an architecturally UNKNOWN value.

FnV, bit [10]

FAR not Valid, for a synchronous External abort other than a synchronous External abort on a translation table walk.

Value	Meaning
`0b0`	FAR is valid.
`0b1`	FAR is not valid, and holds an UNKNOWN value.

This field is valid only if the DFSC code is 0b010000. It is RES0 for all other aborts.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. This bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field resets to an architecturally UNKNOWN value.

CM, bit [8]

Cache maintenance. Indicates whether the Data Abort came from a cache maintenance or address translation instruction:

Value	Meaning
`0b0`	The Data Abort was not generated by the execution of one of the System instructions identified in the description of value 1.
`0b1`	The Data Abort was generated by either the execution of a cache maintenance instruction or by a synchronous fault on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

S1PTW, bit [7]

For a stage 2 fault, indicates whether the fault was a stage 2 fault on an access made for a stage 1 translation table walk:

Value	Meaning
`0b0`	Fault not on a stage 2 translation for a stage 1 translation table walk.
`0b1`	Fault on the stage 2 translation of an access for a stage 1 translation table walk.

For any abort other than a stage 2 fault this bit is RES0.

This field resets to an architecturally UNKNOWN value.

WnR, bit [6]

Write not Read. Indicates whether a synchronous abort was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

Value	Meaning
`0b0`	Abort caused by an instruction reading from a memory location.
`0b1`	Abort caused by an instruction writing to a memory location.

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

For faults from an atomic instruction that both reads and writes from a memory location, this bit is set to 0 if a read of the address specified by the instruction would have generated the fault which is being reported, otherwise it is set to 1. The architecture permits, but does not require, a relaxation of this requirement such that for all stage 2 aborts on stage 1 translation table walks for atomic instructions, the WnR bit is always 0.

For Page table LC or SC permission violation faults from an atomic instruction that both reads and writes a valid capability from a memory location, this bit is set to 1 if a write of a valid capability from the memory location would have generated the fault which is being reported, otherwise it is set to 0.

This field is UNKNOWN for:

- An External abort on an Atomic access.
- A fault reported using a DFSC value of 0b110101 or 0b110001, indicating an unsupported Exclusive or atomic access.

This field resets to an architecturally UNKNOWN value.

DFSC, bits [5:0]

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

Value	Meaning
`0b000000`	Address size fault, level 0 of translation or translation table base register.
`0b000001`	Address size fault, level 1.
`0b000010`	Address size fault, level 2.
`0b000011`	Address size fault, level 3.
`0b000100`	Translation fault, level 0.
`0b000101`	Translation fault, level 1.
`0b000110`	Translation fault, level 2.
`0b000111`	Translation fault, level 3.
`0b001001`	Access flag fault, level 1.
`0b001010`	Access flag fault, level 2.
`0b001011`	Access flag fault, level 3.
`0b001101`	Permission fault, level 1.
`0b001110`	Permission fault, level 2.
`0b001111`	Permission fault, level 3.
`0b010000`	Synchronous External abort, not on translation table walk.
`0b010001`	Synchronous Tag Check fail
`0b010100`	Synchronous External abort, on translation table walk, level 0.
`0b010101`	Synchronous External abort, on translation table walk, level 1.
`0b010110`	Synchronous External abort, on translation table walk, level 2.
`0b010111`	Synchronous External abort, on translation table walk, level 3.
`0b011000`	Synchronous parity or ECC error on memory access, not on translation table walk.
`0b011100`	Synchronous parity or ECC error on memory access on translation table walk, level 0.

Value	Meaning
`0b011101`	Synchronous parity or ECC error on memory access on translation table walk, level 1.
`0b011110`	Synchronous parity or ECC error on memory access on translation table walk, level 2.
`0b011111`	Synchronous parity or ECC error on memory access on translation table walk, level 3.
`0b100001`	Alignment fault.
`0b101000`	Capability tag fault.
`0b101001`	Capability sealed fault.
`0b101010`	Capability bound fault.
`0b101011`	Capability permission fault.
`0b101100`	Page table LC or SC permission violation fault.
`0b110000`	TLB conflict abort.
`0b110001`	Unsupported atomic hardware update fault, if the implementation includes xARMv8.1-TTHM. Otherwise reserved.
`0b110100`	IMPLEMENTATION DEFINED fault (Lockdown).
`0b110101`	IMPLEMENTATION DEFINED fault (Unsupported Exclusive or Atomic access).
`0b110110`	Unsupported LDCT or SDCT to Device or Non-cacheable.
`0b111101`	Section Domain Fault, used only for faults reported in the PAR_EL1.
`0b111110`	Page Domain Fault, used only for faults reported in the PAR_EL1.

All other values are reserved.

When the RAS Extension is implemented, 0b011000, 0b011100, 0b011101, 0b011110, and 0b011111, are reserved.

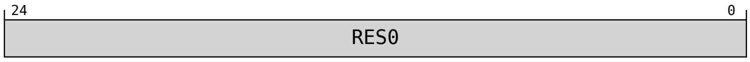
For more information about the lookup level associated with a fault, see x'The level associated with MMU faults' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Because Access flag faults and Permission faults can only result from a Block or Page translation table descriptor, they cannot occur at level 0.

If the S1PTW bit is set, then the level refers the level of the stage2 translation that is translating a stage 1 translation walk.

This field resets to an architecturally UNKNOWN value.

an exception from an access to the Morello architecture

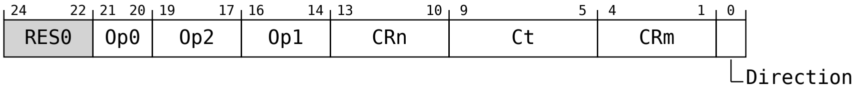


Bits [24:0]

Reserved, RES0.

In an implementation that supports Morello architecture, from an Exception level using AArch64, the CPACR_EL1.CEN, CPTR_EL2.{CEN, DC} and CPTR_EL3.EC bits control whether Morello instructions and accesses to Morello System registers are trapped.

an exception from capability MSR or MRS instruction execution



Bits [24:22]

Reserved, RES0.

Op0, bits [21:20]

The Op0 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op2, bits [19:17]

The Op2 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Op1, bits [16:14]

The Op1 value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

CRn, bits [13:10]

The CRn value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

Ct, bits [9:5]

The Ct value from the issued instruction, the capability register used for the transfer.

This field resets to an architecturally UNKNOWN value.

CRm, bits [4:1]

The CRm value from the issued instruction.

This field resets to an architecturally UNKNOWN value.

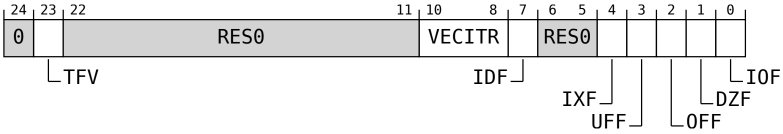
Direction, bit [0]

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

Value	Meaning
0b0	Write access, including MSR instructions.
0b1	Read access, including MRS instructions.

This field resets to an architecturally UNKNOWN value.

an exception from a trapped floating-point exception



Bit [24]

Reserved, RES0.

TFV, bit [23]

Trapped Fault Valid bit. Indicates whether the IDF, IXF, UFF, OFF, DZF, and IOF bits hold valid information about trapped floating-point exceptions. The possible values of this bit are:

Value	Meaning
`0b0`	The IDF, IXF, UFF, OFF, DZF, and IOF bits do not hold valid information about trapped floating-point exceptions and are UNKNOWN.
`0b1`	One or more floating-point exceptions occurred during an operation performed while executing the reported instruction. The IDF, IXF, UFF, OFF, DZF, and IOF bits indicate trapped floating-point exceptions that occurred. For more information see x'Floating- point exceptions and exception traps'.

It is IMPLEMENTATION DEFINED whether this field is set to 0 on an exception generated by a trapped floating point exception from a vector instruction.

This is not a requirement. Implementations can set this field to 1 on a trapped floating-point exception from a vector instruction and return valid information in the {IDF, IXF, UFF, OFF, DZF, IOF} fields.

This field resets to an architecturally UNKNOWN value.

Bits [22:11]

Reserved, RES0.

VECITR, bits [10:8]

For a trapped floating-point exception from an instruction executed in AArch32 state this field is RES1.

For a trapped floating-point exception from an instruction executed in AArch64 state this field is UNKNOWN.

This field resets to an architecturally UNKNOWN value.

IDF, bit [7]

Input Denormal floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Input denormal floating-point exception has not occurred.
`0b1`	Input denormal floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

Bits [6:5]

Reserved, RES0.

IXF, bit [4]

Inexact floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Inexact floating-point exception has not occurred.
`0b1`	Inexact floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

UFF, bit [3]

Underflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Underflow floating-point exception has not occurred.
`0b1`	Underflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

OFF, bit [2]

Overflow floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Overflow floating-point exception has not occurred.
`0b1`	Overflow floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

DZF, bit [1]

Divide by Zero floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

Value	Meaning
`0b0`	Divide by Zero floating-point exception has not occurred.

Value	Meaning
0b1	Divide by Zero floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

IOF, bit [0]

Invalid Operation floating-point exception trapped bit. If the TFV field is 0, this bit is UNKNOWN. Otherwise, the possible values of this bit are:

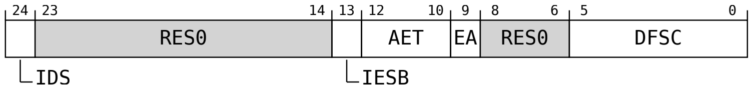
Value	Meaning
0b0	Invalid Operation floating-point exception has not occurred.
0b1	Invalid Operation floating-point exception occurred during execution of the reported instruction.

This field resets to an architecturally UNKNOWN value.

In an implementation that supports the trapping of floating-point exceptions:

- From an Exception level using AArch64, the FPCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.
- From an Exception level using AArch32, the FPSCR.{IDE, IXE, UFE, OFE, DZE, IOE} bits enable each of the floating-point exception traps.

an SError interrupt



IDS, bit [24]

IMPLEMENTATION DEFINED syndrome. Possible values of this bit are:

Value	Meaning
0b0	Bits[23:0] of the ISS field holds the fields described in this encoding. If the RAS Extension is not implemented, this means that bits[23:0] of the ISS field are RES0.
0b1	Bits[23:0] of the ISS field holds IMPLEMENTATION DEFINED syndrome information that can be used to provide additional information about the SError interrupt.

This field was previously called ISV.

This field resets to an architecturally UNKNOWN value.

Bits [23:14]

Reserved, RES0.

IESB, bit [13]

When ARMv8.2-IESB is implemented:

Implicit error synchronization event.

Value	Meaning
0b0	The SError interrupt was either not synchronized by the implicit error synchronization event or not taken immediately.
0b1	The SError interrupt was synchronized by the implicit error synchronization event and taken immediately.

This field is RES0 if the value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension and xARMv8.2-IESB.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

AET, bits [12:10]

Asynchronous Error Type.

When the RAS Extension is implemented and DFSC is 0b010001, describes the state of the PE after taking the SError interrupt exception. The possible values of this field are:

Value	Meaning
0b000	Uncontainable error (UC).
0b001	Unrecoverable error (UEU).
0b010	Restartable error (UEO).
0b011	Recoverable error (UER).
0b110	Corrected error (CE).

All other values are reserved.

If multiple errors are taken as a single SError interrupt exception, the overall state of the PE is reported. For example, if both a Recoverable and Unrecoverable error occurred, the state is Unrecoverable.

Software can use this information to determine what recovery might be possible. The recovery software must also examine any implemented fault records to determine the location and extent of the error.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

EA, bit [9]

External abort type. When the RAS Extension is implemented, this bit can provide an IMPLEMENTATION DEFINED classification of External aborts.

For any abort other than an External abort this bit returns a value of 0.

This field is RES0 if either:

- The RAS Extension is not implemented.
- The value returned in the DFSC field is not 0b010001.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

Bits [8:6]

Reserved, RES0.

DFSC, bits [5:0]

Data Fault Status Code. When the RAS Extension is implemented, possible values of this field are:

Value	Meaning
0b000000	Uncategorized.
0b010001	Asynchronous SError interrupt.

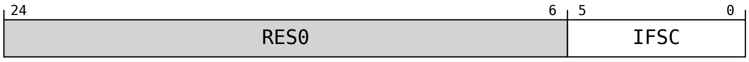
All other values are reserved.

If the RAS Extension is not implemented, this field is RES0.

Armv8.2 requires the implementation of the RAS Extension.

This field resets to an architecturally UNKNOWN value.

an exception from a Breakpoint or Vector Catch debug exception



Bits [24:6]

Reserved, RES0.

IFSC, bits [5:0]

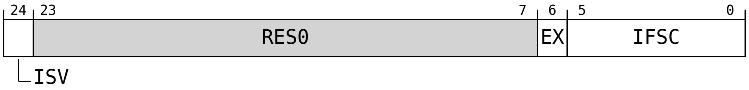
Instruction Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions:

- For exceptions from AArch64, see x‘Breakpoint exceptions’.
- For exceptions from AArch32, see x‘Breakpoint exceptions’ and x‘Vector Catch exceptions’.

an exception from a Software Step exception



ISV, bit [24]

Instruction syndrome valid. Indicates whether the EX bit, ISS[6], is valid, as follows:

Value	Meaning
`0b0`	EX bit is RES0.
`0b1`	EX bit is valid.

See the EX bit description for more information.

This field resets to an architecturally UNKNOWN value.

Bits [23:7]

Reserved, RES0.

EX, bit [6]

Exclusive operation. If the ISV bit is set to 1, this bit indicates whether a Load-Exclusive instruction was stepped.

Value	Meaning
`0b0`	An instruction other than a Load- Exclusive instruction was stepped.
`0b1`	A Load-Exclusive instruction was stepped.

If the ISV bit is set to 0, this bit is RES0, indicating no syndrome data is available.

This field resets to an architecturally UNKNOWN value.

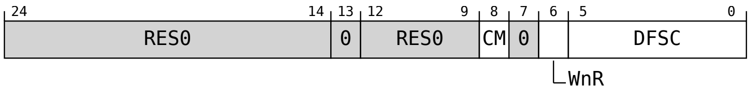
IFSC, bits [5:0]

Instruction Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x‘Software Step exceptions’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile,.

an exception from a Watchpoint exception



Bits [24:14]

Reserved, RES0.

Bit [13]

Reserved, RES0.

Bits [12:9]

Reserved, RES0.

CM, bit [8]

Cache maintenance. Indicates whether the Watchpoint exception came from a cache maintenance or address translation instruction:

Value	Meaning
`0b0`	The Watchpoint exception was not generated by the execution of one of the System instructions identified in the description of value 1.
`0b1`	The Watchpoint exception was generated by either the execution of a cache maintenance instruction or by a synchronous Watchpoint exception on the execution of an address translation instruction. The DC ZVA instruction is not classified as a cache maintenance instruction, and therefore its execution cannot cause this field to be set to 1.

This field resets to an architecturally UNKNOWN value.

Bit [7]

Reserved, RES0.

WnR, bit [6]

Write not Read. Indicates whether the Watchpoint exception was caused by an instruction writing to a memory location, or by an instruction reading from a memory location. The possible values of this bit are:

Value	Meaning
`0b0`	Watchpoint exception caused by an instruction reading from a memory location.
`0b1`	Watchpoint exception caused by an instruction writing to a memory location.

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

For Watchpoint exceptions from an atomic instruction, this field is set to 0 if a read of the location would have generated the Watchpoint exception, otherwise it is set to 1.

If multiple watchpoints match on the same access, it is UNPREDICTABLE which watchpoint generates the Watchpoint exception.

This field resets to an architecturally UNKNOWN value.

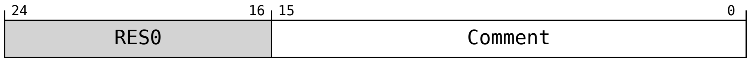
DFSC, bits [5:0]

Data Fault Status Code. This field is set to 0b100010, to indicate a Debug exception.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x‘Watchpoint exceptions’ in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from execution of a Breakpoint instruction



Bits [24:16]

Reserved, RES0.

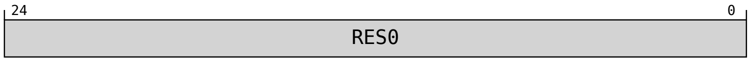
Comment, bits [15:0]

Set to the instruction comment field value, zero extended as necessary. For the AArch32 BKPT instructions, the comment field is described as the immediate field.

This field resets to an architecturally UNKNOWN value.

For more information about generating these exceptions, see x'Breakpoint instruction exceptions' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

an exception from a Pointer Authentication instruction when HCR_EL2.API == 0 || SCR_EL3.API == 0



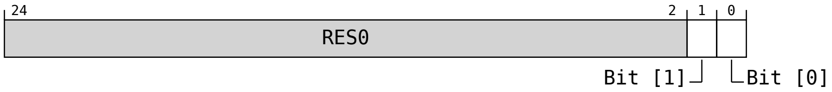
Bits [24:0]

Reserved, RES0.

For more information about generating these exceptions, see:

- HCR_EL2.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL2.
- SCR_EL3.API, for exceptions from Pointer authentication instructions, using AArch64 state, trapped to EL3.

an exception from a Pointer Authentication instruction authentication failure



Bits [24:2]

Reserved, RES0.

Bit [1], bit [1]

This field indicates whether the exception is as a result of an Instruction key or a Data key.

Value	Meaning
0b0	Instruction Key.
0b1	Data Key.

This field resets to an architecturally UNKNOWN value.

Bit [0], bit [0]

This field indicates whether the exception is as a result of an A key or a B key.

Value	Meaning
0b0	A key.
0b1	B key.

This field resets to an architecturally UNKNOWN value.

The following instructions generate an exception when the Pointer Authentication Code (PAC) is incorrect:

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

It is IMPLEMENTATION DEFINED whether the following instructions generate an exception directly from the authorization failure, rather than changing the address in a way that will generate a translation fault when the address is accessed:

- RETAA, RETAB.
- BRAA, BRAB, BLRAA, BLRAB.
- BRAAZ, BRABZ, BLRAAZ, BLRABZ.
- ERETAA, ERETAB.
- LDRAA, LDRAB, whether the authenticated address is written back to the base register or not.

Accessing the ESR_EL3

Read using name ESR_EL3

The assembler syntax is:

```
MRS <Xt>, ESR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ESR_EL3;
```

Write using name ESR_EL3

The assembler syntax is:

```
MSR ESR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0101	0b0010	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        ESR_EL3 = X[t];
```

3.2.28 FAR_EL1, Fault Address Register (EL1)

The FAR_EL1 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction or Data Abort, PC alignment fault and Watchpoint exceptions that are taken to EL1.

Attributes

FAR_EL1 is a 64-bit register.

Configuration

AArch64 System register FAR_EL1[31:0] is architecturally mapped to AArch32 System register DFAR[31:0] (NS).

AArch64 System register FAR_EL1[63:32] is architecturally mapped to AArch32 System register IFAR[31:0] (NS).

Field descriptions

The FAR_EL1 bit assignments are:

63	32
Faulting Virtual Address for synchronous exceptions taken to EL1	

31	0
Faulting Virtual Address for synchronous exceptions taken to EL1	

Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL1. Exceptions that set the FAR_EL1 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x35). ESR_EL1.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL1 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL1.FnV is 0, and the FAR_EL1 is UNKNOWN if ESR_EL1.FnV is 1.

For all other exceptions taken to EL1, the FAR_EL1 is UNKNOWN.

If a memory fault that sets FAR_EL1 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL1 is taken from an Exception level that is using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store is CONSTRAINED UNPREDICTABLE. See 'Out of range VA' in Appendix K1 Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging in AArch64 state' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Execution at EL0 makes FAR_EL1 become UNKNOWN.

If the Morello architecture is implemented, this field holds the address with any capability memory relocation applied. If the memory fault is generated from a data cache maintenance or other DC instruction, this field holds the address supplied in the register argument of the instruction with any capability memory relocation applied.

If the Morello architecture is implemented, for capability faults due to instruction performing multiple data accesses, such as load or store of pairs, this field holds the faulting address. The faulting address is the lowest address accessed by one of the data accesses. It is IMPLEMENTATION DEFINED which data access is selected to provide the faulting address.

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL1 is made UNKNOWN on an exception return from EL1.

This field resets to an architecturally UNKNOWN value.

Accessing the FAR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic FAR_EL1 or FAR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name FAR_EL1

The assembler syntax is:

```
MRS <Xt>, FAR_EL1
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return FAR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return FAR_EL2;
    else
        return FAR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return FAR_EL1;
```

Write using name FAR_EL1

The assembler syntax is:

```
MSR FAR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        FAR_EL2 = X[t];
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        FAR_EL1 = X[t];
```

Read using name FAR_EL12

The assembler syntax is:

```
MRS <Xt>, FAR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

Accessibility:

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

```
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return FAR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return FAR_EL1;
    else
        UNDEFINED;
```

Write using name FAR_EL12

The assembler syntax is:

```
MSR FAR_EL12, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0110	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            FAR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            FAR_EL1 = X[t];
    else
        UNDEFINED;
```

3.2.29 FAR_EL2, Fault Address Register (EL2)

The FAR_EL2 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction or Data Abort, PC alignment fault and Watchpoint exceptions that are taken to EL2.

Attributes

FAR_EL2 is a 64-bit register.

Configuration

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

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

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register HDFAR[31:0].

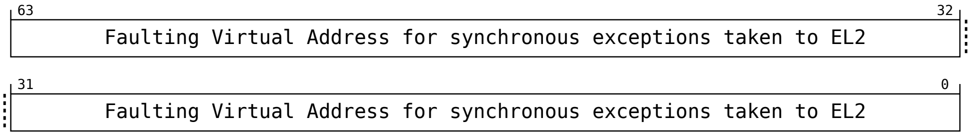
AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register HIFAR[31:0].

AArch64 System register FAR_EL2[31:0] is architecturally mapped to AArch32 System register DFAR[31:0] (S)when HaveEL(EL2).

AArch64 System register FAR_EL2[63:32] is architecturally mapped to AArch32 System register IFAR[31:0] (S)when HaveEL(EL2).

Field descriptions

The FAR_EL2 bit assignments are:


63
32
Faulting Virtual Address for synchronous exceptions taken to EL2
31
0
Faulting Virtual Address for synchronous exceptions taken to EL2

Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL2. Exceptions that set the FAR_EL2 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), PC alignment faults (EC 0x22), and Watchpoints (EC 0x34 or 0x35). ESR_EL2.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL2 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL2.FnV is 0, and the FAR_EL2 is UNKNOWN if ESR_EL2.FnV is 1.

For all other exceptions taken to EL2, the FAR_EL2 is UNKNOWN.

If a memory fault that sets FAR_EL2 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL2 is taken from an Exception level that is using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE. See 'Out of range VA' in Appendix K1 Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging in AArch64 state' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Execution at EL1 or EL0 makes FAR_EL2 become UNKNOWN.

If the Morello architecture is implemented, this field holds the address with any capability memory relocation applied. If the memory fault is generated from a data cache maintenance or other DC instruction, this field holds the address supplied in the register argument of the instruction with any capability memory relocation applied.

If the Morello architecture is implemented, for capability faults due to instruction performing multiple data accesses, such as load or store of pairs, this field holds the faulting address. The faulting address is the lowest address accessed by one of the data accesses. It is IMPLEMENTATION DEFINED which data access is selected to provide the faulting address.

The address held in this field is an address accessed by the instruction fetch or data access that caused the exception that gave rise to the instruction or data abort. It is the lower address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL2 is made UNKNOWN on an exception return from EL2.

This field resets to an architecturally UNKNOWN value.

Accessing the FAR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic FAR_EL2 or FAR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name FAR_EL2

The assembler syntax is:

```
MRS <Xt>, FAR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return FAR_EL2;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return FAR_EL2;
```

Write using name FAR_EL2

The assembler syntax is:

```
MSR FAR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0110	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        FAR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        FAR_EL2 = X[t];
```

Read using name FAR_EL1

The assembler syntax is:

```
MRS <Xt>, FAR_EL1
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TRVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return FAR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
```

```
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return FAR_EL2;
    else
        return FAR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return FAR_EL1;
```

Write using name FAR_EL1

The assembler syntax is:

```
MSR FAR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

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

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TVM == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        FAR_EL2 = X[t];
    else
        FAR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        FAR_EL1 = X[t];
```

3.2.30 FAR_EL3, Fault Address Register (EL3)

The FAR_EL3 characteristics are:

Purpose

Holds the faulting Virtual Address for all synchronous Instruction or Data Abort and PC alignment fault exceptions that are taken to EL3.

Attributes

FAR_EL3 is a 64-bit register.

Configuration

This register is present only when HaveEL(EL3). Otherwise, direct accesses to FAR_EL3 are UNDEFINED.

Field descriptions

The FAR_EL3 bit assignments are:

63	32
Faulting Virtual Address for synchronous exceptions taken to EL3	

31	0
Faulting Virtual Address for synchronous exceptions taken to EL3	

Bits [63:0]

Faulting Virtual Address for synchronous exceptions taken to EL3. Exceptions that set the FAR_EL3 are Instruction Aborts (EC 0x20 or 0x21), Data Aborts (EC 0x24 or 0x25), and PC alignment faults (EC 0x22). ESR_EL3.EC holds the EC value for the exception.

For a synchronous External abort, if the VA that generated the abort was from an address range for which TCR_ELx.TBI{<0|1>} == 1 for the translation regime in use when the abort was generated, then the top eight bits of FAR_EL3 are UNKNOWN.

For a synchronous External abort other than a synchronous External abort on a translation table walk, this field is valid only if ESR_EL3.FnV is 0, and the FAR_EL3 is UNKNOWN if ESR_EL3.FnV is 1.

For all other exceptions taken to EL3, the FAR_EL3 is UNKNOWN.

If a memory fault that sets FAR_EL3 is generated from a data cache maintenance or other DC instruction, this field holds the address specified in the register argument of the instruction.

If the exception that updates FAR_EL3 is taken from an Exception Level using AArch32, the top 32 bits are all zero, unless both of the following apply, in which case the top 32 bits of FAR_ELx are 0x00000001:

- The faulting address was generated by a load or store instruction that sequentially incremented from address 0xFFFFFFFF. Such a load or store instruction is CONSTRAINED UNPREDICTABLE. See 'Out of range VA' in Appendix K1 Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.
- The implementation treats such incrementing as setting bit[32] of the virtual address to 1.

For a Data Abort or Watchpoint exception, if address tagging is enabled for the address accessed by the data access that caused the exception, then this field includes the tag. For more information about address tagging, see 'Address tagging in AArch64 state' in the Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile.

Execution at EL2, EL1 or EL0 makes FAR_EL3 become UNKNOWN.

If the Morello architecture is implemented, this field holds the address with any capability memory relocation applied. If the memory fault is generated from a data cache maintenance or other DC instruction, this field holds the address supplied in the register argument of the instruction with any capability memory relocation applied.

If the Morello architecture is implemented, for capability faults due to instruction performing multiple data accesses, such as load or store of pairs, this field holds the faulting address. The faulting address is the lowest address accessed by one of the data accesses. It is IMPLEMENTATION DEFINED which data access is selected to provide the faulting address.

The address held in this register is an address accessed by the instruction fetch or data access that caused the exception that actually gave rise to the instruction or data abort. It is the lowest address that gave rise to the fault. Where different faults from different addresses arise from the same instruction, such as for an instruction that loads or stores a mis-aligned address that crosses a page boundary, the architecture does not prioritize between those different faults.

FAR_EL3 is made UNKNOWN on an exception return from EL3.

This field resets to an architecturally UNKNOWN value.

Accessing the FAR_EL3

Read using name FAR_EL3

The assembler syntax is:

```
MRS <Xt>, FAR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return FAR_EL3;
```

Write using name FAR_EL3

The assembler syntax is:

```
MSR FAR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0110	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
```

```
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        FAR_EL3 = X[t];
```

3.2.31 ID_AA64PFR1_EL1, AArch64 Processor Feature Register 1

The ID_AA64PFR1_EL1 characteristics are:

Purpose

Reserved for future expansion of information about implemented PE features in AArch64 state.

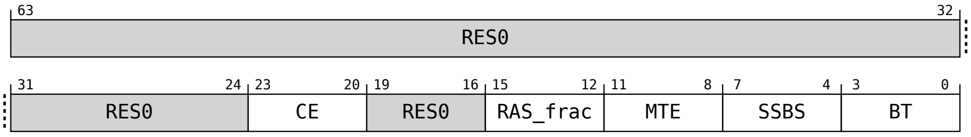
For general information about the interpretation of the ID registers, see x‘Principles of the ID scheme for fields in ID registers’.

Attributes

ID_AA64PFR1_EL1 is a 64-bit register.

Field descriptions

The ID_AA64PFR1_EL1 bit assignments are:



Bits [63:24]

Reserved, RES0.

CE, bits [23:20]

When Morello is implemented:

Morello architecture.

Value	Meaning
`0b0000`	Morello architecture is not implemented.
`0b0001`	Morello architecture is implemented.

All other values are reserved.

Otherwise:

RES0

Bits [19:16]

Reserved, RES0.

RAS_frac, bits [15:12]

From ARMv8.4:

RAS Extension fractional field.

Value	Meaning
`0b0000`	If ID_AA64PFR0_EL1.RAS == 0b0001, RAS Extension implemented.

Value	Meaning
`0b0001`	If ID_AA64PFR0_EL1.RAS == 0b0001, as 0b0000 and adds support for: • Additional ERXMISC<m>_EL1 System registers. • Additional System registers ERXPFGCDN_EL1, ERXPFGCTL_EL1, and ERXPFGF_EL1, and the SCR_EL3.FIEN and HCR_EL2.FIEN trap controls, to support the optional RAS Common Fault Injection Model Extension. Error records accessed through System registers conform to RAS System Architecture v1.1, which includes simplifications to ext-ERR<n>STATUS, and support for the optional RAS Timestamp and RAS Common Fault Injection Model Extensions.

All other values are reserved.

This field is valid only if ID_AA64PFR0_EL1.RAS == 0b0001.

Otherwise:

RES0

MTE, bits [11:8]

From ARMv8.5:

Support for the Memory Tagging Extension.

Value	Meaning
`0b0000`	Memory Tagging Extension is not implemented.
`0b0001`	Memory Tagging Extension instructions accessible at EL0 are implemented. Instructions and System Registers defined by the extension not configurably accessible at EL0 are Unallocated and other System Register fields defined by the extension are RES0.
`0b0010`	Memory Tagging Extension is implemented.

All other values are reserved.

xARMv8.5-MemTag implements the functionality identified by the value 0b0001.

When ID_AA64PFR1_EL1.MTE != 0b0010:

- All register fields added to existing System registers and Special-purpose registers as part of the extension are RES0, and treated as 0.
- The following System registers are UNDEFINED:
  - GMID_EL1, GCR_EL1, RGSR_EL1, TFSRE0_EL1, and TFSR_ELx.
- The following System instructions are UNDEFINED:
  - DC CGSW, DC CIGSW, DC IGSW, DC CGDSW, DC CIGDSW, DC IGDSW, DC IGVAC, and DC IGDVAC.
- The following instructions are UNDEFINED:
  - LDGM, STGM, and STZGM.

- The Tagged memory type encoding in MAIR_ELx is UNPREDICTABLE.

Otherwise:

RES0

SSBS, bits [7:4]

From ARMv8.5:

Speculative Store Bypassing controls in AArch64 state. Defined values are:

Value	Meaning
`0b0000`	AArch64 provides no mechanism to control the use of Speculative Store Bypassing.
`0b0001`	AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypass Safe.
`0b0010`	AArch64 provides the PSTATE.SSBS mechanism to mark regions that are Speculative Store Bypassing Safe, and the MSR and MRS instructions to directly read and write the PSTATE.SSBS field

All other values are reserved.

Otherwise:

RES0

BT, bits [3:0]

From ARMv8.5:

Branch Target Identification mechanism support in AArch64 state. Defined values are:

Value	Meaning
`0b0000`	The Branch Target Identification mechanism is not implemented.
`0b0001`	The Branch Target Identification mechanism is implemented.

All other values are reserved.

xARMv8.5-BTI implements the functionality identified by the value 0b0001.

From Armv8.5, the only permitted value is 0b0001.

Otherwise:

RES0

Accessing the ID_AA64PFR1_EL1

Read using name ID_AA64PFR1_EL1

The assembler syntax is:

```
MRS <Xt>, ID_AA64PFR1_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0000	0b0100	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TID3 == '1' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    else
        return ID_AA64PFR1_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ID_AA64PFR1_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return ID_AA64PFR1_EL1;
```

3.2.32 PMBSR_EL1, Profiling Buffer Status/syndrome Register

The PMBSR_EL1 characteristics are:

Purpose

Provides syndrome information to software when the buffer is disabled because the management interrupt has been raised.

Attributes

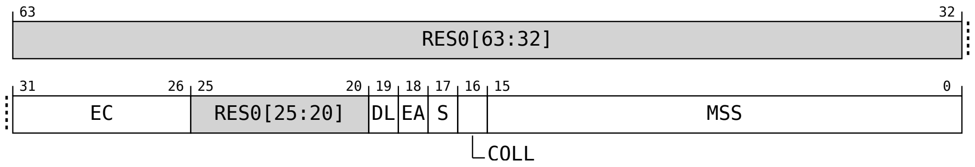
PMBSR_EL1 is a 64-bit register.

Configuration

This register is present only when SPE is implemented. Otherwise, direct accesses to PMBSR_EL1 are UNDEFINED.

Field descriptions

The PMBSR_EL1 bit assignments are:



Bits [63:32, 25:20]

Reserved, RES0.

EC, bits [31:26]

Exception class

Top-level description of the cause of the buffer management event

Value	Meaning	Link
`0b000000`	Other buffer management event. All buffer management events other than those described by other defined Exception class codes.	MSS - other buffer management events
`0b100100`	Stage 1 Data Abort on write to Profiling Buffer.	MSS - stage 1 or stage 2 Data Aborts on write to buffer
`0b100101`	Stage 2 Data Abort on write to Profiling Buffer.	MSS - stage 1 or stage 2 Data Aborts on write to buffer

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Writing a reserved value to this field will make the value of this field UNKNOWN. Values that are not supported act as reserved values when writing to this register.

On a warm reset, this field resets to an architecturally UNKNOWN value.

DL, bit [19]

Partial record lost.

Following a buffer management event other than an asynchronous External abort, indicates whether the last record written to the Profiling Buffer is complete.

Value	Meaning
0b0	PMBPTR_EL1 points to the first byte after the last complete record written to the Profiling Buffer.
0b1	Part of a record was lost because of a buffer management event or synchronous External abort. PMBPTR_EL1 might not point to the first byte after the last complete record written to the buffer, and so restarting collection might result in a data record stream that software cannot parse. All records prior to the last record have been written to the buffer.

When the buffer management event was because of an asynchronous external abort, this bit is set to 1 and software must not assume that any valid data has been written to the Profiling Buffer.

This bit is RES0 if the PE never sets this bit as a result of a buffer management event caused by an asynchronous External abort.

On a warm reset, this field resets to an architecturally UNKNOWN value.

EA, bit [18]

External abort.

Value	Meaning
0b0	An external abort has not been asserted.
0b1	An external abort has been asserted and detected by the Statistical Profiling Extension.

This bit is RES0 if the PE never sets this bit as the result of an External abort.

On a warm reset, this field resets to an architecturally UNKNOWN value.

S, bit [17]

Service

Value	Meaning
0b0	PMBIRQ is not asserted.
0b1	PMBIRQ is asserted. All profiling data has either been written to the buffer or discarded.

On a warm reset, this field resets to an architecturally UNKNOWN value.

COLL, bit [16]

Collision detected.

Value	Meaning
0b0	No collision events detected.
0b1	At least one collision event was recorded.

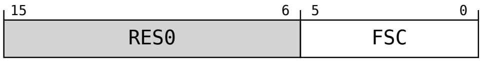
On a warm reset, this field resets to an architecturally UNKNOWN value.

MSS, bits [15:0]

Management Event Specific Syndrome.

Contains syndrome specific to the management event.

stage 1 or stage 2 Data Aborts on write to buffer



Bits [15:6]

Reserved, RES0.

FSC, bits [5:0]

Fault status code

Value	Meaning	Applies
0b0000xx	Address Size fault. Bits [1:0] encode the level.	
0b0001xx	Translation fault. Bits [1:0] encode the level.	
0b0010xx	Access Flag fault. Bits [1:0] encode the level.	
0b0011xx	Permission fault. Bits [1:0] encode the level.	
0b010000	Synchronous External abort on write.	
0b0101xx	Synchronous External abort on translation table walk or hardware update of translation table. Bits [1:0] encode the level.	
0b010001	Asynchronous External abort on write.	
0b100001	Alignment fault.	
0b101000	Capability tag fault.	When Morello is implemented
0b101001	Capability sealed fault.	When Morello is implemented
0b101010	Capability bound fault.	When Morello is implemented
0b101011	Capability permission fault.	When Morello is implemented
0b110000	TLB Conflict fault.	
0b110001	Unsupported atomic hardware update fault.	When ARMv8.1-TTHM is implemented

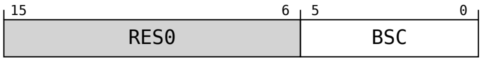
All other values are reserved. Reserved values might be defined in a future version of the architecture.

Writing a reserved value to this field will make the value of this field UNKNOWN. Values that are not supported act as reserved values when writing to this register.

It is IMPLEMENTATION DEFINED whether each of the Access Flag fault, asynchronous External abort and synchronous External abort, Alignment fault, and TLB Conflict abort values can be generated by the PE. For more information see x'Faults and Watchpoints'.

On a warm reset, this field resets to an architecturally UNKNOWN value.

other buffer management events



Bits [15:6]

Reserved, RES0.

BSC, bits [5:0]

Buffer status code

Value	Meaning
`0b000000`	Buffer not filled
`0b000001`	Buffer filled

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Writing a reserved value to this field will make the value of this field UNKNOWN. Values that are not supported act as reserved values when writing to this register.

On a warm reset, this field resets to an architecturally UNKNOWN value.

The syndrome contents for each management event are described in the following sections.

Accessing the PMBSR_EL1

Read using name PMBSR_EL1

The assembler syntax is:

```
MRS <Xt>, PMBSR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMBSR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMBSR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return PMBSR_EL1;
```

Write using name PMBSR_EL1

The assembler syntax is:

```
MSR PMBSR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1001	0b1010	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.E2PB == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMBSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' && MDCR_EL3.NSPB != '01' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && SCR_EL3.NS == '1' && MDCR_EL3.NSPB != '11' then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        PMBSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
```

```
        PMBSR_EL1 = X[t];
```

3.2.33 RDDC_EL0, Restricted Default Data Capability

The RDDC_EL0 characteristics are:

Purpose

Holds the default data capability associated when the PE is in Restricted

Attributes

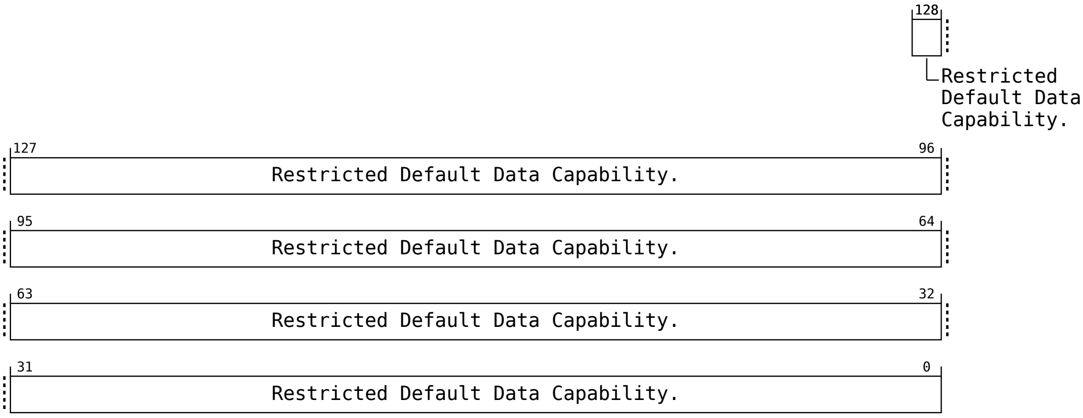
RDDC_EL0 is a 129-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to RDDC_EL0 are UNDEFINED.

Field descriptions

The RDDC_EL0 bit assignments are:


128
Restricted Default Data Capability.
127
96
Restricted Default Data Capability.
95
64
Restricted Default Data Capability.
63
32
Restricted Default Data Capability.
31
0
Restricted Default Data Capability.

Bits [128:0]

Restricted Default Data Capability.

This field resets to `0x1FFFFC000000100050000000000000000`.

Accessing the RDDC_EL0

Read using name RDDC_EL0

The assembler syntax is:

```
MRS <Ct>, RDDC_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0011	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RDDC_EL0;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RDDC_EL0;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RDDC_EL0;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RDDC_EL0;
```

Write using name RDDC_EL0

The assembler syntax is:

```
MSR RDDC_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0011	0b001

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RDDC_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RDDC_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RDDC_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RDDC_EL0 = C[t];
```

Read using name DDC

The assembler syntax is:

```
MRS <Ct>, DDC
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
```

```
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    return RDDC_EL0;
elsif PSTATE.SP == '0' then
    return DDC_EL0;
elsif PSTATE.EL == EL0 then
    return DDC_EL0;
elsif PSTATE.EL == EL1 then
    return DDC_EL1;
elsif PSTATE.EL == EL2 then
    return DDC_EL2;
elsif PSTATE.EL == EL3 then
    return DDC_EL3;
```

Write using name DDC

The assembler syntax is:

```
MSR DDC, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b0100	0b0001	0b001

Accessibility:

```
if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') &&
    ↪CPACR_EL1.CEN != '11' then
    if EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    else
        AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL == EL1 && CPACR_EL1.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL1, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' &&
    ↪CPTR_EL2.CEN != '11' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' &&
    ↪CPTR_EL2.CEN == 'x0' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif PSTATE.EL IN {EL0, EL2, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' &&
    ↪CPTR_EL2.TC == '1' then
    AArch64.SystemAccessTrap(EL2, 0x29);
elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
    AArch64.SystemAccessTrap(EL3, 0x29);
elsif IsInRestricted() then
    RDDC_EL0 = C[t];
elsif PSTATE.SP == '0' then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL0 then
    DDC_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    DDC_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    DDC_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    DDC_EL3 = C[t];
```

3.2.34 RSP_EL0, Restricted Stack Pointer

The RSP_EL0 characteristics are:

Purpose

Holds the stack pointer when the PE is in Restricted. This is used as the current stack pointer at all Exception levels when the PE is in Restricted.

Attributes

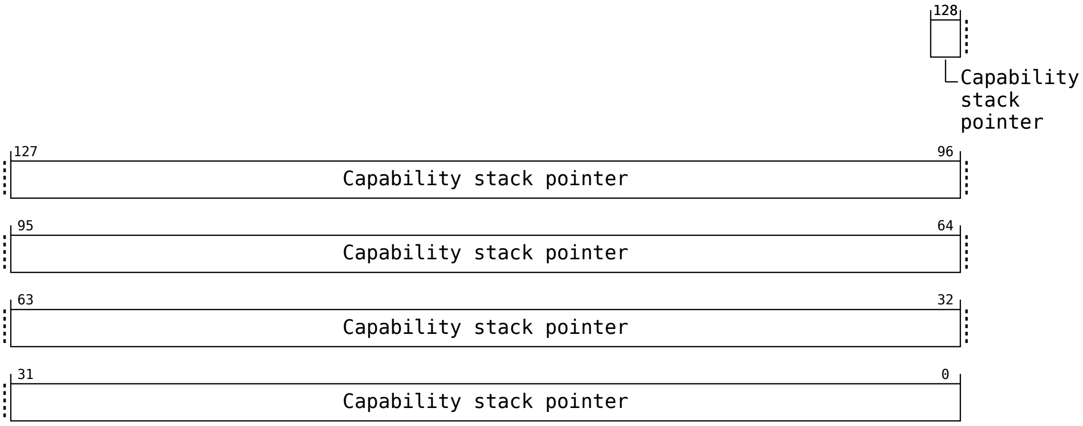
RSP_EL0 is a 129-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to RSP_EL0 are UNDEFINED.

Field descriptions

The RSP_EL0 bit assignments are:



Bits [128:0]

Capability stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the RSP_EL0

When the PE is in Restricted, this register is accessible as the current stack pointer.

Read using name RSP_EL0

The assembler syntax is:

```
MRS <Xt>, RSP_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b111	0b0100	0b0001	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11'
     ↪then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0<63:0>;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0<63:0>;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0<63:0>;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0<63:0>;
```

Write using name RSP_EL0

The assembler syntax is:

```
MSR RSP_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b111	0b0100	0b0001	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11'
     ↪then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = ZeroExtend(X[t]);
```

Read using name RCSP_EL0

The assembler syntax is:

```
MRS <Ct>, RCSP_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b111	0b0100	0b0001	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RSP_EL0;
```

Write using name RCSP_EL0

The assembler syntax is:

```
MSR RCSP_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b111	0b0100	0b0001	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11'
        ↪then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
```

```
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RSP_EL0 = C[t];
```

3.2.35 RTPIDR_EL0, Restricted Read/Write Software Thread ID Register

The RTPIDR_EL0 characteristics are:

Purpose

Provides a location where software can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Attributes

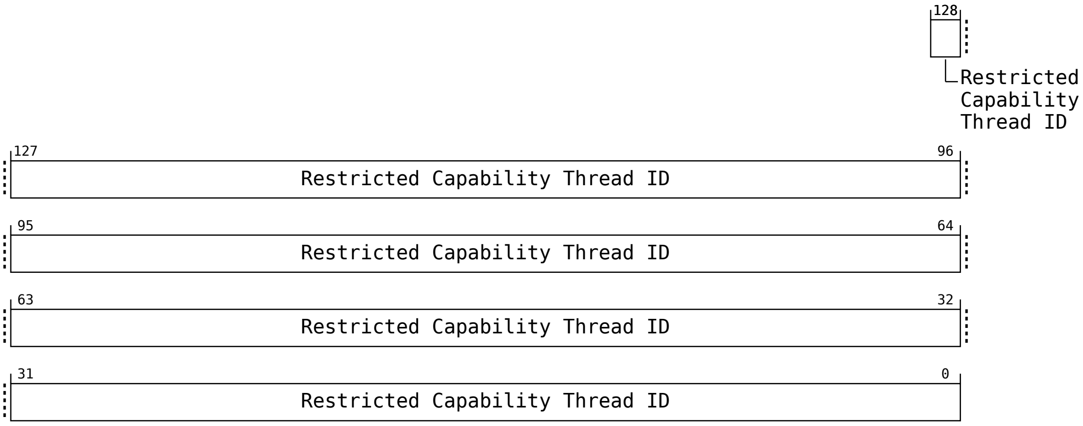
RTPIDR_EL0 is a 129-bit register.

Configuration

This register is present only when Morello is implemented. Otherwise, direct accesses to RTPIDR_EL0 are UNDEFINED.

Field descriptions

The RTPIDR_EL0 bit assignments are:


128
Restricted Capability Thread ID
127
96
Restricted Capability Thread ID
95
64
Restricted Capability Thread ID
63
32
Restricted Capability Thread ID
31
0
Restricted Capability Thread ID

Bits [128:0]

Restricted Thread ID. The version of the Thread ID when the PE is in Restricted.

This field resets to an architecturally UNKNOWN value.

Accessing the RTPIDR_EL0

Access to RTPIDR_EL0 via MSR aand MRS instructions is only possible when the PE is in Executive.

When the PE is in Restricted, operations which use TPIDR_ELx or CTPIDR_ELx access RTPIDR_EL0.

Read using name RTPIDR_EL0

The assembler syntax is:

```
MRS <Xt>, RTPIDR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11'
     ↪then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0<63:0>;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0<63:0>;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0<63:0>;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0<63:0>;
```

Write using name RTPIDR_EL0

The assembler syntax is:

```
MSR RTPIDR_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11'
     ↪then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
```

```
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = ZeroExtend(X[t]);
```

Read using name TPIDR_EL0

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL0<63:0>;
elsif PSTATE.EL == EL1 then
    return TPIDR_EL0<63:0>;
elsif PSTATE.EL == EL2 then
    return TPIDR_EL0<63:0>;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL0<63:0>;
```

Write using name TPIDR_EL0

The assembler syntax is:

```
MSR TPIDR_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL1 then
    TPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    TPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    TPIDR_EL0 = ZeroExtend(X[t]);
```

Read using name TPIDR_EL1

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    return TPIDR_EL1<63:0>;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL1<63:0>;
```

Write using name TPIDR_EL1

The assembler syntax is:

```
MSR TPIDR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    TPIDR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    TPIDR_EL1 = ZeroExtend(X[t]);
```

Read using name TPIDR_EL2

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL2<63:0>;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL2<63:0>;
```

Write using name TPIDR_EL2

The assembler syntax is:

```
MSR TPIDR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL2 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    TPIDR_EL2 = ZeroExtend(X[t]);
```

Read using name TPIDR_EL3

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL3<63:0>;
```

Write using name TPIDR_EL3

The assembler syntax is:

```
MSR TPIDR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL3 = ZeroExtend(X[t]);
```

Read using name RCTPIDR_EL0

The assembler syntax is:

```
MRS <Ct>, RCTPIDR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return RTPIDR_EL0;
```

Write using name RCTPIDR_EL0

The assembler syntax is:

```
MSR RCTPIDR_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        RTPIDR_EL0 = C[t];
```

Read using name CTPIDR_EL0

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL0;
```

Write using name CTPIDR_EL0

The assembler syntax is:

```
MSR CTPIDR_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL0 = C[t];
```

Read using name CTPIDR_EL1

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL1;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL1;
```

Write using name CTPIDR_EL1

The assembler syntax is:

```
MSR CTPIDR_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL1 = C[t];
```

Read using name CTPIDR_EL2

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL2;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL2;
```

Write using name CTPIDR_EL2

The assembler syntax is:

```
MSR CTPIDR_EL2, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL2 = C[t];
```

Read using name CTPIDR_EL3

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL3;
```

Write using name CTPIDR_EL3

The assembler syntax is:

```
MSR CTPIDR_EL3, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL3 = C[t];
```

3.2.36 SP_EL0, Stack Pointer (EL0)

The SP_EL0 characteristics are:

Purpose

Holds the capability stack pointer associated with EL0 and Executive state. At higher Exception levels, this is used as the current capability stack pointer when the value of SPSel.SP is 0 and the PE is in Executive.

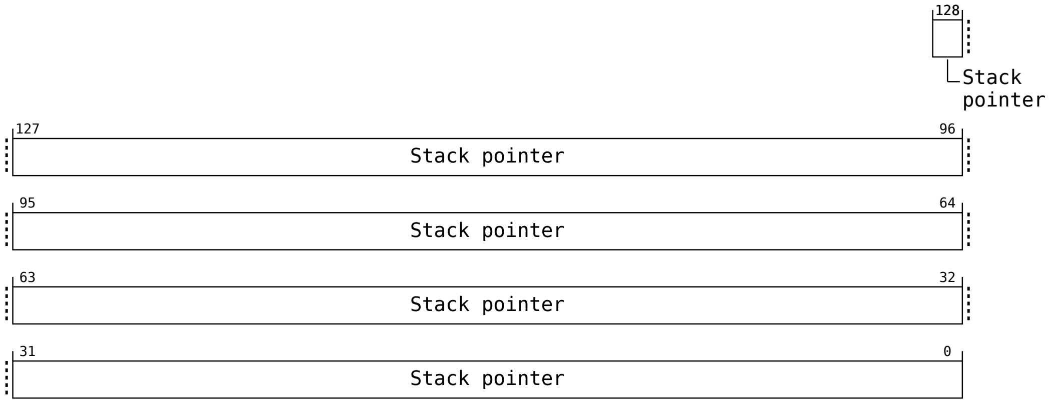
Attributes

SP_EL0 is a 129-bit register.

Field descriptions

The SP_EL0 bit assignments are:

When Morello is implemented:

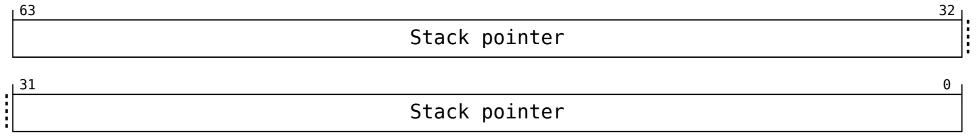


Bits [128:0]

Stack pointer

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the SP_EL0

When the value of PSTATE.SP is 0 and the PE is in Executive, this register is accessible at all Exception levels as the current stack pointer.

Read using name SP_EL0

The assembler syntax is:

```
MRS <Xt>, SP_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        return SP_EL0<63:0>;
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        return SP_EL0<63:0>;
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        return SP_EL0<63:0>;
```

Write using name SP_EL0

The assembler syntax is:

```
MSR SP_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        SP_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
```

```
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        SP_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        SP_EL0 = ZeroExtend(X[t]);
```

Read using name CSP_EL0

The assembler syntax is:

```
MRS <Ct>, CSP_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return SP_EL0;
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return SP_EL0;
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return SP_EL0;
```

Write using name CSP_EL0

The assembler syntax is:

```
MSR CSP_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        SP_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        SP_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if PSTATE.SP == '0' then
        UNDEFINED;
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        SP_EL0 = C[t];
```

3.2.37 SP_EL1, Stack Pointer (EL1)

The SP_EL1 characteristics are:

Purpose

Holds the capability stack pointer associated with EL1 and Executive. When executing at EL1, the values of SPSel.SP and the Executive bit of PCC determine the current capability stack pointer:

SPSel.SP	Executive bit of PCC	Current stack pointer
0bx	0b0	RSP_EL0
0b0	0b1	SP_EL0
0b1	0b1	SP_EL1

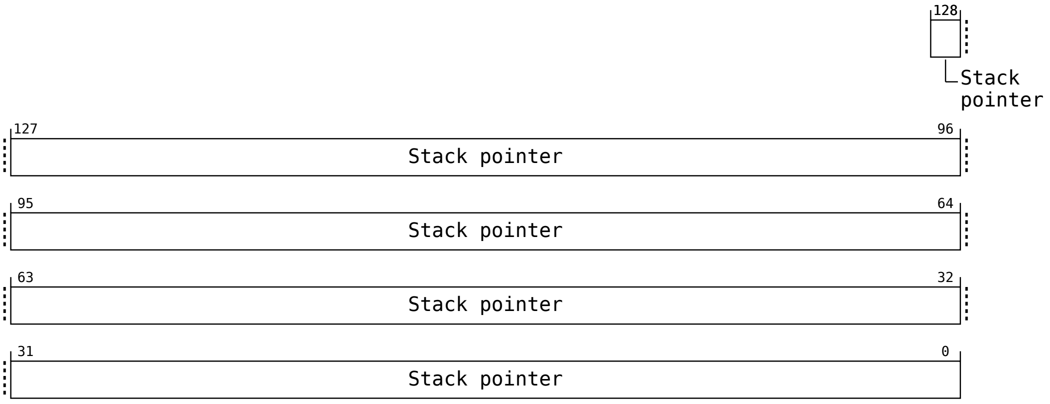
Attributes

SP_EL1 is a 129-bit register.

Field descriptions

The SP_EL1 bit assignments are:

When Morello is implemented:

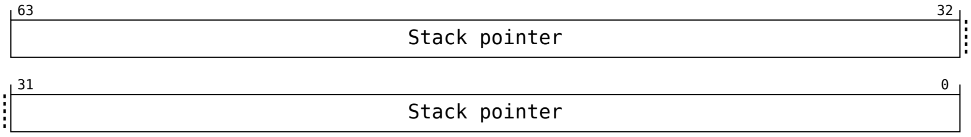


Bits [128:0]

Stack pointer

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the SP_EL1

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL1 as the current stack pointer.

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Read using name SP_EL1

The assembler syntax is:

```
MRS <Xt>, SP_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        return SP_EL1<63:0>;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        return SP_EL1<63:0>;
```

Write using name SP_EL1

The assembler syntax is:

```
MSR SP_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        SP_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
```

```
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        SP_EL1 = ZeroExtend(X[t]);
```

Read using name CSP_EL1

The assembler syntax is:

```
MRS <Ct>, CSP_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return SP_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return SP_EL1;
```

Write using name CSP_EL1

The assembler syntax is:

```
MSR CSP_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
```

```
        UNDEFINED;
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        SP_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        SP_EL1 = C[t];
```

3.2.38 SP_EL2, Stack Pointer (EL2)

The SP_EL2 characteristics are:

Purpose

Holds the capability stack pointer associated with EL2 and Executive state. When executing at EL2, the values of SPSel.SP and the Executive bit of PCC determine the current capability stack pointer:

SPSel.SP	Executive bit of PCC	Current stack pointer
0bx	0b0	RSP_EL0
0b0	0b1	SP_EL0
0b1	0b1	SP_EL2

Attributes

SP_EL2 is a 129-bit register.

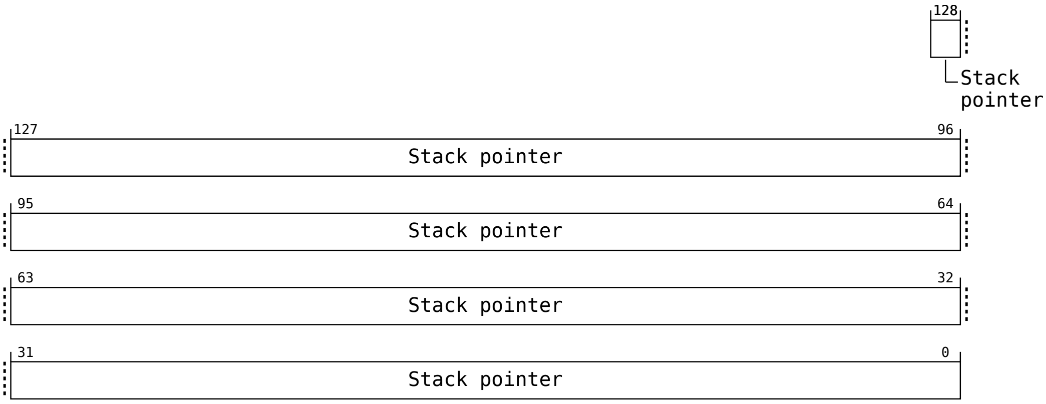
Configuration

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

Field descriptions

The SP_EL2 bit assignments are:

When Morello is implemented:

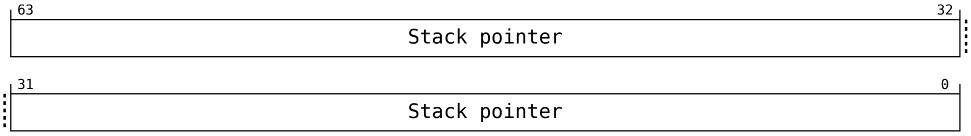


Bits [128:0]

Stack pointer

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the SP_EL2

This accessibility information only applies to accesses using the MRS or MSR instructions.

When the value of SPSel.SP is 1, this register is also accessible at EL2 as the current stack pointer.

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

Read using name SP_EL2

The assembler syntax is:

```
MRS <Xt>, SP_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        return SP_EL2<63:0>;
```

Write using name SP_EL2

The assembler syntax is:

```
MSR SP_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    else
        SP_EL2 = ZeroExtend(X[t]);
```

Read using name CSP_EL2

The assembler syntax is:

```
MRS <Ct>, CSP_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return SP_EL2;
```

Write using name CSP_EL2

The assembler syntax is:

```
MSR CSP_EL2, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0001	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
```

```
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        UNDEFINED;
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        SP_EL2 = C[t];
```

3.2.39 SP_EL3, Stack Pointer (EL3)

The SP_EL3 characteristics are:

Purpose

Holds the capability stack pointer associated with EL3. When executing at EL3, the values of SPSel.SP and the Executive bit of PCC determine the current capability stack pointer:

SPSel.SP	Executive bit of PCC	Current stack pointer
0bx	0b0	RSP_EL0
0b0	0b1	SP_EL0
0b1	0b1	SP_EL3

Attributes

SP_EL3 is a 129-bit register.

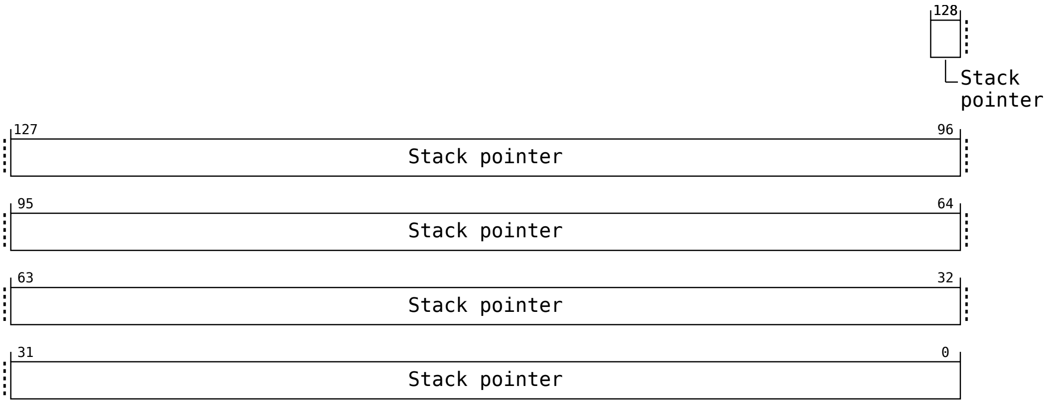
Configuration

This register is present only when HaveEL(EL3). Otherwise, direct accesses to SP_EL3 are UNDEFINED.

Field descriptions

The SP_EL3 bit assignments are:

When Morello is implemented:

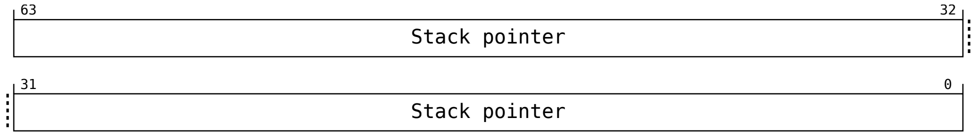


Bits [128:0]

Stack pointer

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Stack pointer.

This field resets to an architecturally UNKNOWN value.

Accessing the SP_EL3

This register is not accessible using MRS and MSR instructions.

When the value of SPSel.SP is 1, this register is accessible at EL3 as the current stack pointer.

When the value of SPSel.SP is 0, SP_EL0 is used as the current stack pointer at all Exception levels.

3.2.40 SPSR_EL1, Saved Program Status Register (EL1)

The SPSR_EL1 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL1.

Attributes

SPSR_EL1 is a 64-bit register.

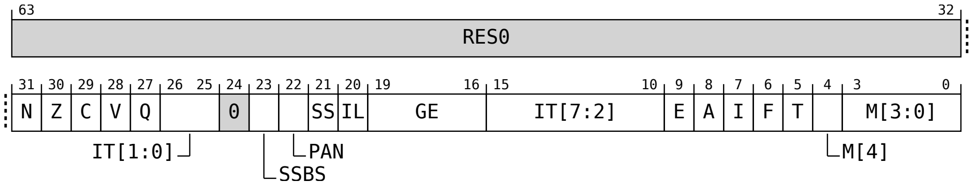
Configuration

AArch64 System register SPSR_EL1[31:0] is architecturally mapped to AArch32 System register SPSR_svc[31:0].

Field descriptions

The SPSR_EL1 bit assignments are:

When exception taken from AArch32 state:



An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL1, and copied to PSTATE.Q on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to EL1, and copied to PSTATE.IT[1:0] on executing an exception return operation in EL1.

On executing an exception return operation in EL1 SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

Bit [24]

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL1, and copied to PSTATE.GE on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to EL1, and copied to PSTATE.IT[7:2] on executing an exception return operation in EL1.

SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL1, and copied to PSTATE.E on executing an exception return operation in EL1.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an exception return operation in EL1, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL1, and copied to PSTATE.T on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to 0b1, the value of PSTATE.nRW, on taking an exception to EL1 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

Value	Meaning
`0b1`	AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

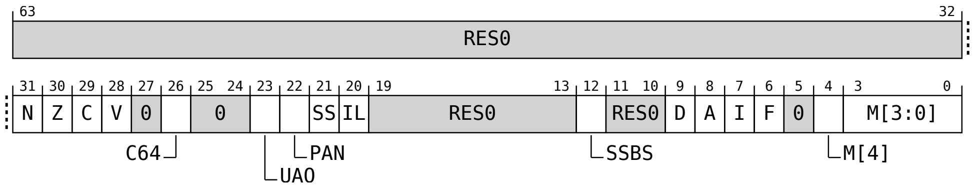
AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL1, and copied to PSTATE.M[3:0] on executing an exception return operation in EL1.

Value	Meaning
`0b0000`	User.
`0b0001`	FIQ.
`0b0010`	IRQ.
`0b0011`	Supervisor.
`0b0111`	Abort.
`0b1011`	Undefined.
`0b1111`	System.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in x'Illegal return events from AArch64 state'.

This field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:



An exception return from EL1 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL1, and copied to PSTATE.N on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL1, and copied to PSTATE.Z on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL1, and copied to PSTATE.C on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL1, and copied to PSTATE.V on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Bit [27]

Reserved, RES0.

C64, bit [26]

When Morello is implemented:

Current instruction set state. Set to the value of PSTATE.C64 on taking an exception to EL1, and copied to PSTATE.C64 on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [25:24]

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL1, and copied to PSTATE.UAO on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL1, and copied to PSTATE.PAN on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL1, and conditionally copied to PSTATE.SS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL1, and copied to PSTATE.IL on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL1, and copied to PSTATE.SSBS on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [11:10]

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL1, and copied to PSTATE.D on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL1, and copied to PSTATE.A on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL1, and copied to PSTATE.I on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL1, and copied to PSTATE.F on executing an exception return operation in EL1.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0b0, the value of PSTATE.nRW, on taking an exception to EL1 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL1.

Value	Meaning
`0b0`	AArch64 execution state.

If AArch32 is not supported at any Exception level, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

Value	Meaning
`0b0000`	EL0t.
`0b0100`	EL1t.
`0b0101`	EL1h.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL1 is an illegal return event, as described in x'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL1 and copied to PSTATE.EL on executing an exception return operation in EL1.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL1 and copied to PSTATE.SP on executing an exception return operation in EL1

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using the mnemonic SPSR_EL1 or SPSR_EL12 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name SPSR_EL1

The assembler syntax is:

```
MRS <Xt>, SPSR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return SPSR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SPSR_EL2;
    else
        return SPSR_EL1;
elsif PSTATE.EL == EL3 then
    return SPSR_EL1;
```

Write using name SPSR_EL1

The assembler syntax is:

```
MSR SPSR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    SPSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SPSR_EL2 = X[t];
    else
        SPSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_EL1 = X[t];
```

Read using name SPSR_EL12

The assembler syntax is:

```
MRS <Xt>, SPSR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SPSR_EL1;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        return SPSR_EL1;
    else
        UNDEFINED;
```

Write using name SPSR_EL12

The assembler syntax is:

```
MSR SPSR_EL12, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SPSR_EL1 = X[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        SPSR_EL1 = X[t];
    else
        UNDEFINED;
```

3.2.41 SPSR_EL2, Saved Program Status Register (EL2)

The SPSR_EL2 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL2.

Attributes

SPSR_EL2 is a 64-bit register.

Configuration

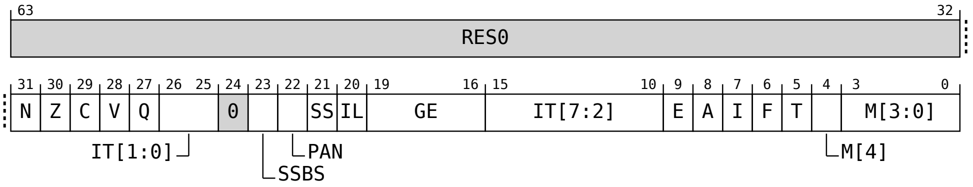
This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register SPSR_EL2[31:0] is architecturally mapped to AArch32 System register SPSR_hyp[31:0].

Field descriptions

The SPSR_EL2 bit assignments are:

When exception taken from AArch32 state:



An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL2, and copied to PSTATE.Q on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to EL2, and copied to PSTATE.IT[1:0] on executing an exception return operation in EL2.

On executing an exception return operation in EL2 SPSR_EL2.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

Bit [24]

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL2, and copied to PSTATE.GE on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to EL2, and copied to PSTATE.IT[7:2] on executing an exception return operation in EL2.

SPSR_EL2.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL2, and copied to PSTATE.E on executing an exception return operation in EL2.

If the implementation does not support big-endian operation, SPSR_EL2.E is RES0. If the implementation does not support little-endian operation, SPSR_EL2.E is RES1. On executing an exception return operation in EL2, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL2.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL2.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL2, and copied to PSTATE.T on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to 0b1, the value of PSTATE.nRW, on taking an exception to EL2 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

Value	Meaning
0b1	AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

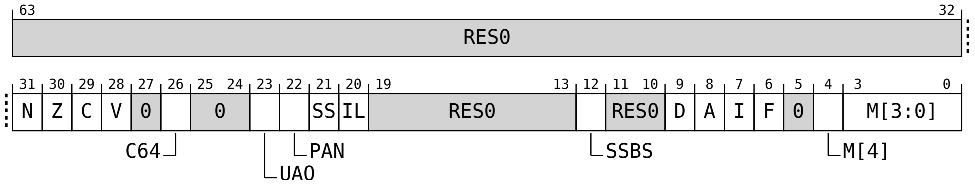
AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL2, and copied to PSTATE.M[3:0] on executing an exception return operation in EL2.

Value	Meaning
0b0000	User.
0b0001	FIQ.
0b0010	IRQ.
0b0011	Supervisor.
0b0111	Abort.
0b1010	Hyp.
0b1011	Undefined.
0b1111	System.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in x'Illegal return events from AArch64 state'.

This field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:



An exception return from EL2 using AArch64 makes SPSR_EL2 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL2, and copied to PSTATE.N on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL2, and copied to PSTATE.Z on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL2, and copied to PSTATE.C on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL2, and copied to PSTATE.V on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Bit [27]

Reserved, RES0.

C64, bit [26]

When Morello is implemented:

Current instruction set state. Set to the value of PSTATE.C64 on taking an exception to EL2, and copied to PSTATE.C64 on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [25:24]

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL2, and copied to PSTATE.UAO on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL2, and copied to PSTATE.PAN on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL2, and conditionally copied to PSTATE.SS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL2, and copied to PSTATE.IL on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL2, and copied to PSTATE.SSBS on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [11:10]

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL2, and copied to PSTATE.D on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL2, and copied to PSTATE.A on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL2, and copied to PSTATE.I on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL2, and copied to PSTATE.F on executing an exception return operation in EL2.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0b0, the value of PSTATE.nRW, on taking an exception to EL2 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL2.

Value	Meaning
0b0	AArch64 execution state.

If AArch32 is not supported at any Exception level, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

Value	Meaning
0b0000	EL0t.
0b0100	EL1t.
0b0101	EL1h.
0b1000	EL2t.
0b1001	EL2h.

Other values are reserved. If SPSR_EL2.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL2 is an illegal return event, as described in x'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL2 and copied to PSTATE.EL on executing an exception return operation in EL2.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL2 and copied to PSTATE.SP on executing an exception return operation in EL2

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using the mnemonic SPSR_EL2 or SPSR_EL1 are not guaranteed to be ordered with respect to accesses using the other mnemonic.

Read using name SPSR_EL2

The assembler syntax is:

```
MRS <Xt>, SPSR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    return SPSR_EL2;
elsif PSTATE.EL == EL3 then
    return SPSR_EL2;
```

Write using name SPSR_EL2

The assembler syntax is:

```
MSR SPSR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    SPSR_EL2 = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_EL2 = X[t];
```

Read using name SPSR_EL1

The assembler syntax is:

```
MRS <Xt>, SPSR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
```

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    return SPSR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        return SPSR_EL2;
    else
        return SPSR_EL1;
elsif PSTATE.EL == EL3 then
    return SPSR_EL1;
```

Write using name SPSR_EL1

The assembler syntax is:

```
MSR SPSR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    SPSR_EL1 = X[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        SPSR_EL2 = X[t];
    else
        SPSR_EL1 = X[t];
elsif PSTATE.EL == EL3 then
    SPSR_EL1 = X[t];
```

3.2.42 SPSR_EL3, Saved Program Status Register (EL3)

The SPSR_EL3 characteristics are:

Purpose

Holds the saved process state when an exception is taken to EL3.

Attributes

SPSR_EL3 is a 64-bit register.

Configuration

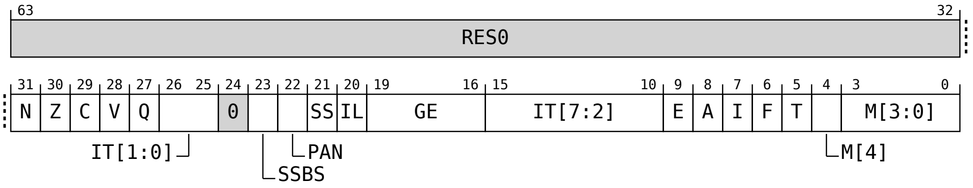
AArch64 System register SPSR_EL3[31:0] can be mapped to AArch32 System register SPSR_mon[31:0], but this is not architecturally mandated.

This register is present only when HaveEL(EL3). Otherwise, direct accesses to SPSR_EL3 are UNDEFINED.

Field descriptions

The SPSR_EL3 bit assignments are:

When exception taken from AArch32 state:



An exception return from EL3 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Q, bit [27]

Overflow or saturation flag. Set to the value of PSTATE.Q on taking an exception to EL3, and copied to PSTATE.Q on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

IT[1:0], bits [26:25]

If-Then. Set to the value of PSTATE.IT[1:0] on taking an exception to EL3, and copied to PSTATE.IT[1:0] on executing an exception return operation in EL3.

On executing an exception return operation in EL3 SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

Bit [24]

Reserved, RES0.

SSBS, bit [23]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

GE, bits [19:16]

Greater than or Equal flags. Set to the value of PSTATE.GE on taking an exception to EL3, and copied to PSTATE.GE on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

IT[7:2], bits [15:10]

If-Then. Set to the value of PSTATE.IT[7:2] on taking an exception to EL3, and copied to PSTATE.IT[7:2] on executing an exception return operation in EL3.

SPSR_EL1.IT must contain a value that is valid for the instruction being returned to.

This field resets to an architecturally UNKNOWN value.

E, bit [9]

Endianness. Set to the value of PSTATE.E on taking an exception to EL3, and copied to PSTATE.E on executing an exception return operation in EL3.

If the implementation does not support big-endian operation, SPSR_EL1.E is RES0. If the implementation does not support little-endian operation, SPSR_EL1.E is RES1. On executing an exception return operation in EL3, if the implementation does not support big-endian operation at the Exception level being returned to, SPSR_EL1.E is RES0, and if the implementation does not support little-endian operation at the Exception level being returned to, SPSR_EL1.E is RES1.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

T, bit [5]

T32 Instruction set state. Set to the value of PSTATE.T on taking an exception to EL3, and copied to PSTATE.T on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

M[4], bit [4]

Execution state. Set to 0b1, the value of PSTATE.nRW, on taking an exception to EL3 from AArch32 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

Value	Meaning
`0b1`	AArch32 execution state.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

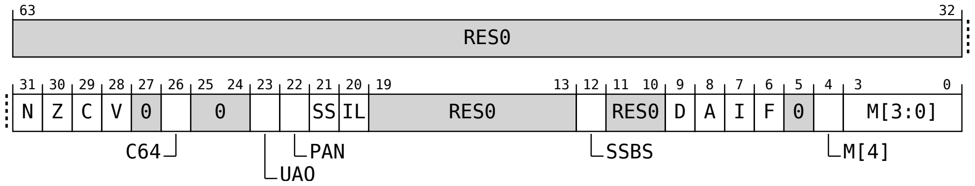
AArch32 Mode. Set to the value of PSTATE.M[3:0] on taking an exception to EL3, and copied to PSTATE.M[3:0] on executing an exception return operation in EL3.

Value	Meaning
`0b0000`	User.
`0b0001`	FIQ.
`0b0010`	IRQ.
`0b0011`	Supervisor.
`0b0110`	Monitor.
`0b0111`	Abort.
`0b1010`	Hyp.
`0b1011`	Undefined.
`0b1111`	System.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in x'Illegal return events from AArch64 state'.

This field resets to an architecturally UNKNOWN value.

When exception taken from AArch64 state:



An exception return from EL3 using AArch64 makes SPSR_EL1 become UNKNOWN.

Bits [63:32]

Reserved, RES0.

N, bit [31]

Negative Condition flag. Set to the value of PSTATE.N on taking an exception to EL3, and copied to PSTATE.N on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Z, bit [30]

Zero Condition flag. Set to the value of PSTATE.Z on taking an exception to EL3, and copied to PSTATE.Z on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

C, bit [29]

Carry Condition flag. Set to the value of PSTATE.C on taking an exception to EL3, and copied to PSTATE.C on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

V, bit [28]

Overflow Condition flag. Set to the value of PSTATE.V on taking an exception to EL3, and copied to PSTATE.V on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Bit [27]

Reserved, RES0.

C64, bit [26]

When Morello is implemented:

Current instruction set state. Set to the value of PSTATE.C64 on taking an exception to EL3, and copied to PSTATE.C64 on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bit [25:24]

Reserved, RES0.

UAO, bit [23]

When ARMv8.2-UAO is implemented:

User Access Override. Set to the value of PSTATE.UAO on taking an exception to EL3, and copied to PSTATE.UAO on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

PAN, bit [22]

When ARMv8.1-PAN is implemented:

Privileged Access Never. Set to the value of PSTATE.PAN on taking an exception to EL3, and copied to PSTATE.PAN on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

SS, bit [21]

Software Step. Set to the value of PSTATE.SS on taking an exception to EL3, and conditionally copied to PSTATE.SS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

IL, bit [20]

Illegal Execution state. Set to the value of PSTATE.IL on taking an exception to EL3, and copied to PSTATE.IL on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Bits [19:13]

Reserved, RES0.

SSBS, bit [12]

When ARMv8.0-SSBS is implemented:

Speculative Store Bypass. Set to the value of PSTATE.SSBS on taking an exception to EL3, and copied to PSTATE.SSBS on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Otherwise:

RES0

Bits [11:10]

Reserved, RES0.

D, bit [9]

Debug exception mask. Set to the value of PSTATE.D on taking an exception to EL3, and copied to PSTATE.D on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

A, bit [8]

SError interrupt mask. Set to the value of PSTATE.A on taking an exception to EL3, and copied to PSTATE.A on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

I, bit [7]

IRQ interrupt mask. Set to the value of PSTATE.I on taking an exception to EL3, and copied to PSTATE.I on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

F, bit [6]

FIQ interrupt mask. Set to the value of PSTATE.F on taking an exception to EL3, and copied to PSTATE.F on executing an exception return operation in EL3.

This field resets to an architecturally UNKNOWN value.

Bit [5]

Reserved, RES0.

M[4], bit [4]

Execution state. Set to 0b0, the value of PSTATE.nRW, on taking an exception to EL3 from AArch64 state, and copied to PSTATE.nRW on executing an exception return operation in EL3.

Value	Meaning
`0b0`	AArch64 execution state.

If AArch32 is not supported at any Exception level, this bit is RES0.

This field resets to an architecturally UNKNOWN value.

M[3:0], bits [3:0]

AArch64 Exception level and selected Stack Pointer.

Value	Meaning
`0b0000`	EL0t.
`0b0100`	EL1t.
`0b0101`	EL1h.
`0b1000`	EL2t.
`0b1001`	EL2h.
`0b1100`	EL3t.
`0b1101`	EL3h.

Other values are reserved. If SPSR_EL1.M[3:0] has a Reserved value, or a value for an unimplemented Exception level, executing an exception return operation in EL3 is an illegal return event, as described in x'Illegal return events from AArch64 state'.

The bits in this field are interpreted as follows:

- M[3:2] is set to the value of PSTATE.EL on taking an exception to EL3 and copied to PSTATE.EL on executing an exception return operation in EL3.
- M[1] is unused and is 0 for all non-reserved values.
- M[0] is set to the value of PSTATE.SP on taking an exception to EL3 and copied to PSTATE.SP on executing an exception return operation in EL3

This field resets to an architecturally UNKNOWN value.

Accessing the SPSR_EL3

Read using name SPSR_EL3

The assembler syntax is:

```
MRS <Xt>, SPSR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SPSR_EL3;
```

Write using name SPSR_EL3

The assembler syntax is:

```
MSR SPSR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b0100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    SPSR_EL3 = X[t];
```

3.2.43 TPIDR_EL0, EL0 Read/Write Software Thread ID Register

The TPIDR_EL0 characteristics are:

Purpose

Provides a location where software executing at EL0 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Attributes

TPIDR_EL0 is a 129-bit register.

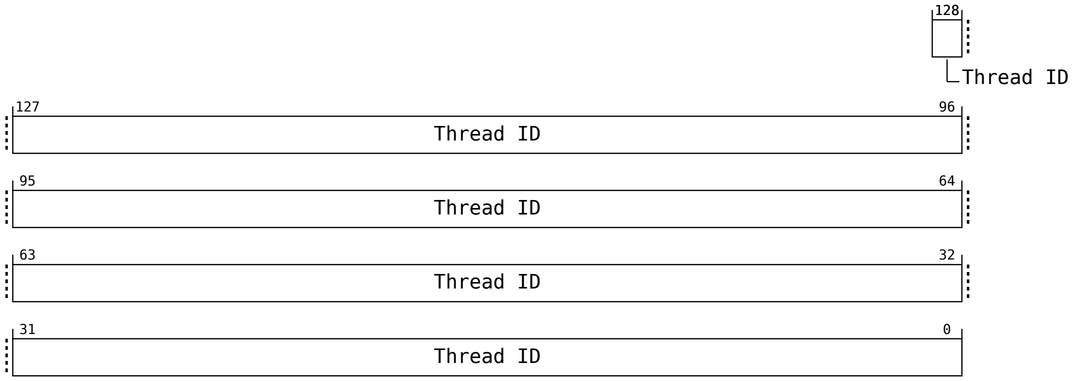
Configuration

AArch64 System register TPIDR_EL0[31:0] is architecturally mapped to AArch32 System register TPIDRURW[31:0].

Field descriptions

The TPIDR_EL0 bit assignments are:

When Morello is implemented:

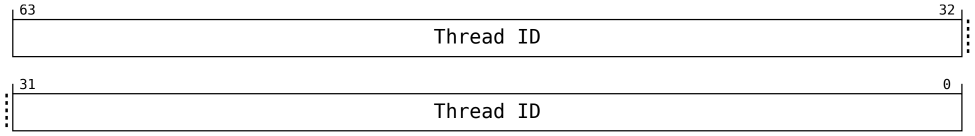


Bits [128:0]

Thread ID. Thread identifying information stored by software running at this Exception level

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL0

Read using name TPIDR_EL0

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL0<63:0>;
elsif PSTATE.EL == EL1 then
    return TPIDR_EL0<63:0>;
elsif PSTATE.EL == EL2 then
    return TPIDR_EL0<63:0>;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL0<63:0>;
```

Write using name TPIDR_EL0

The assembler syntax is:

```
MSR TPIDR_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL1 then
    TPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    TPIDR_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    TPIDR_EL0 = ZeroExtend(X[t]);
```

Read using name CTPIDR_EL0

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL0;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL0;
```

Write using name CTPIDR_EL0

The assembler syntax is:

```
MSR CTPIDR_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL0 = C[t];
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL0 = C[t];
```

3.2.44 TPIDR_EL1, EL1 Software Thread ID Register

The TPIDR_EL1 characteristics are:

Purpose

Provides a location where software executing at EL1 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Attributes

TPIDR_EL1 is a 129-bit register.

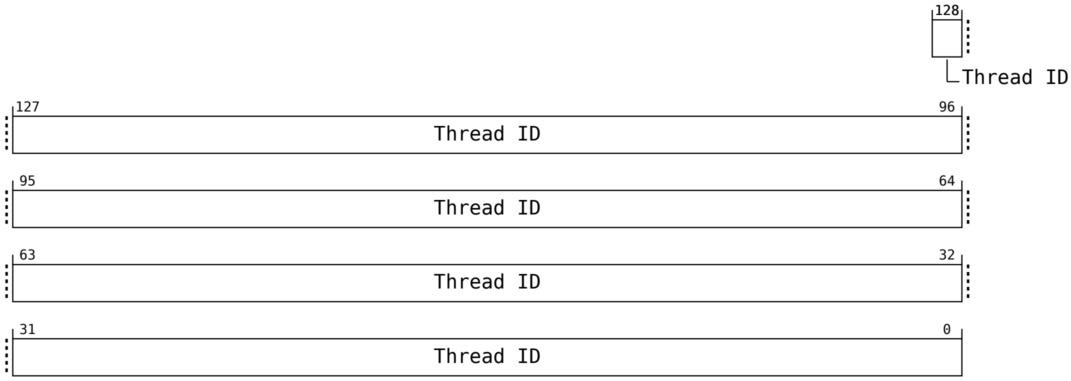
Configuration

AArch64 System register TPIDR_EL1[31:0] is architecturally mapped to AArch32 System register TPIDRPRW[31:0].

Field descriptions

The TPIDR_EL1 bit assignments are:

When Morello is implemented:

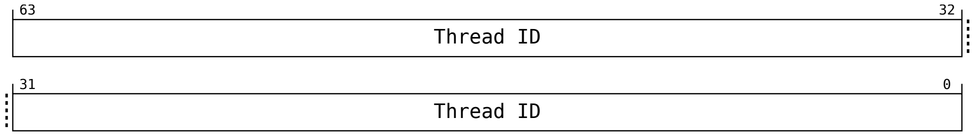


Bits [128:0]

Thread ID. Thread identifying information stored by software running at this Exception level

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL1

Read using name TPIDR_EL1

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    return TPIDR_EL1<63:0>;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL1<63:0>;
```

Write using name TPIDR_EL1

The assembler syntax is:

```
MSR TPIDR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    TPIDR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    TPIDR_EL1 = ZeroExtend(X[t]);
```

Read using name CTPIDR_EL1

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL1;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL1;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL1;
```

Write using name CTPIDR_EL1

The assembler syntax is:

```
MSR CTPIDR_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1101	0b0000	0b100

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL1 = C[t];
```

3.2.45 TPIDR_EL2, EL2 Software Thread ID Register

The TPIDR_EL2 characteristics are:

Purpose

Provides a location where software executing at EL2 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Attributes

TPIDR_EL2 is a 129-bit register.

Configuration

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

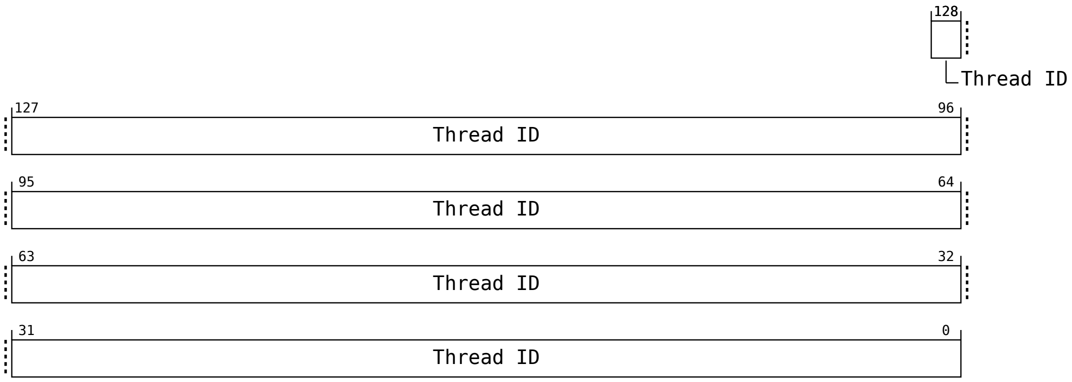
This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register TPIDR_EL2[31:0] is architecturally mapped to AArch32 System register HTPIDR[31:0].

Field descriptions

The TPIDR_EL2 bit assignments are:

When Morello is implemented:

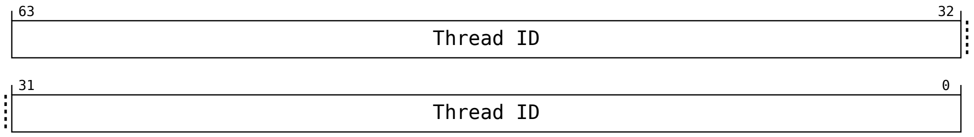


Bits [128:0]

Thread ID. Thread identifying information stored by software running at this Exception level

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL2

Read using name TPIDR_EL2

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL2<63:0>;
elsif PSTATE.EL == EL3 then
    return TPIDR_EL2<63:0>;
```

Write using name TPIDR_EL2

The assembler syntax is:

```
MSR TPIDR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL2 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    TPIDR_EL2 = ZeroExtend(X[t]);
```

Read using name CTPIDR_EL2

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL2;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDR_EL2;
```

Write using name CTPIDR_EL2

The assembler syntax is:

```
MSR CTPIDR_EL2, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDR_EL2 = C[t];
```

3.2.46 TPIDR_EL3, EL3 Software Thread ID Register

The TPIDR_EL3 characteristics are:

Purpose

Provides a location where software executing at EL3 can store thread identifying information, for OS management purposes.

The PE makes no use of this register.

Attributes

TPIDR_EL3 is a 129-bit register.

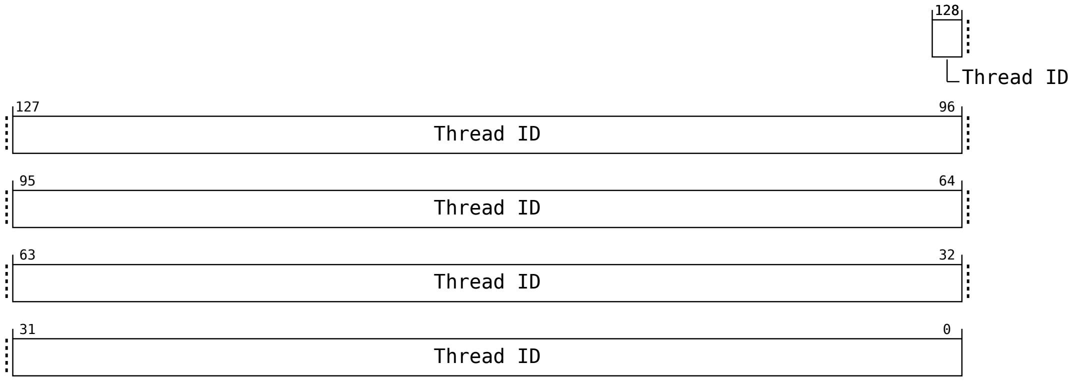
Configuration

This register is present only when HaveEL(EL3). Otherwise, direct accesses to TPIDR_EL3 are UNDEFINED.

Field descriptions

The TPIDR_EL3 bit assignments are:

When Morello is implemented:

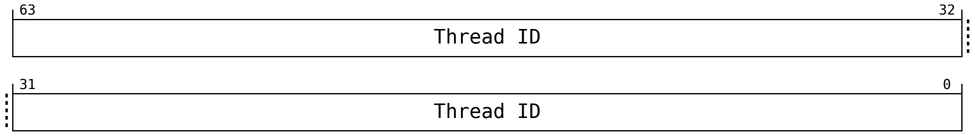


Bits [128:0]

Thread ID. Thread identifying information stored by software running at this Exception level

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDR_EL3

Read using name TPIDR_EL3

The assembler syntax is:

```
MRS <Xt>, TPIDR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0<63:0>;
    else
        return TPIDR_EL3<63:0>;
```

Write using name TPIDR_EL3

The assembler syntax is:

```
MSR TPIDR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = ZeroExtend(X[t]);
    else
        TPIDR_EL3 = ZeroExtend(X[t]);
```

Read using name CTPIDR_EL3

The assembler syntax is:

```
MRS <Ct>, CTPIDR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        return RTPIDR_EL0;
    else
        return TPIDR_EL3;
```

Write using name CTPIDR_EL3

The assembler syntax is:

```
MSR CTPIDR_EL3, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1101	0b0000	0b010

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif IsFeatureImplemented("Morello") && IsInRestricted() && !Halted() then
        RTPIDR_EL0 = C[t];
    else
        TPIDR_EL3 = C[t];
```

3.2.47 TPIDRRO_EL0, EL0 Read-Only Software Thread ID Register

The TPIDRRO_EL0 characteristics are:

Purpose

Provides a location where software executing at EL1 or higher can store thread identifying information that is visible to software executing at EL0, for OS management purposes.

The PE makes no use of this register.

Attributes

TPIDRRO_EL0 is a 129-bit register.

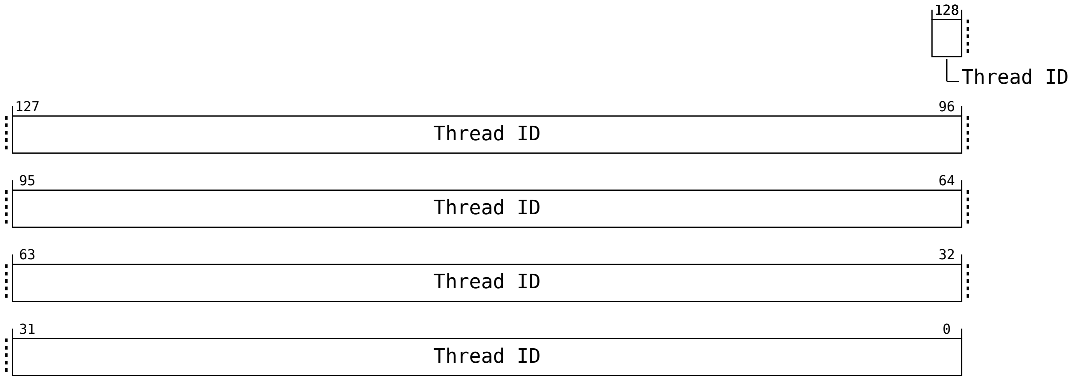
Configuration

AArch64 System register TPIDRRO_EL0[31:0] is architecturally mapped to AArch32 System register TPIDRURO[31:0].

Field descriptions

The TPIDRRO_EL0 bit assignments are:

When Morello is implemented:

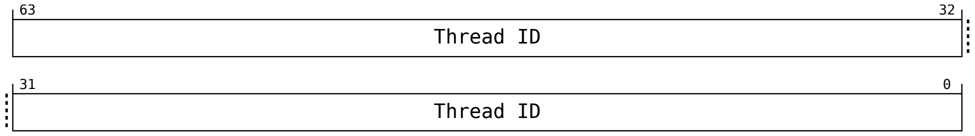


Bits [128:0]

Thread ID. Thread identifying information stored by software running at this Exception level

This field resets to an architecturally UNKNOWN value.

When Morello is not implemented:



Bits [63:0]

Thread ID. Thread identifying information stored by software running at this Exception level.

This field resets to an architecturally UNKNOWN value.

Accessing the TPIDRRO_EL0

Read using name TPIDRRO_EL0

The assembler syntax is:

```
MRS <Xt>, TPIDRRO_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    return TPIDRRO_EL0<63:0>;
elsif PSTATE.EL == EL1 then
    return TPIDRRO_EL0<63:0>;
elsif PSTATE.EL == EL2 then
    return TPIDRRO_EL0<63:0>;
elsif PSTATE.EL == EL3 then
    return TPIDRRO_EL0<63:0>;
```

Write using name TPIDRRO_EL0

The assembler syntax is:

```
MSR TPIDRRO_EL0, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    TPIDRRO_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    TPIDRRO_EL0 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    TPIDRRO_EL0 = ZeroExtend(X[t]);
```

Read using name CTPIDRRO_EL0

The assembler syntax is:

```
MRS <Ct>, CTPIDRRO_EL0
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    if !ELUsingAArch32(EL1) && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') && CPACR_EL1.CEN != '11' then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        else
            AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H,TGE> == '11' && CPTR_EL2.CEN != '11' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDRRO_EL0;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDRRO_EL0;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDRRO_EL0;
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return TPIDRRO_EL0;
```

Write using name CTPIDRRO_EL0

The assembler syntax is:

```
MSR CTPIDRRO_EL0, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b011	0b1101	0b0000	0b011

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
```

```
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDRRO_EL0 = C[t];
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDRRO_EL0 = C[t];
elsif PSTATE.EL == EL3 then
    if CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        TPIDRRO_EL0 = C[t];
```

3.2.48 VBAR_EL1, Vector Base Address Register (EL1)

The VBAR_EL1 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL1.

Attributes

VBAR_EL1 is a 129-bit register.

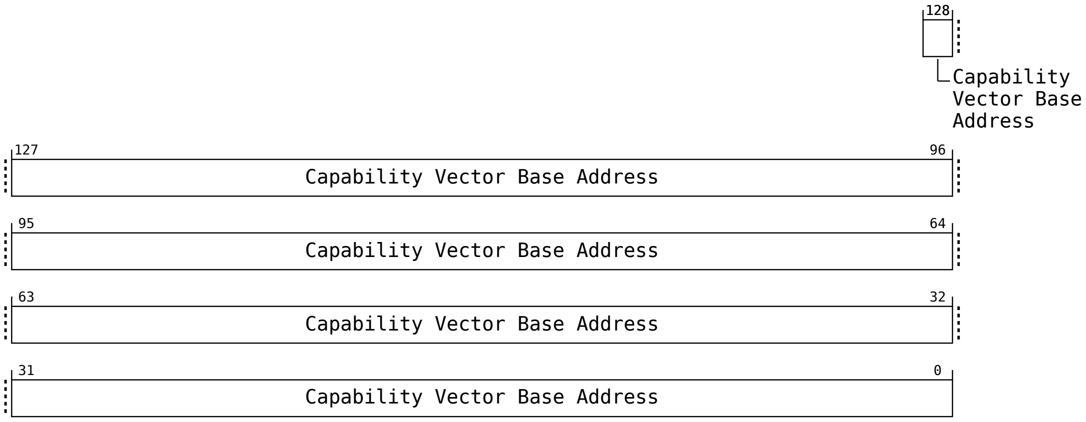
Configuration

AArch64 System register VBAR_EL1[31:0] is architecturally mapped to AArch32 System register VBAR[31:0].

Field descriptions

The VBAR_EL1 bit assignments are:

When Morello is implemented and Capability access at EL1 is not trapped:


128
Capability Vector Base Address
127
96
Capability Vector Base Address
95
64
Capability Vector Base Address
63
32
Capability Vector Base Address
31
0
Capability Vector Base Address

Bits [128:0]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

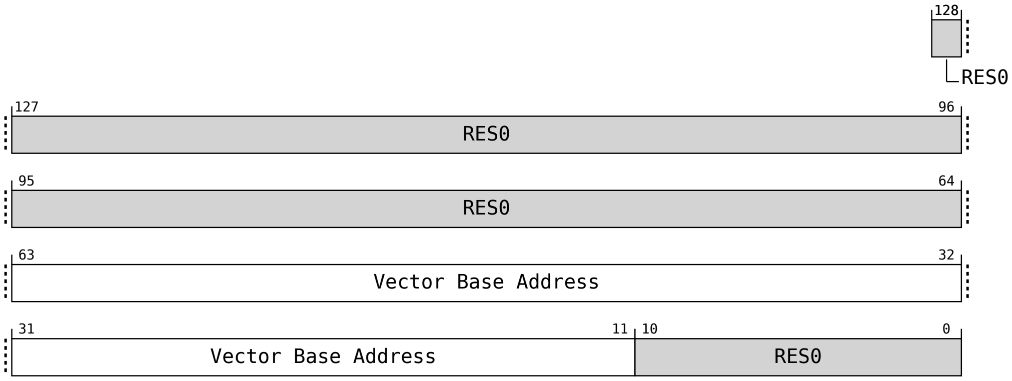
If the implementation supports xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

Bits [10:0] are treated as 0 for the purpose of calculating the exception vector address.

This field resets to an architecturally UNKNOWN value.

When Morello is implemented and Capability access at EL1 is trapped:



Bits [128:64]

Reserved, RES0.

Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports xARMv8.2-LVA, then:

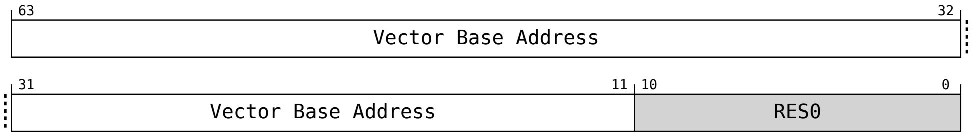
- If tagged addresses are being used, bits [55:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

When Morello is not implemented:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL1.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

- If tagged addresses are not being used, bits [63:48] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL1 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL1

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL3 using a mnemonic ending in _EL1 or _EL12 are not guaranteed to be ordered with respect to accesses using a mnemonic with the other ending.

Read using name VBAR_EL1

The assembler syntax is:

```
MRS <Xt>, VBAR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return VBAR_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return VBAR_EL2<63:0>;
    else
        return VBAR_EL1<63:0>;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return VBAR_EL1<63:0>;
```

Write using name VBAR_EL1

The assembler syntax is:

```
MSR VBAR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        VBAR_EL2 = ZeroExtend(X[t]);
    else
        VBAR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL1 = ZeroExtend(X[t]);
```

Read using name VBAR_EL12

The assembler syntax is:

```
MRS <Xt>, VBAR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
```

```
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return VBAR_EL1<63:0>;
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            return VBAR_EL1<63:0>;
    else
        UNDEFINED;
```

Write using name VBAR_EL12

The assembler syntax is:

```
MSR VBAR_EL12, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x18);
            else
                AArch64.SystemAccessTrap(EL3, 0x18);
        else
            VBAR_EL1 = ZeroExtend(X[t]);
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x18);
        else
            VBAR_EL1 = ZeroExtend(X[t]);
    else
        UNDEFINED;
```

Read using name CVBAR_EL1

The assembler syntax is:

```
MRS <Ct>, CVBAR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x2A);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return VBAR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        return VBAR_EL2;
    else
        return VBAR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return VBAR_EL1;
```

Write using name CVBAR_EL1

The assembler syntax is:

```
MSR CVBAR_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
```

```
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x2A);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        VBAR_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        VBAR_EL2 = C[t];
    else
        VBAR_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        VBAR_EL1 = C[t];
```

Read using name CVBAR_EL12

The assembler syntax is:

```
MRS <Ct>, CVBAR_EL12
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x2A);
            else
                AArch64.SystemAccessTrap(EL3, 0x2A);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            return VBAR_EL1;
    else
```

```
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x2A);
        elsif CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            return VBAR_EL1;
    else
        UNDEFINED;
```

Write using name CVBAR_EL12

The assembler syntax is:

```
MSR CVBAR_EL12, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b101	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            if TargetELForCapabilityExceptions() == EL2 then
                AArch64.SystemAccessTrap(EL2, 0x2A);
            else
                AArch64.SystemAccessTrap(EL3, 0x2A);
        elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
            AArch64.SystemAccessTrap(EL2, 0x29);
        elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            VBAR_EL1 = C[t];
    else
        UNDEFINED;
elsif PSTATE.EL == EL3 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' then
        if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
            AArch64.SystemAccessTrap(EL3, 0x2A);
        elsif CPTR_EL3.EC == '0' then
            AArch64.SystemAccessTrap(EL3, 0x29);
        else
            VBAR_EL1 = C[t];
    else
        UNDEFINED;
```

3.2.49 VBAR_EL2, Vector Base Address Register (EL2)

The VBAR_EL2 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL2.

Attributes

VBAR_EL2 is a 129-bit register.

Configuration

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

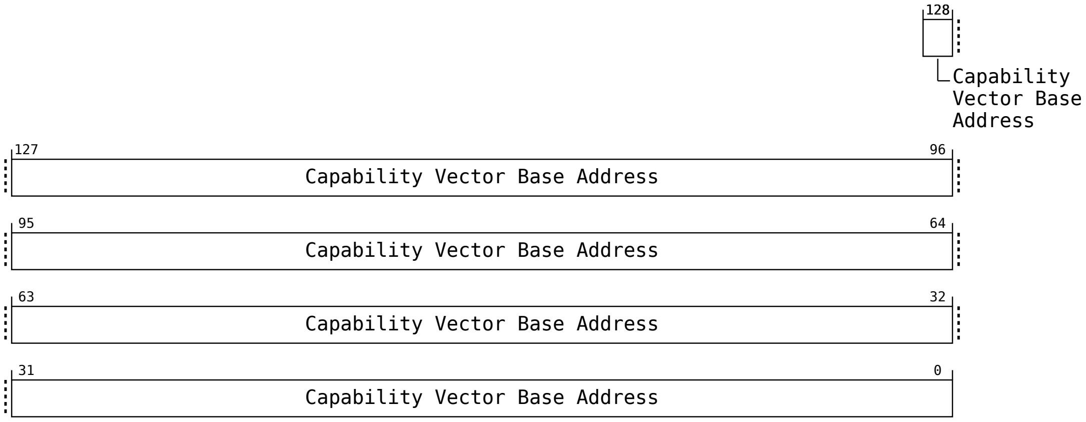
This register has no effect if EL2 is not enabled in the current Security state.

AArch64 System register VBAR_EL2[31:0] is architecturally mapped to AArch32 System register HVBAR[31:0].

Field descriptions

The VBAR_EL2 bit assignments are:

When Morello is implemented and Capability access at EL2 is not trapped:


128
Capability Vector Base Address
127
96
Capability Vector Base Address
95
64
Capability Vector Base Address
63
32
Capability Vector Base Address
31
0
Capability Vector Base Address

Bits [128:0]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

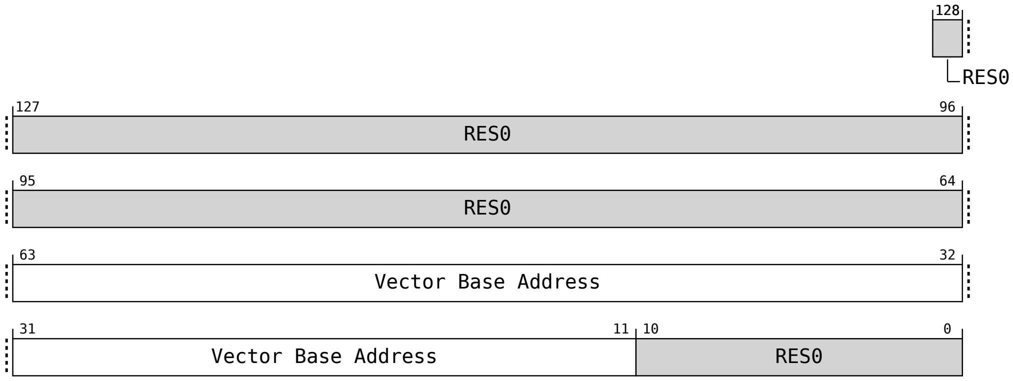
If the implementation supports xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

Bits [10:0] are treated as 0 for the purpose of calculating the exception vector address.

This field resets to an architecturally UNKNOWN value.

When Morello is implemented and Capability access at EL2 is trapped:



Bits [128:64]

Reserved, RES0.

Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports xARMv8.2-LVA, then:

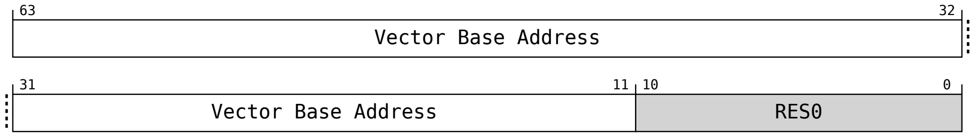
- If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

When Morello is not implemented:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL2.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

- If tagged addresses are not being used, bits [63:48] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL2 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL2

When HCR_EL2.E2H is 1, without explicit synchronization, access from EL2 using a mnemonic ending in _EL2 or _EL1 is not guaranteed to be ordered with respect to accesses using a mnemonic with the other ending.

Read using name VBAR_EL2

The assembler syntax is:

```
MRS <Xt>, VBAR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return VBAR_EL2<63:0>;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return VBAR_EL2<63:0>;
```

Write using name VBAR_EL2

The assembler syntax is:

```
MSR VBAR_EL2, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL2 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL2 = ZeroExtend(X[t]);
```

Read using name VBAR_EL1

The assembler syntax is:

```
MRS <Xt>, VBAR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return VBAR_EL1<63:0>;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        return VBAR_EL2<63:0>;
    else
        return VBAR_EL1<63:0>;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return VBAR_EL1<63:0>;
```

Write using name VBAR_EL1

The assembler syntax is:

```
MSR VBAR_EL1, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x18);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x18);
        else
            AArch64.SystemAccessTrap(EL3, 0x18);
    elsif HCR_EL2.E2H == '1' then
        VBAR_EL2 = ZeroExtend(X[t]);
    else
        VBAR_EL1 = ZeroExtend(X[t]);
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL1 = ZeroExtend(X[t]);
```

Read using name CVBAR_EL2

The assembler syntax is:

```
MRS <Ct>, CVBAR_EL2
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
```

```
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return VBAR_EL2;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return VBAR_EL2;
```

Write using name CVBAR_EL2

The assembler syntax is:

```
MSR CVBAR_EL2, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b100	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        VBAR_EL2 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        VBAR_EL2 = C[t];
```

Read using name CVBAR_EL1

The assembler syntax is:

```
MRS <Ct>, CVBAR_EL1
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x2A);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return VBAR_EL1;
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        return VBAR_EL2;
    else
        return VBAR_EL1;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return VBAR_EL1;
```

Write using name CVBAR_EL1

The assembler syntax is:

```
MSR CVBAR_EL1, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b000	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
```

```
        if TargetELForCapabilityExceptions() == EL1 then
            AArch64.SystemAccessTrap(EL1, 0x2A);
        elsif TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPACR_EL1.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL1, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H != '1' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        VBAR_EL1 = C[t];
elsif PSTATE.EL == EL2 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        if TargetELForCapabilityExceptions() == EL2 then
            AArch64.SystemAccessTrap(EL2, 0x2A);
        else
            AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0' then
        AArch64.SystemAccessTrap(EL2, 0x29);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    elsif HCR_EL2.E2H == '1' then
        VBAR_EL2 = C[t];
    else
        VBAR_EL1 = C[t];
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        VBAR_EL1 = C[t];
```

3.2.50 VBAR_EL3, Vector Base Address Register (EL3)

The VBAR_EL3 characteristics are:

Purpose

Holds the vector base address for any exception that is taken to EL3.

Attributes

VBAR_EL3 is a 129-bit register.

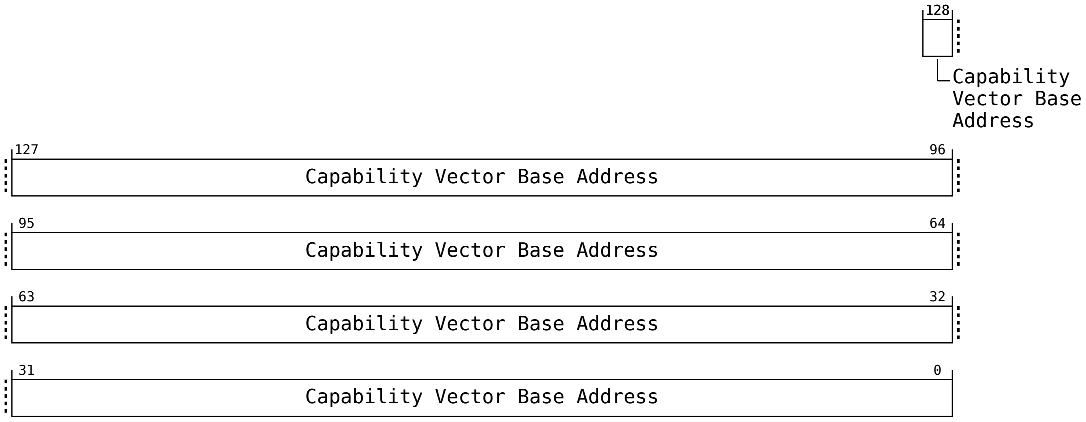
Configuration

This register is present only when HaveEL(EL3). Otherwise, direct accesses to VBAR_EL3 are UNDEFINED.

Field descriptions

The VBAR_EL3 bit assignments are:

When Morello is implemented and Capability access at EL3 is not trapped:


128
Capability Vector Base Address
127
96
Capability Vector Base Address
95
64
Capability Vector Base Address
63
32
Capability Vector Base Address
31
0
Capability Vector Base Address

Bits [128:0]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL3.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

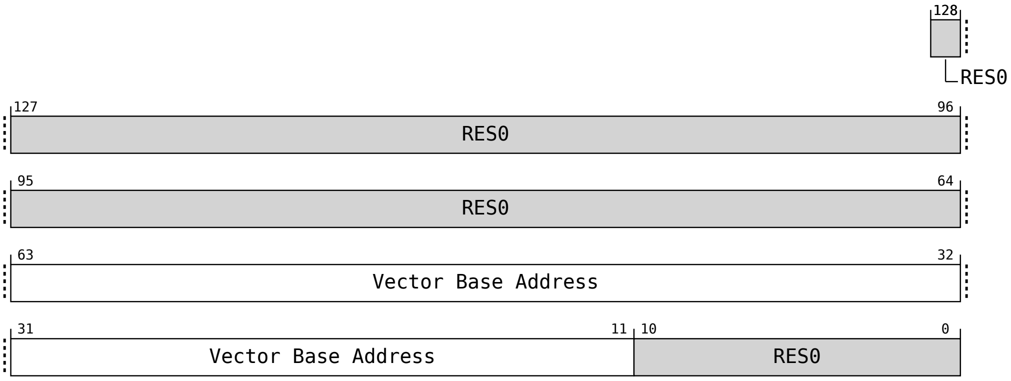
If the implementation supports xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

Bits [10:0] are treated as 0 for the purpose of calculating the exception vector address.

This field resets to an architecturally UNKNOWN value.

When Morello is implemented and Capability access at EL3 is trapped:



Bits [128:64]

Reserved, RES0.

Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL3.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:48] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports xARMv8.2-LVA, then:

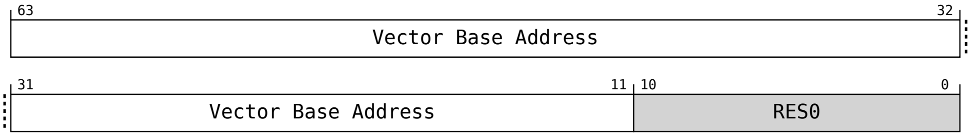
- If tagged addresses are being used, bits [55:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

When Morello is not implemented:



Bits [63:11]

Vector Base Address. Base address of the exception vectors for exceptions taken to EL3.

If the implementation does not support xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:48] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

- If tagged addresses are not being used, bits [63:48] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

If the implementation supports xARMv8.2-LVA, then:

- If tagged addresses are being used, bits [55:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.
- If tagged addresses are not being used, bits [63:52] of VBAR_EL3 must be the same or else the use of the vector address will result in a recursive exception.

This field resets to an architecturally UNKNOWN value.

Bits [10:0]

Reserved, RES0.

Accessing the VBAR_EL3

Read using name VBAR_EL3

The assembler syntax is:

```
MRS <Xt>, VBAR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        return VBAR_EL3<63:0>;
```

Write using name VBAR_EL3

The assembler syntax is:

```
MSR VBAR_EL3, <Xt>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
```

```
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x18);
    else
        VBAR_EL3 = ZeroExtend(X[t]);
```

Read using name CVBAR_EL3

The assembler syntax is:

```
MRS <Ct>, CVBAR_EL3
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        return VBAR_EL3;
```

Write using name CVBAR_EL3

The assembler syntax is:

```
MSR CVBAR_EL3, <Ct>
```

The encoding for this is in the System instruction encoding space:

op0	op1	CRn	CRm	op2
0b11	0b110	0b1100	0b0000	0b000

Accessibility:

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    if IsFeatureImplemented("Morello") && !CapIsSystemAccessEnabled() && !Halted() then
        AArch64.SystemAccessTrap(EL3, 0x2A);
    elsif CPTR_EL3.EC == '0' then
        AArch64.SystemAccessTrap(EL3, 0x29);
    else
        VBAR_EL3 = C[t];
```

Chapter 4
Instruction definitions

4.1 The instruction sets

I_JTQND This chapter describes the instructions available in the A64 and C64 instruction sets in the Morello architecture.

I_XJGLX This chapter contains:

- Instructions that are new in the Morello architecture.
- Instructions that are modified by the Morello architecture. Most of these instructions are changed by the addition of capability memory relocation checks.

Instructions that are not described in this chapter are not modified by the Morello architecture, and have the same behavior as described in the *Arm® Architecture Reference Manual, Armv8-A*.

An instruction is available in both A64 and C64, unless specified in the description. When reading these descriptions, the text at the start of each page provides a simple description of the instruction behavior. These descriptions are not updated to account for the differences in C64, but the rules of the specification and operation pseudocode cover these in detail.

The descriptions also include cross-references shown in italics. These are references to sections in the *Arm® Architecture Reference Manual, Armv8-A*, unless otherwise specified.

The assembler syntax indicates how the syntax differs in A64 and C64, for example:

```
ADR <Xd>, <label> //(PSTATE.C64 == '0')

ADR <Cd>, <label> //(PSTATE.C64 == '1')
```

The A64 syntax is described by the PSTATE.C64 == ‘0’ line, and the C64 syntax is described by the PSTATE.C64 == ‘1’ line.

Unless otherwise stated, when the syntax does not include discrimination, the syntax applies in both A64 and C64.

The Operation pseudocode shows the A64 and C64 behavior by switching on the value of `IsInC64()`.

I_{NZHVM} The letter C denotes a capability general-purpose register holding a capability.

CZR can be used in some instructions to represent a Capability where bits[128:0] are 0.

4.2 Modified base instructions

4.2.1 ADR

Form PCC-relative address adds an immediate value to the PCC value to form a PCC-relative address, and writes the result to the destination register.

31	30 29	28 24	23	22 5	4 0
0	immlo	1 0 0 0 0	P	immhi	Rd
op					

```
ADR     <Xd>, <label> //  (PSTATE.C64 == '0')
```

```
ADR     <Cd>, <label> //  (PSTATE.C64 == '1')
```

```
integer d = UInt(Rd);
bits(64) imm = SignExtend(P:immhi:immlo, 64);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Rd" field.

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

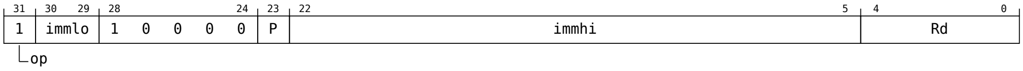
<label> Is the program label whose address is to be calculated, in the range +/-1MB, encoded in "P:immhi:immlo".

Operation

```
if IsInC64() then
    Capability addr = PCC[];

    C[d] = CapAdd(addr,imm);
else
    bits (64) addr;
    if CCTLR[].PCCBO == '1' then
        addr = CapGetOffset(PCC[]);
    else
        addr = CapGetValue(PCC[]);

    X[d] = addr + imm;
```

4.2.2 ADRP

Form PCC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits to the PCC value with the bottom 12 bits masked out to form a PCC-relative address and writes the result to the destination register. This description only applies in A64.



```
ADRP    <Xd>, <label>
```

```
integer d = UInt(Rd);
bits(64) imm;

if IsInC64() then
    if P == '1' then
        imm = SignExtend(immhi:immlo:Zeros(12), 64);
    else
        imm = ZeroExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(P:immhi:immlo:Zeros(12), 64);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

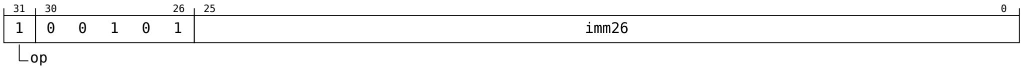
<label> Is the program label whose 4KB page address is to be calculated, in the range +/-4GB, encoded in "P:immhi:immlo".

Operation

```
if IsInC64() then
    Capability addr;
    if  P == '0' then
        if CCTLR[].ADRDPB == '1' then
            addr = C[28];
        else
            addr = DDC[];
    else
        addr = PCC[];

    bits(64) newvalue = CapGetValue(addr) AND NOT(ZeroExtend(Ones(12),64));
    bits(64) offset   = newvalue - CapGetValue(addr) + imm;

    Capability result = CapAdd(addr,offset);

    if CapIsSealed(addr) then
        result = CapWithTagClear(result);

    C[d] = result;
else
    bits(64) addr;
    if CCTLR[].PCCBO == '1' then
        addr = CapGetOffset(PCC[]);
    else
        addr = CapGetValue(PCC[]);

    addr<11:0> = Zeros(12);

    X[d] = addr + imm;
```

4.2.3 BL

Branch with Link branches to a PC-relative offset, setting the register X30 to PC+4. It provides a hint that this is a subroutine call.



```
BL  <label>
```

```
BranchType branch_type = if op == '1' then BranchType_DIRCALL else BranchType_DIR;
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler Symbols

<label> Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
if branch_type == BranchType_DIRCALL then
    if IsInC64() then
        if CCTLR[].SBL == '1' then
            C[30] = CapSetObjectType(CapAdd(PCC[], 5), CAP_SEAL_TYPE_RB);
        else
            C[30] = CapAdd(PCC[], 5);
    elsif CCTLR[].PCCBO == '1' then
        X[30] = PC[] + 4 - CapGetBase(PCC[]);
    else
        X[30] = PC[] + 4;

BranchToOffset(offset, branch_type);
```

4.2.4 BLR

Branch with Link to Register calls a subroutine at an address in a register, setting register X30 to PC+4.

31 … 25	24	23	22 21	20 … 16	15 … 12	11	10	9 … 5	4 … 0
1 1 0 1 0 1 1	0	0	0 1	1 1 1 1 1	0 0 0 0	0	0	Rn	0 0 0 0 0
	Z		op			A	M		Rm

```
BLR  <Xn>
```

```
integer n = UInt(Rn);
BranchType branch_type;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
Capability target;
if CCTLR[].PCCBO == '1' then
    target = CapSetOffset(PCC[], X[n]);
else
    target = CapSetValue(PCC[], X[n]);

if branch_type == BranchType_INDCALL then
    if IsInC64() then
        if CCTLR[].SBL == '1' then
            C[30] = CapSetObjectType(CapAdd(PCC[], 5), CAP_SEAL_TYPE_RB);
        else
            C[30] = CapAdd(PCC[], 5);
    elsif CCTLR[].PCCBO == '1' then
        X[30] = PC[] + 4 - CapGetBase(PCC[]);
    else
        X[30] = PC[] + 4;

BranchToCapability(target,branch_type);
```

4.2.5 BR

Branch to Register branches unconditionally to an address in a register, with a hint that this is not a subroutine return.

31 25	24	23	22 21	20 16	15 12	11	10	9 5	4 0
1 1 0 1 0 1 1	0	0	0 0	1 1 1 1 1	0 0 0 0	0	0	Rn	0 0 0 0 0
	Z		op			A	M		Rm

```
BR  <Xn>
```

```
integer n = UInt(Rn);
BranchType branch_type;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;
```

Assembler Symbols

<Xn> Is the 64-bit name of the general-purpose register holding the address to be branched to, encoded in the "Rn" field.

Operation

```
Capability target;
if CCTLR[].PCCBO == '1' then
    target = CapSetOffset(PCC[], X[n]);
else
    target = CapSetValue(PCC[], X[n]);

if branch_type == BranchType_INDCALL then
    if IsInC64() then
        if CCTLR[].SBL == '1' then
            C[30] = CapSetObjectType(CapAdd(PCC[], 5), CAP_SEAL_TYPE_RB);
        else
            C[30] = CapAdd(PCC[], 5);
    elsif CCTLR[].PCCBO == '1' then
        X[30] = PC[] + 4 - CapGetBase(PCC[]);
    else
        X[30] = PC[] + 4;

BranchToCapability(target,branch_type);
```

4.2.6 CAS, CASA, CASAL, CASL

Compare and Swap word or doubleword in memory reads a 32-bit word or 64-bit doubleword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, or <Xs>, is restored to the value held in the register before the instruction was executed.

No offset
(FEAT_LSE)

31	30	29					24	23	22	21	20 16	15	14				10	9 5	4 0
1	x	0	0	1	0	0	0	1	L	1	Rs	o0	1	1	1	1	1	Rn	Rt
size																			

32-bit CAS (size == 10 && L == 0 && o0 == 0)

```
CAS  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CAS  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

32-bit CASA (size == 10 && L == 1 && o0 == 0)

```
CASA  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CASA  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

32-bit CASAL (size == 10 && L == 1 && o0 == 1)

```
CASAL  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CASAL  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

32-bit CASL (size == 10 && L == 0 && o0 == 1)

```
CASL  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CASL  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CAS (size == 11 && L == 0 && o0 == 0)

```
CAS  <Xs>, <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CAS  <Xs>, <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CASA (size == 11 && L == 1 && o0 == 0)

```
CASA  <Xs>, <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CASA  <Xs>, <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CASAL (size == 11 && L == 1 && o0 == 1)

```
CASAL  <Xs>, <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
CASAL  <Xs>, <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CASL (size == 11 && L == 0 && o0 == 1)

```
CASL  <Xs>, <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASL  <Xs>, <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

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

Assembler Symbols

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

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

<Xs> Is the 64-bit name of the general-purpose register to be compared and loaded, encoded in the "Rs" field.

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

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

comparevalue = X[s];
newvalue = X[t];

VirtualAddress base = BaseReg[n];
data = MemAtomicCompareAndSwap(base, comparevalue, newvalue, ldacctype, stacctype);

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

4.2.7 CASB, CASAB, CASALB, CASLB

Compare and Swap byte in memory reads an 8-bit byte from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset
(FEAT_LSE)

31 30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
0 0	0 0 1 0 0 0	1	L	1	Rs	o0	1 1 1 1 1	Rn	Rt
size									

CASAB (L == 1 && o0 == 0)

```
CASAB  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASAB  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

CASALB (L == 1 && o0 == 1)

```
CASALB  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASALB  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

CASB (L == 0 && o0 == 0)

```
CASB  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASB  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

CASLB (L == 0 && o0 == 1)

```
CASLB  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASLB  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

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

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

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

comparevalue = X[s];
newvalue = X[t];

VirtualAddress base = BaseReg[n];
data = MemAtomicCompareAndSwap(base, comparevalue, newvalue, ldacctype, stacctype);

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

4.2.8 CASH, CASAH, CASALH, CASLH

Compare and Swap halfword in memory reads a 16-bit halfword from memory, and compares it against the value held in a first register. If the comparison is equal, the value in a second register is written to memory. If the write is performed, the read and write occur atomically such that no other modification of the memory location can take place between the read and write.

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the register which is compared and loaded, that is <Ws>, is restored to the values held in the register before the instruction was executed.

No offset
(FEAT_LSE)

31 30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	1	L	1	Rs	o0	1 1 1 1 1	Rn	Rt
size									

CASAH (L == 1 && o0 == 0)

```
CASAH  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASAH  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

CASALH (L == 1 && o0 == 1)

```
CASALH  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASALH  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

CASH (L == 0 && o0 == 0)

```
CASH  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASH  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

CASLH (L == 0 && o0 == 1)

```
CASLH  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASLH  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

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

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

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

comparevalue = X[s];
newvalue = X[t];

VirtualAddress base = BaseReg[n];
data = MemAtomicCompareAndSwap(base, comparevalue, newvalue, ldacctype, stacctype);

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

4.2.9 CASP, CASPA, CASPAL, CASPL

Compare and Swap Pair of words or doublewords in memory reads a pair of 32-bit words or 64-bit doublewords from memory, and compares them against the values held in the first pair of registers. If the comparison is equal, the values in the second pair of registers are written to memory. If the writes are performed, the reads and writes occur atomically such that no other modification of the memory location can take place between the reads and writes.

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

The architecture permits that the data read clears any exclusive monitors associated with that location, even if the compare subsequently fails.

If the instruction generates a synchronous Data Abort, the registers which are compared and loaded, that is <Ws> and <W(s+1)>, or <Xs> and <X(s+1)>, are restored to the values held in the registers before the instruction was executed.

No offset
(FEAT_LSE)

31	30	29					24	23	22	21	20 - 16	15	14				10	9 - 5	4 - 0
0	sz	0	0	1	0	0	0	0	L	1	Rs	o0	1	1	1	1	1	Rn	Rt
													Rt2						

32-bit CASP (sz == 0 && L == 0 && o0 == 0)

```
CASP  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASP  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

32-bit CASPA (sz == 0 && L == 1 && o0 == 0)

```
CASPA  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASPA  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

32-bit CASPAL (sz == 0 && L == 1 && o0 == 1)

```
CASPAL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASPAL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

32-bit CASPL (sz == 0 && L == 0 && o0 == 1)

```
CASPL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASPL  <Ws>, <W(s+1)>, <Wt>, <W(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CASP (sz == 1 && L == 0 && o0 == 0)

```
CASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASP  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CASPA (sz == 1 && L == 1 && o0 == 0)

```
CASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
CASPA  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CASPAL (sz == 1 && L == 1 && o0 == 1)

```
CASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CASPAL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit CASPL (sz == 1 && L == 0 && o0 == 1)

```
CASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

CASPL  <Xs>, <X(s+1)>, <Xt>, <X(t+1)>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

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

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

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

Assembler Symbols

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

<W(s+1)> Is the 32-bit name of the second general-purpose register to be compared and loaded.

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

<W(t+1)> Is the 32-bit name of the second general-purpose register to be conditionally stored.

<Xs> Is the 64-bit name of the first general-purpose register to be compared and loaded, encoded in the "Rs" field. <Xs> must be an even-numbered register.

<X(s+1)> Is the 64-bit name of the second general-purpose register to be compared and loaded.

<Xt> Is the 64-bit name of the first general-purpose register to be conditionally stored, encoded in the "Rt" field. <Xt> must be an even-numbered register.

<X(t+1)> Is the 64-bit name of the second general-purpose register to be conditionally stored.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

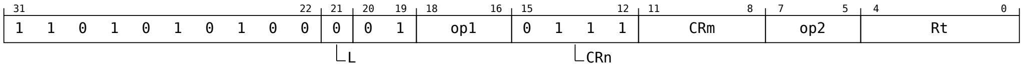
```
bits(2*datasize) comparevalue;
bits(2*datasize) newvalue;
bits(2*datasize) data;

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

VirtualAddress base = BaseReg[n];
data = MemAtomicCompareAndSwap(base, comparevalue, newvalue, ldacctype, stacctype);

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

4.2.10 DC

Data Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of SYS. This means:

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



```
DC  <dc_op>, <Xt> //  (PSTATE.C64 == '0' or when <dc_op> does not take a VA)
```

```
DC  <dc_op>, <Ct> //  (PSTATE.C64 == '1' when <dc_op> takes a VA)
```

is equivalent to

```
SYS#<op1>, C7, <Cm>, #<op2>, <Xt>
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_DC`.

Assembler Symbols

<dc_op> Is a DC instruction name, as listed for the DC system instruction group, encoded in"op1:CRm:op2":

op1	CRm	op2	<dc_op>	Architectural Feature
000	0110	001	IVAC	-
000	0110	010	ISW	-
000	1010	010	CSW	-
000	1110	010	CISW	-
011	0100	001	ZVA	-
011	1010	001	CVAC	-
011	1011	001	CVAU	-
011	1100	001	CVAP	FEAT_DPB
011	1101	001	CVADP	FEAT_DPB2
011	1110	001	CIVAC	-

<Ct> Is the source capability register, encoded in the "Rt" field.

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

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

Operation

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

4.2.11 ERET

Exception Return using the ELR and SPSR for the current Exception level. When executed, the PE restores *PSTATE* from the SPSR, and branches to the address held in the ELR.

The PE checks the SPSR for the current Exception level for an illegal return event. See *Illegal return events from AArch64 state*.

ERET is UNDEFINED at EL0.

31 25	24	23 21	20 16	15 12	11	10	9 5	4 0
1 1 0 1 0 1 1	0	1 0 0	1 1 1 1 1	0 0 0 0	0	0	1 1 1 1 1	0 0 0 0 0
					A	M	Rn	op4

```
ERET
```

```
if PSTATE.EL == EL0 then UNDEFINED;
```

Operation

```
Capability target;
if IsAccessToCapabilitiesEnabledAtEL(PSTATE.EL) then
    target = CELR[];
else
    target = CapSetValue(PCC[], ELR[]);

AArch64.ExceptionReturnToCapability(target, SPSR[]);
```

4.2.12 IC

Instruction Cache operation. For more information, see *op0==0b01, cache maintenance, TLB maintenance, and address translation instructions*.

This is an alias of SYS. This means:

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

31 22	21	20	19	18 16	15 12	11 8	7 5	4 0
1 1 0 1 0 1 0 1 0 0	0	0	1	op1	0 1 1 1	CRm	op2	Rt
	L				CRn			

```
IC  <ic_op>{, <Xt>} //  (PSTATE.C64 == '0' or when <ic_op> does not take a VA)
```

```
IC  <ic_op>{, <Ct>} //  (PSTATE.C64 == '1' when <ic_op> takes a VA)
```

is equivalent to

```
SYS#<op1>, C7, <Cm>, #<op2>{, <Xt>}
```

and is the preferred disassembly when `SysOp(op1,'0111',CRm,op2) == Sys_IC`.

Assembler Symbols

<ic_op> Is an IC instruction name, as listed for the IC system instruction pages, encoded in"op1:CRm:op2":

op1	CRm	op2	<ic_op>
000	0001	000	IALLUIS
000	0101	000	IALLU
011	0101	001	IVAU

<Ct> Is the optional source capability register, defaulting to '11111', encoded in the "Rt" field.

<op1> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op1" field.

<Cm> Is a name 'Cm', with 'm' in the range 0 to 15, encoded in the "CRm" field.

<op2> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "op2" field.

<Xt> Is the 64-bit name of the optional general-purpose source register, defaulting to '11111', encoded in the "Rt" field.

Operation

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

4.2.13 LDADD, LDADDA, LDADDAL, LDADDL

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

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STADD, STADDL.

Integer
(FEAT_LSE)

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

32-bit LDADD (size == 10 && A == 0 && R == 0)

```
LDADD  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADD  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDADDA (size == 10 && A == 1 && R == 0)

```
LDADDA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDADDAL (size == 10 && A == 1 && R == 1)

```
LDADDAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDADDL (size == 10 && A == 0 && R == 1)

```
LDADDL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDADD (size == 11 && A == 0 && R == 0)

```
LDADD  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADD  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDADDA (size == 11 && A == 1 && R == 0)

```
LDADDA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDADDAL (size == 11 && A == 1 && R == 1)

```
LDADDAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDADDL (size == 11 && A == 0 && R == 1)

```
LDADDL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDADDL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STADD, STADDL	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.14 LDADDB, LDADDAB, LDADDALB, LDADDLB

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

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STADDB, STADDLB.

Integer

(FEAT_LSE)

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

LDADDAB (A == 1 && R == 0)

```
LDADDAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDADDALB (A == 1 && R == 1)

```
LDADDALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDADDB (A == 0 && R == 0)

```
LDADDB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDADDLB (A == 0 && R == 1)

```
LDADDLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STADDB, STADDLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.15 LDADDH, LDADDAH, LDADDALH, LDADDLH

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

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STADDH, STADDLH.

Integer

(FEAT_LSE)

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

LDADDAH (A == 1 && R == 0)

```
LDADDAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDADDALH (A == 1 && R == 1)

```
LDADDALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDADDH (A == 0 && R == 0)

```
LDADDH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDADDLH (A == 0 && R == 1)

```
LDADDLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDADDLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STADDH, STADDLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.16 LDAPR

Load-Acquire RCpc Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from the derived address in memory, and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer
(FEAT_LRCPC)

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

32-bit (size == 10)

```
LDAPR  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAPR  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDAPR  <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAPR  <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
VACheckAddress(base, address, dbytes, CAP_PERM_LOAD, acctype);

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

4.2.17 LDAPRB

Load-Acquire RCpc Register Byte derives an address from a base register value, loads a byte from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer
(FEAT_LRCPC)

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

```
LDAPRB  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAPRB  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
VACheckAddress(base, address, dbytes, CAP_PERM_LOAD, acctype);

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

4.2.18 LDAPRH

Load-Acquire RCpc Register Halfword derives an address from a base register value, loads a halfword from the derived address in memory, zero-extends it and writes it to a register.

The instruction has memory ordering semantics as described in *Load-Acquire, Load-AcquirePC, and Store-Release*, except that:

- There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction.
- The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode.

For information about memory accesses, see *Load/Store addressing modes*.

Integer
(FEAT_LRCPC)

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

```
LDAPRH  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAPRH  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = AccType_ORDERED;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
VACheckAddress(base, address, dbytes, CAP_PERM_LOAD, acctype);

data = Mem[address, dbytes, acctype];
X[t] = ZeroExtend(data, regsize);
```

4.2.19 LDAR

Load-Acquire Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0 0 1 0 0 0	1	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

32-bit (size == 10)

```
LDAR  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

LDAR  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDAR  <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

LDAR  <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.20 LDARB

Load-Acquire Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29 … 24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0 0 1 0 0 0	1	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

```
LDARB  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDARB  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.21 LDARH

Load-Acquire Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

31	30	29					24	23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	1	0	0	0	1	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size									L		Rs	o0	Rt2		

```
LDARH  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDARH  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

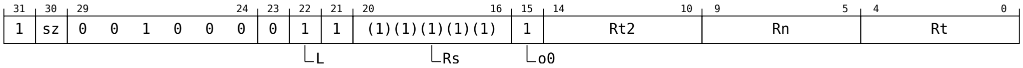
<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.22 LDAXP

Load-Acquire Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (sz == 0)

```
LDAXP  <Wt1>, <Wt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAXP  <Wt1>, <Wt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (sz == 1)

```
LDAXP  <Xt1>, <Xt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAXP  <Xt1>, <Xt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDAXP*.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
```

```

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.23 LDAXR

Load-Acquire Exclusive Register derives an address from a base register value, loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0 0 1 0 0 0	0	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

32-bit (size == 10)

```
LDAXR  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAXR  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDAXR  <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAXR  <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.24 LDAXRB

Load-Acquire Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31 30	29 … 24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 0	0 0 1 0 0 0	0	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

```
LDAXRB  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAXRB  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
```

```
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.25 LDAXRH

Load-Acquire Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31 30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
0 1	0 0 1 0 0 0	0	1	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L		Rs	o0	Rt2		

```
LDAXRH  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDAXRH  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

```

```
VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.26 LDCLR, LDCLRA, LDCLRAL, LDCLRL

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

- If the destination register is not one of WZR or XZR, `LDCLRA` and `LDCLRAL` load from memory with acquire semantics.
- `LDCLRL` and `LDCLRAL` store to memory with release semantics.
- `LDCLR` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STCLR, STCLRL.

Integer
(FEAT_LSE)

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

32-bit LDCLR (size == 10 && A == 0 && R == 0)

```
LDCLR  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLR  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDCLRA (size == 10 && A == 1 && R == 0)

```
LDCLRA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLRA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDCLRAL (size == 10 && A == 1 && R == 1)

```
LDCLRAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLRAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDCLRL (size == 10 && A == 0 && R == 1)

```
LDCLRL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLRL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDCLR (size == 11 && A == 0 && R == 0)

```
LDCLR  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLR  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDCLRA (size == 11 && A == 1 && R == 0)

```
LDCLRA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLRA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDCLRAL (size == 11 && A == 1 && R == 1)

```
LDCLRAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLRAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDCLRL (size == 11 && A == 0 && R == 1)

```
LDCLRL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCLRL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STCLR, STCLRL	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.27 LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB

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

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

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STCLRB, STCLRLB.

Integer

(FEAT_LSE)

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

LDCLRAB (A == 1 && R == 0)

```
LDCLRAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDCLRALB (A == 1 && R == 1)

```
LDCLRALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDCLRB (A == 0 && R == 0)

```
LDCLRB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDCLRLB (A == 0 && R == 1)

```
LDCLRLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STCLRB, STCLRLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.28 LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH

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

- If the destination register is not WZR, `LDCLRAH` and `LDCLRALH` load from memory with acquire semantics.
- `LDCLRLH` and `LDCLRALH` store to memory with release semantics.
- `LDCLRH` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STCLRH, STCLRLH.

Integer

(FEAT_LSE)

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

LDCLRAH (A == 1 && R == 0)

```
LDCLRAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDCLRALH (A == 1 && R == 1)

```
LDCLRALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDCLRH (A == 0 && R == 0)

```
LDCLRH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDCLRLH (A == 0 && R == 1)

```
LDCLRLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDCLRLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STCLRH, STCLRLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.29 LDEOR, LDEORA, LDEORAL, LDEORL

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

- If the destination register is not one of WZR or XZR, `LDEORA` and `LDEORAL` load from memory with acquire semantics.
- `LDEORL` and `LDEORAL` store to memory with release semantics.
- `LDEOR` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STEOR, STEORL.

Integer
(FEAT_LSE)

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

32-bit LDEOR (size == 10 && A == 0 && R == 0)

```
LDEOR  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEOR  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDEORA (size == 10 && A == 1 && R == 0)

```
LDEORA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEORA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDEORAL (size == 10 && A == 1 && R == 1)

```
LDEORAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEORAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDEORL (size == 10 && A == 0 && R == 1)

```
LDEORL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEORL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDEOR (size == 11 && A == 0 && R == 0)

```
LDEOR  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEOR  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDEORA (size == 11 && A == 1 && R == 0)

```
LDEORA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEORA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDEORAL (size == 11 && A == 1 && R == 1)

```
LDEORAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEORAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDEORL (size == 11 && A == 0 && R == 1)

```
LDEORL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDEORL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STEOR, STEORL	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.30 LDEORB, LDEORAB, LDEORALB, LDEORLB

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

- If the destination register is not WZR, LDEORAB and LDEORALB load from memory with acquire semantics.
- LDEORLB and LDEORALB store to memory with release semantics.
- LDEORB has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STEORB, STEORLB.

Integer

(FEAT_LSE)

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

LDEORAB (A == 1 && R == 0)

```
LDEORAB <Ws>, <Wt>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
LDEORAB <Ws>, <Wt>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

LDEORALB (A == 1 && R == 1)

```
LDEORALB <Ws>, <Wt>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
LDEORALB <Ws>, <Wt>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

LDEORB (A == 0 && R == 0)

```
LDEORB <Ws>, <Wt>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
LDEORB <Ws>, <Wt>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

LDEORLB (A == 0 && R == 1)

```
LDEORLB <Ws>, <Wt>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
LDEORLB <Ws>, <Wt>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STEORB, STEORLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.31 LDEORH, LDEORAH, LDEORALH, LDEORLH

Atomic exclusive OR on halfword in memory atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDEORAH` and `LDEORALH` load from memory with acquire semantics.
- `LDEORLH` and `LDEORALH` store to memory with release semantics.
- `LDEORH` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STEORH, STEORLH.

Integer
(FEAT_LSE)

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

LDEORAH (A == 1 && R == 0)

```
LDEORAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDEORAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDEORALH (A == 1 && R == 1)

```
LDEORALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDEORALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDEORH (A == 0 && R == 0)

```
LDEORH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDEORH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDEORLH (A == 0 && R == 1)

```
LDEORLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDEORLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STEORH, STEORLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.32 LDLAR

Load LOAcquire Register loads a 32-bit word or 64-bit doubleword from memory, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0 0 1 0 0 0	1	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

32-bit (size == 10)

```
LDLAR  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

LDLAR  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDLAR  <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

LDLAR  <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.33 LDLARB

Load LOAcquire Register Byte loads a byte from memory, zero-extends it and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset

(FEAT_LOR)

31	30	29 … 24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	0 0 1 0 0 0	1	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

```
LDLARB  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDLARB  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.34 LDLARH

Load LOAcquire Register Halfword loads a halfword from memory, zero-extends it, and writes it to a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

For this instruction, if the destination is WZR/ZXR, it is impossible for software to observe the presence of the acquire semantic other than its effect on the arrival at endpoints.

No offset
(FEAT_LOR)

31	30	29					24	23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	1	0	0	0	1	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size									L		Rs	o0	Rt2		

```
LDLARH  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDLARH  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

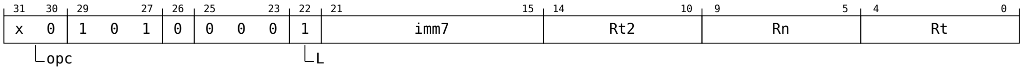
Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.35 LDNP

Load Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers.

For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.



32-bit (opc == 00)

```
LDNP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')

LDNP  <Wt1>, <Wt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')

LDNP  <Xt1>, <Xt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP*.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        X[t]  = data1;
        X[t2] = data2;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.36 LDP

Load Pair of Registers calculates an address from a base register value and an immediate offset, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	0	1	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDP  <Wt1>, <Wt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDP  <Xt1>, <Xt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	1	1	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDP  <Wt1>, <Wt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDP  <Xt1>, <Xt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	1	0	1	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
LDP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDP  <Wt1>, <Wt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDP  <Xt1>, <Xt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP*.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;         // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;     // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        if signed then
            X[t]  = SignExtend(data1, 64);
            X[t2] = SignExtend(data2, 64);
        else
            X[t]  = data1;
            X[t2] = data2;

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.37 LDPSW

Load Pair of Registers Signed Word calculates an address from a base register value and an immediate offset, loads two 32-bit words from memory, sign-extends them, and writes them to two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
0	1	1	0	1	0	0	0	1	1	imm7	Rt2	Rn	Rt
	opc								L				

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDPSW  <Xt1>, <Xt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
0	1	1	0	1	0	0	1	1	1	imm7	Rt2	Rn	Rt
	opc								L				

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDPSW  <Xt1>, <Xt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
0	1	1	0	1	0	0	1	0	1	imm7	Rt2	Rn	Rt
	opc								L				

```
LDPSW  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDPSW  <Xt1>, <Xt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDPSW*.

Assembler Symbols

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the post-index and pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;         // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;     // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        if signed then
            X[t]  = SignExtend(data1, 64);
            X[t2] = SignExtend(data2, 64);
        else
            X[t]  = data1;
            X[t2] = data2;

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.38 LDR (immediate)

Load Register (immediate) loads a word or doubleword from memory and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*. The Unsigned offset variant scales the immediate offset value by the size of the value accessed before adding it to the base register value.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
LDR  <Wt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDR  <Wt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDR  <Xt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDR  <Xt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
LDR  <Wt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDR  <Wt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDR  <Xt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDR  <Xt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 10	9 5	4 0
1	x	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

32-bit (size == 10)

```
LDR  <Wt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDR  <Wt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDR  <Xt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
LDR  <Xt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDR (immediate)*.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.39 LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23 5	4 0
0	x	0	1	1	0	0	0	imm19	Rt
opc									

32-bit (opc == 00)

```
LDR  <Wt>, <label>
```

64-bit (opc == 01)

```
LDR  <Xt>, <label>
```

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

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

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

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
VirtualAddress base = VAFromCapability(PCC);
bits(64) address = VAddress(base) + offset;

bits(size*8) data;

case memop of
    when MemOp_LOAD
        VACheckAddress(base, address, size, CAP_PERM_LOAD, AccType_NORMAL);
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

4.2.40 LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	0	1	1	Rm	option	S	1	0	Rn	Rt
└size								└opc									

32-bit (size == 10)

```
LDR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '0')

LDR <Wt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '0')

LDR <Xt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;             // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
```

```
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.41 LDRB (immediate)

Load Register Byte (immediate) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size								opc							

```
LDRB  <Wt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDRB  <Wt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size								opc							

```
LDRB  <Wt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDRB  <Wt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 10	9 5	4 0
0	0	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

```
LDRB  <Wt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDRB  <Wt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRH (immediate)*.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address,

encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
```

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

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.42 LDRB (register)

Load Register Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	1	1	Rm	option	S	1	0	Rn	Rt
size								opc									

Extended register (option != 011)

```
LDRB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '0')

LDRB  <Wt>, [<Cn|CSP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '1')
```

Shifted register (option == 011)

```
LDRB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '0')

LDRB  <Wt>, [<Cn|CSP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in"option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
```

```
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.43 LDRH (immediate)

Load Register Halfword (immediate) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	0	1	Rn	Rt
size								opc							

```
LDRH  <Wt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDRH  <Wt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	1	0	imm9	1	1	Rn	Rt
size								opc							

```
LDRH  <Wt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDRH  <Wt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	1	1	1	1	0	0	1	0	1	imm12	Rn	Rt
size								opc				

```
LDRH  <Wt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDRH  <Wt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRH (immediate)*.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address,

encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
```

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

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.44 LDRH (register)

Load Register Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	0 1	1	Rm	option	S	1 0	Rn	Rt
size				opc							

```
LDRH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
LDRH  <Wt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;            // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
```

```
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.45 LDRSB (immediate)

Load Register Signed Byte (immediate) loads a byte from memory, sign-extends it to either 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	1	x	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDRSB  <Wt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')

LDRSB  <Wt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDRSB  <Xt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')

LDRSB  <Xt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	1	x	0	imm9	1	1	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDRSB  <Wt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')

LDRSB  <Wt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDRSB  <Xt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')

LDRSB  <Xt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 10	9 5	4 0
0	0	1	1	1	0	0	1	1	x	imm12	Rn	Rt
size								opc				

32-bit (opc == 11)

```
LDRSB  <Wt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')

LDRSB  <Wt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDRSB  <Xt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')

LDRSB  <Xt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSB (immediate)*.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

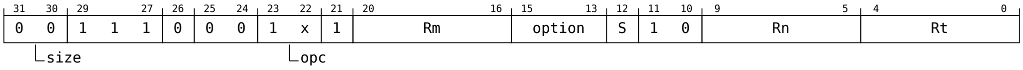
Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

```

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.46 LDRSB (register)

Load Register Signed Byte (register) calculates an address from a base register value and an offset register value, loads a byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit with extended register offset (opc == 11 && option != 011)

```
LDRSB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '0')

LDRSB  <Wt>, [<Cn|CSP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '1')
```

32-bit with shifted register offset (opc == 11 && option == 011)

```
LDRSB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '0')

LDRSB  <Wt>, [<Cn|CSP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '1')
```

64-bit with extended register offset (opc == 10 && option != 011)

```
LDRSB  <Xt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '0')

LDRSB  <Xt>, [<Cn|CSP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '1')
```

64-bit with shifted register offset (opc == 10 && option == 011)

```
LDRSB  <Xt>, [<Xn|SP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '0')

LDRSB  <Xt>, [<Cn|CSP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in"option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.47 LDRSH (immediate)

Load Register Signed Halfword (immediate) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	1	x	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDRSH  <Wt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDRSH  <Wt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDRSH  <Xt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDRSH  <Xt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	1	x	0	imm9	1	1	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDRSH  <Wt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDRSH  <Wt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDRSH  <Xt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDRSH  <Xt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 10	9 5	4 0
0	1	1	1	1	0	0	1	1	x	imm12	Rn	Rt
size								opc				

32-bit (opc == 11)

```
LDRSH  <Wt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDRSH  <Wt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDRSH  <Xt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDRSH  <Xt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSH (immediate)*.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

```

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.48 LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31 30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	1	Rm	option	S	1 0	Rn	Rt
size				opc							

32-bit (opc == 11)

```
LDRSH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')

LDRSH  <Wt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDRSH  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')

LDRSH  <Xt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
```

```
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
```

```
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.49 LDRSW (immediate)

Load Register Signed Word (immediate) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	0	1	1	1	0	0	0	1	0	0	imm9	0	1	Rn	Rt
└size								└opc							

```
LDRSW  <Xt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDRSW  <Xt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	0	1	1	1	0	0	0	1	0	0	imm9	1	1	Rn	Rt
└size								└opc							

```
LDRSW  <Xt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDRSW  <Xt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
1	0	1	1	1	0	0	1	1	0	imm12	Rn	Rt
└size								└opc				

```
LDRSW  <Xt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDRSW  <Xt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDRSW (immediate)*.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address,

encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

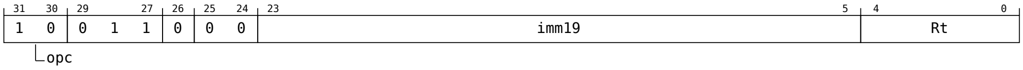
Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
```

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

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.50 LDRSW (literal)

Load Register Signed Word (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



```
LDRSW  <Xt>, <label>
```

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

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

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
VirtualAddress base = VAFromCapability(PCC);
bits(64) address = VAddress(base) + offset;

bits(size*8) data;

case memop of
    when MemOp_LOAD
        VACheckAddress(base, address, size, CAP_PERM_LOAD, AccType_NORMAL);
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

4.2.51 LDRSW (register)

Load Register Signed Word (register) calculates an address from a base register value and an offset register value, loads a word from memory, sign-extends it to form a 64-bit value, and writes it to a register. The offset register value can be shifted left by 0 or 2 bits. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 … 16	15 … 13	12	11	10	9 … 5	4 … 0
1	0	1	1	1	0	0	0	1	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

```
LDRSW  <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
LDRSW  <Xt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#2

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.52 LDSET, LDSETA, LDSETAL, LDSETL

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

- If the destination register is not one of WZR or XZR, `LDSETA` and `LDSETAL` load from memory with acquire semantics.
- `LDSETL` and `LDSETAL` store to memory with release semantics.
- `LDSET` has no memory ordering requirements.

For more information about memory ordering semantics see *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This instruction is used by the alias STSET, STSETL.

Integer
(FEAT_LSE)

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

32-bit LDSET (size == 10 && A == 0 && R == 0)

```
LDSET  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSET  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSETA (size == 10 && A == 1 && R == 0)

```
LDSETA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSETA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSETAL (size == 10 && A == 1 && R == 1)

```
LDSETAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSETAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSETL (size == 10 && A == 0 && R == 1)

```
LDSETL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSETL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSET (size == 11 && A == 0 && R == 0)

```
LDSET  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSET  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSETA (size == 11 && A == 1 && R == 0)

```
LDSETA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSETA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSETAL (size == 11 && A == 1 && R == 1)

```
LDSETAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSETAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSETL (size == 11 && A == 0 && R == 1)

```
LDSETL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSETL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSET, STSETL	A == '0' && Rt == '11111'

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.53 LDSETB, LDSETAB, LDSETALB, LDSETLB

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

- If the destination register is not WZR, `LDSETAB` and `LDSETALB` load from memory with acquire semantics.
- `LDSETLB` and `LDSETALB` store to memory with release semantics.
- `LDSETB` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSETB, STSETLB.

Integer

(FEAT_LSE)

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

LDSETAB (A == 1 && R == 0)

```
LDSETAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSETALB (A == 1 && R == 1)

```
LDSETALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSETB (A == 0 && R == 0)

```
LDSETB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSETLB (A == 0 && R == 1)

```
LDSETLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSETB, STSETLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.54 LDSETH, LDSETAH, LDSETALH, LDSETLH

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

- If the destination register is not WZR, LDSETAH and LDSETALH load from memory with acquire semantics.
- LDSETLH and LDSETALH store to memory with release semantics.
- LDSETH has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSETH, STSETLH.

Integer
(FEAT_LSE)

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

LDSETAH (A == 1 && R == 0)

```
LDSETAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSETALH (A == 1 && R == 1)

```
LDSETALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSETH (A == 0 && R == 0)

```
LDSETH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSETLH (A == 0 && R == 1)

```
LDSETLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSETLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSETH, STSETLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.55 LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL

Atomic signed maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSMAXA` and `LDSMAXAL` load from memory with acquire semantics.
- `LDSMAXL` and `LDSMAXAL` store to memory with release semantics.
- `LDSMAX` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSMAX, STSMAXL.

Integer
(FEAT_LSE)

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

32-bit LDSMAX (size == 10 && A == 0 && R == 0)

```
LDSMAX  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAX  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSMAXA (size == 10 && A == 1 && R == 0)

```
LDSMAXA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAXA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSMAXAL (size == 10 && A == 1 && R == 1)

```
LDSMAXAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAXAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSMAXL (size == 10 && A == 0 && R == 1)

```
LDSMAXL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAXL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMAX (size == 11 && A == 0 && R == 0)

```
LDSMAX  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAX  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMAXA (size == 11 && A == 1 && R == 0)

```
LDSMAXA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAXA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMAXAL (size == 11 && A == 1 && R == 1)

```
LDSMAXAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAXAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMAXL (size == 11 && A == 0 && R == 1)

```
LDSMAXL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMAXL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMAX, STSMAXL	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.56 LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB

Atomic signed maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMAXAB and LDSMAXALB load from memory with acquire semantics.
- LDSMAXLB and LDSMAXALB store to memory with release semantics.
- LDSMAXB has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSMAXB, STSMAXLB.

Integer

(FEAT_LSE)

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

LDSMAXAB (A == 1 && R == 0)

```
LDSMAXAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMAXALB (A == 1 && R == 1)

```
LDSMAXALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMAXB (A == 0 && R == 0)

```
LDSMAXB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMAXLB (A == 0 && R == 1)

```
LDSMAXLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMAXB, STSMAXLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.57 LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH

Atomic signed maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSMAXAH` and `LDSMAXALH` load from memory with acquire semantics.
- `LDSMAXLH` and `LDSMAXALH` store to memory with release semantics.
- `LDSMAXH` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSMAXH, STSMAXLH.

Integer

(FEAT_LSE)

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

LDSMAXAH (A == 1 && R == 0)

```
LDSMAXAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMAXALH (A == 1 && R == 1)

```
LDSMAXALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMAXH (A == 0 && R == 0)

```
LDSMAXH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMAXLH (A == 0 && R == 1)

```
LDSMAXLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMAXLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMAXH, STSMAXLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.58 LDSMIN, LDSMINA, LDSMINAL, LDSMINL

Atomic signed minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDSMINA` and `LDSMINAL` load from memory with acquire semantics.
- `LDSMINL` and `LDSMINAL` store to memory with release semantics.
- `LDSMIN` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSMIN, STSMINL.

Integer
(FEAT_LSE)

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

32-bit LDSMIN (size == 10 && A == 0 && R == 0)

```
LDSMIN  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMIN  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSMINA (size == 10 && A == 1 && R == 0)

```
LDSMINA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSMINAL (size == 10 && A == 1 && R == 1)

```
LDSMINAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDSMINL (size == 10 && A == 0 && R == 1)

```
LDSMINL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMIN (size == 11 && A == 0 && R == 0)

```
LDSMIN  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMIN  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMINA (size == 11 && A == 1 && R == 0)

```
LDSMINA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMINAL (size == 11 && A == 1 && R == 1)

```
LDSMINAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDSMINL (size == 11 && A == 0 && R == 1)

```
LDSMINL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMINL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMIN, STSMINL	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.59 LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB

Atomic signed minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDSMINAB and LDSMINALB load from memory with acquire semantics.
- LDSMINLB and LDSMINALB store to memory with release semantics.
- LDSMINB has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSMINB, STSMINLB.

Integer

(FEAT_LSE)

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

LDSMINAB (A == 1 && R == 0)

```
LDSMINAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMINAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMINALB (A == 1 && R == 1)

```
LDSMINALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMINALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMINB (A == 0 && R == 0)

```
LDSMINB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMINB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMINLB (A == 0 && R == 1)

```
LDSMINLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDSMINLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMINB, STSMINLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.60 LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH

Atomic signed minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDSMINAH` and `LDSMINALH` load from memory with acquire semantics.
- `LDSMINLH` and `LDSMINALH` store to memory with release semantics.
- `LDSMINH` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STSMINH, STSMINLH.

Integer
(FEAT_LSE)

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

LDSMINAH (A == 1 && R == 0)

```
LDSMINAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMINALH (A == 1 && R == 1)

```
LDSMINALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMINH (A == 0 && R == 0)

```
LDSMINH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDSMINLH (A == 0 && R == 1)

```
LDSMINLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDSMINLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STSMINH, STSMINLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.61 LDTR

Load Register (unprivileged) loads a word or doubleword from memory, and writes it to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
1 x	1 1 1	0	0 0	0 1	0	imm9	1 0	Rn	Rt
size				opc					

32-bit (size == 10)

```
LDTR  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDTR  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDTR  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDTR  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.62 LDTRB

Load Register Byte (unprivileged) loads a byte from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size								opc							

```
LDTRB  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
LDTRB  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.63 LDTRH

Load Register Halfword (unprivileged) loads a halfword from memory, zero-extends it, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	0	1	0	imm9	1	0	Rn	Rt
size								opc							

```
LDTRH  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
LDTRH  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.64 LDTRSB

Load Register Signed Byte (unprivileged) loads a byte from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	1	x	0	imm9	1	0	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDTRSB  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDTRSB  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDTRSB  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDTRSB  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
```

```
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.65 LDTRSH

Load Register Signed Halfword (unprivileged) loads a halfword from memory, sign-extends it to 32 bits or 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
0 1	1 1 1	0	0 0	1 x	0	imm9	1 0	Rn	Rt
size				opc					

32-bit (opc == 11)

```
LDTRSH  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDTRSH  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDTRSH  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDTRSH  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
```

```
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.66 LDTRSW

Load Register Signed Word (unprivileged) loads a word from memory, sign-extends it to 64 bits, and writes the result to a register. The address that is used for the load is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
1	0	1	1	1	0	0	0	1	0	0	imm9	1	0	Rn	Rt
size								opc							

```
LDTRSW  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
LDTRSW  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.67 LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL

Atomic unsigned maximum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDUMAXA` and `LDUMAXAL` load from memory with acquire semantics.
- `LDUMAXL` and `LDUMAXAL` store to memory with release semantics.
- `LDUMAX` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STUMAX, STUMAXL.

Integer
(FEAT_LSE)

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

32-bit LDUMAX `(size == 10 && A == 0 && R == 0)`

```
LDUMAX  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAX  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDUMAXA `(size == 10 && A == 1 && R == 0)`

```
LDUMAXA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAXA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDUMAXAL `(size == 10 && A == 1 && R == 1)`

```
LDUMAXAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAXAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDUMAXL `(size == 10 && A == 0 && R == 1)`

```
LDUMAXL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAXL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMAX `(size == 11 && A == 0 && R == 0)`

```
LDUMAX  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAX  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMAXA `(size == 11 && A == 1 && R == 0)`

```
LDUMAXA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAXA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMAXAL `(size == 11 && A == 1 && R == 1)`

```
LDUMAXAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAXAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMAXL `(size == 11 && A == 0 && R == 1)`

```
LDUMAXL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMAXL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAX, STUMAXL	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.68 LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB

Atomic unsigned maximum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, LDUMAXAB and LDUMAXALB load from memory with acquire semantics.
- LDUMAXLB and LDUMAXALB store to memory with release semantics.
- LDUMAXB has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STUMAXB, STUMAXLB.

Integer
(FEAT_LSE)

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

LDUMAXAB (A == 1 && R == 0)

```
LDUMAXAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMAXALB (A == 1 && R == 1)

```
LDUMAXALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMAXB (A == 0 && R == 0)

```
LDUMAXB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMAXLB (A == 0 && R == 1)

```
LDUMAXLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAXB, STUMAXLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.69 LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH

Atomic unsigned maximum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMAXAH` and `LDUMAXALH` load from memory with acquire semantics.
- `LDUMAXLH` and `LDUMAXALH` store to memory with release semantics.
- `LDUMAXH` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STUMAXH, STUMAXLH.

Integer

(FEAT_LSE)

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

LDUMAXAH (A == 1 && R == 0)

```
LDUMAXAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMAXALH (A == 1 && R == 1)

```
LDUMAXALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMAXH (A == 0 && R == 0)

```
LDUMAXH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMAXLH (A == 0 && R == 1)

```
LDUMAXLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMAXLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMAXH, STUMAXLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.70 LDUMIN, LDUMINA, LDUMINAL, LDUMINL

Atomic unsigned minimum on word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `LDUMINA` and `LDUMINAL` load from memory with acquire semantics.
- `LDUMINL` and `LDUMINAL` store to memory with release semantics.
- `LDUMIN` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STUMIN, STUMINL.

Integer
(FEAT_LSE)

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

32-bit LDUMIN (size == 10 && A == 0 && R == 0)

```
LDUMIN  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMIN  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDUMINA (size == 10 && A == 1 && R == 0)

```
LDUMINA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMINA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDUMINAL (size == 10 && A == 1 && R == 1)

```
LDUMINAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMINAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit LDUMINL (size == 10 && A == 0 && R == 1)

```
LDUMINL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMINL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMIN (size == 11 && A == 0 && R == 0)

```
LDUMIN  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMIN  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMINA (size == 11 && A == 1 && R == 0)

```
LDUMINA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMINA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMINAL (size == 11 && A == 1 && R == 1)

```
LDUMINAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMINAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit LDUMINL (size == 11 && A == 0 && R == 1)

```
LDUMINL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDUMINL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMIN, STUMINL	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.71 LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB

Atomic unsigned minimum on byte in memory atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMINAB` and `LDUMINALB` load from memory with acquire semantics.
- `LDUMINLB` and `LDUMINALB` store to memory with release semantics.
- `LDUMINB` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STUMINB, STUMINLB.

Integer
(FEAT_LSE)

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

LDUMINAB (A == 1 && R == 0)

```
LDUMINAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMINALB (A == 1 && R == 1)

```
LDUMINALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMINB (A == 0 && R == 0)

```
LDUMINB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMINLB (A == 0 && R == 1)

```
LDUMINLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

Alias	Is preferred when
STUMINB, STUMINLB	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.72 LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH

Atomic unsigned minimum on halfword in memory atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `LDUMINAH` and `LDUMINALH` load from memory with acquire semantics.
- `LDUMINLH` and `LDUMINALH` store to memory with release semantics.
- `LDUMINH` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This instruction is used by the alias STUMINH, STUMINLH.

Integer
(FEAT_LSE)

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

LDUMINAH (A == 1 && R == 0)

```
LDUMINAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMINALH (A == 1 && R == 1)

```
LDUMINALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMINH (A == 0 && R == 0)

```
LDUMINH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

LDUMINLH (A == 0 && R == 1)

```
LDUMINLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

LDUMINLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
MemAtomicOp op;
case opc of
    when '000' op = MemAtomicOp_ADD;
    when '001' op = MemAtomicOp_BIC;
    when '010' op = MemAtomicOp_EOR;
    when '011' op = MemAtomicOp_ORR;
    when '100' op = MemAtomicOp_SMAX;
    when '101' op = MemAtomicOp_SMIN;
    when '110' op = MemAtomicOp_UMAX;
    when '111' op = MemAtomicOp_UMIN;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Alias Conditions

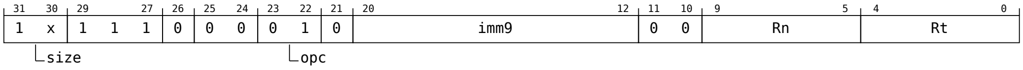
Alias	Is preferred when
STUMINH, STUMINLH	`A == '0' && Rt == '11111'`

Operation

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

value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, op, value, ldacctype, stacctype);

if t != 31 then
    X[t] = ZeroExtend(data, regsize);
```

4.2.73 LDUR

Load Register (unscaled) calculates an address from a base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



32-bit (size == 10)

```
LDUR  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDUR  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDUR  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDUR  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

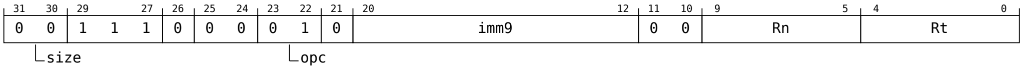
```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.74 LDURB

Load Register Byte (unscaled) calculates an address from a base register and an immediate offset, loads a byte from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



```
LDURB  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
LDURB  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

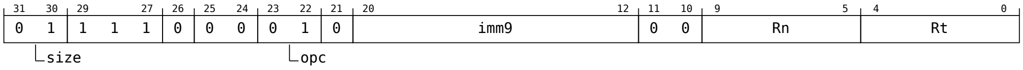
```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.75 LDURH

Load Register Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.



```
LDURH  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
LDURH  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.76 LDURSB

Load Register Signed Byte (unscaled) calculates an address from a base register and an immediate offset, loads a signed byte from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	1	x	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDURSB  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDURSB  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDURSB  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDURSB  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.77 LDURSH

Load Register Signed Halfword (unscaled) calculates an address from a base register and an immediate offset, loads a signed halfword from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	1	x	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (opc == 11)

```
LDURSH  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDURSH  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
LDURSH  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDURSH  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.78 LDURSW

Load Register Signed Word (unscaled) calculates an address from a base register and an immediate offset, loads a signed word from memory, sign-extends it, and writes it to a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
1	0	1	1	1	0	0	0	1	0	0	imm9	0	0	Rn	Rt
size								opc							

```
LDURSW  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
LDURSW  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.79 LDXP

Load Exclusive Pair of Registers derives an address from a base register value, loads two 32-bit words or two 64-bit doublewords from memory, and writes them to two registers. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and is single-copy atomic for each doubleword at doubleword granularity. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	sz	0 0 1 0 0 0	0	1	1	(1)(1)(1)(1)(1)	0	Rt2	Rn	Rt
				L		Rs	o0			

32-bit (sz == 0)

```
LDXP  <Wt1>, <Wt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDXP  <Wt1>, <Wt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (sz == 1)

```
LDXP  <Xt1>, <Xt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDXP  <Xt1>, <Xt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDXP*.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
```

```
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
```

```
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.80 LDXR

Load Exclusive Register derives an address from a base register value, loads a 32-bit word or a 64-bit doubleword from memory, and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0 0 1 0 0 0	0	1	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

32-bit (size == 10)

```
LDXR  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDXR  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
LDXR  <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDXR  <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

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

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

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

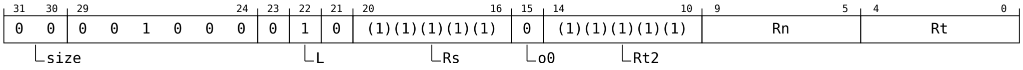
<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.81 LDXRB

Load Exclusive Register Byte derives an address from a base register value, loads a byte from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/Store addressing modes*.


31 30 29 24 23 22 21 20 16 15 14 10 9 5 4 0
0 0 0 0 1 0 0 0 0 1 0 (1)(1)(1)(1)(1) 0 (1)(1)(1)(1)(1) Rn Rt
size
L
Rs
o0
Rt2

```
LDXRB  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDXRB  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

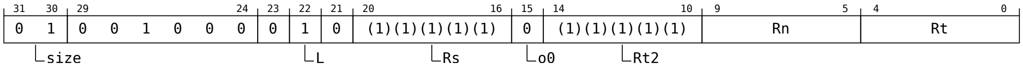
Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
```

```
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.82 LDXRH

Load Exclusive Register Halfword derives an address from a base register value, loads a halfword from memory, zero-extends it and writes it to a register. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See *Synchronization and semaphores*. For information about memory accesses, see *Load/Store addressing modes*.



```
LDXRH  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
LDXRH  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
```

```
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.83 PRFM (immediate)

Prefetch Memory (immediate) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21 10	9 5	4 0
1	1	1	1	1	0	0	1	1	0	imm12	Rn	Rt
	size								opc			

```
PRFM  (<prfop>|#<imm5>), [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
PRFM  (<prfop>|#<imm5>), [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
```

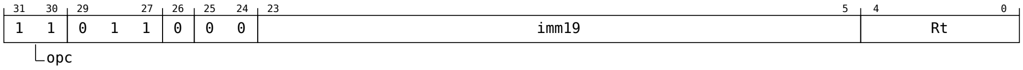
```
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.84 PRFM (literal)

Prefetch Memory (literal) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.



```
PRFM  (<prfop>|#<imm5>), <label>
```

```
integer t = UInt(Rt);
MemOp memop = MemOp_LOAD;
boolean signed = FALSE;
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 4;
        signed = TRUE;
    when '11'
        memop = MemOp_PREFETCH;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
VirtualAddress base = VAFromCapability(PCC);
bits(64) address = VAddress(base) + offset;

bits(size*8) data;

case memop of
    when MemOp_LOAD
        VACheckAddress(base, address, size, CAP_PERM_LOAD, AccType_NORMAL);
        data = Mem[address, size, AccType_NORMAL];
        if signed then
            X[t] = SignExtend(data, 64);
        else
            X[t] = data;

    when MemOp_PREFETCH
        Prefetch(address, t<4:0>);
```

4.2.85 PRFM (register)

Prefetch Memory (register) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
1	1	1 1 1	0	0	0	1	0	1	Rm	option	S	1	0	Rn	Rt
size						opc									

```
PRFM  (<prfop>|#<imm5>), [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
PRFM  (<prfop>|#<imm5>), [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;                 // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

```

```
if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.86 PRFUM

Prefetch Memory (unscaled offset) signals the memory system that data memory accesses from a specified address are likely to occur in the near future. The memory system can respond by taking actions that are expected to speed up the memory accesses when they do occur, such as preloading the cache line containing the specified address into one or more caches.

The effect of an PRFUM instruction is IMPLEMENTATION DEFINED. For more information, see *Prefetch memory*.

For information about memory accesses, see *Load/Store addressing modes*.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
1 1	1 1 1	0	0 0	1 0	0	imm9	0 0	Rn	Rt
size				opc					

```
PRFUM (<prfop>|#<imm5>), [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
PRFUM (<prfop>|#<imm5>), [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<prfop> Is the prefetch operation, defined as <type><target><policy>. <type> is one of:

PLD

Prefetch for load, encoded in the "Rt<4:3>" field as 0b00.

PLI

Preload instructions, encoded in the "Rt<4:3>" field as 0b01.

PST

Prefetch for store, encoded in the "Rt<4:3>" field as 0b10.

<target> is one of:

L1

Level 1 cache, encoded in the "Rt<2:1>" field as 0b00.

L2

Level 2 cache, encoded in the "Rt<2:1>" field as 0b01.

L3

Level 3 cache, encoded in the "Rt<2:1>" field as 0b10.

<policy> is one of:

KEEP

Retained or temporal prefetch, allocated in the cache normally. Encoded in the "Rt<0>" field as 0.

STRM

Streaming or non-temporal prefetch, for data that is used only once. Encoded in the "Rt<0>" field as 1.

For more information on these prefetch operations, see *Prefetch memory*. For other encodings of the "Rt" field, use <imm5>.

<imm5> Is the prefetch operation encoding as an immediate, in the range 0 to 31, encoded in the "Rt" field. This syntax is only for encodings that are not accessible using <prfop>.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
```

```
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.87 RET

Return from subroutine branches unconditionally to an address in a register, with a hint that this is a subroutine return.

31 25	24	23	22 21	20 16	15 12	11	10	9 5	4 0
1 1 0 1 0 1 1	0	0	1 0	1 1 1 1 1	0 0 0 0	0	0	Rn	0 0 0 0 0
	Z		op			A	M		Rm

```
RET  {<Xn>}
```

```
integer n = UInt(Rn);
BranchType branch_type;

case op of
    when '00' branch_type = BranchType_INDIR;
    when '01' branch_type = BranchType_INDCALL;
    when '10' branch_type = BranchType_RET;
    otherwise UNDEFINED;
```

Assembler Symbols

<Xn> Is the optional name of the general-purpose register holding the address to be branched to, defaulting to X30 in A64, encoded in the "Rn" field. On disassembly, the <Xn> argument may be omitted if it is X30 and the ISA is A64.

Operation

```
Capability target;
if CCTLR[].PCCBO == '1' then
    target = CapSetOffset(PCC[], X[n]);
else
    target = CapSetValue(PCC[], X[n]);

if branch_type == BranchType_INDCALL then
    if IsInC64() then
        if CCTLR[].SBL == '1' then
            C[30] = CapSetObjectType(CapAdd(PCC[], 5), CAP_SEAL_TYPE_RB);
        else
            C[30] = CapAdd(PCC[], 5);
    elsif CCTLR[].PCCBO == '1' then
        X[30] = PC[] + 4 - CapGetBase(PCC[]);
    else
        X[30] = PC[] + 4;

BranchToCapability(target,branch_type);
```

4.2.88 STADD, STADDL

Atomic add on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADD` has no memory ordering semantics.
- `STADDL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDADD, LDADDA, LDADDAL, LDADDL. This means:

- The encodings in this description are named to match the encodings of LDADD, LDADDA, LDADDAL, LDADDL.
- The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4	3	2	1	0
1	x	1	1	1	0	0	0	0	R	1	Rs	0	0	0	0	0	0	Rn	1	1	1	1	1
size								A					opc						Rt				

32-bit LDADD alias `(size == 10 && R == 0)`

```
STADD  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADD  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADD<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDADDL alias `(size == 10 && R == 1)`

```
STADDL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADDL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADDL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDADD alias `(size == 11 && R == 0)`

```
STADD  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADD  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADD<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDADDL alias `(size == 11 && R == 1)`

```
STADDL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADDL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADDL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDADD, LDADDA, LDADDAL, LDADDL gives the operational pseudocode for this instruction.

4.2.89 STADDB, STADDLB

Atomic add on byte in memory, without return, atomically loads an 8-bit byte from memory, adds the value held in a register to it, and stores the result back to memory.

- `STADDB` has no memory ordering semantics.
- `STADDLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDADDB, LDADDAB, LDADDALB, LDADDLB. This means:

- The encodings in this description are named to match the encodings of LDADDB, LDADDAB, LDADDALB, LDADDLB.
- The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STADDB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADDB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADDB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STADDLB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADDLB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADDLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDADDB, LDADDAB, LDADDALB, LDADDLB gives the operational pseudocode for this instruction.

4.2.90 STADDH, STADDLH

Atomic add on halfword in memory, without return, atomically loads a 16-bit halfword from memory, adds the value held in a register to it, and stores the result back to memory.

- STADDH has no memory ordering semantics.
- STADDLH stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDADDH, LDADDAH, LDADDALH, LDADDLH. This means:

- The encodings in this description are named to match the encodings of LDADDH, LDADDAH, LDADDALH, LDADDLH.
- The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering **(R == 0)**

```
STADDH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADDH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADDH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release **(R == 1)**

```
STADDLH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STADDLH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDADDLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDADDH, LDADDAH, LDADDALH, LDADDLH gives the operational pseudocode for this instruction.

4.2.91 STCLR, STCLRL

Atomic bit clear on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- STCLR has no memory ordering semantics.
- STCLRL stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDCLR, LDCLRA, LDCLRAL, LDCLRL. This means:

- The encodings in this description are named to match the encodings of LDCLR, LDCLRA, LDCLRAL, LDCLRL.
- The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

32-bit LDCLR alias (size == 10 && R == 0)

```
STCLR  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STCLR  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLR<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDCLRL alias (size == 10 && R == 1)

```
STCLRL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STCLRL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLRL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDCLR alias (size == 11 && R == 0)

```
STCLR  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STCLR  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLR<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDCLRL alias (size == 11 && R == 1)

```
STCLRL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STCLRL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLRL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDCLR, LDCLRA, LDCLRAL, LDCLRL gives the operational pseudocode for this instruction.

4.2.92 STCLRB, STCLRLB

Atomic bit clear on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRB` has no memory ordering semantics.
- `STCLRLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB. This means:

- The encodings in this description are named to match the encodings of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB.
- The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STCLRB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STCLRB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLRB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STCLRLB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STCLRLB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLRLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB gives the operational pseudocode for this instruction.

4.2.93 STCLRH, STCLRLH

Atomic bit clear on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise AND with the complement of the value held in a register on it, and stores the result back to memory.

- `STCLRH` has no memory ordering semantics.
- `STCLRLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH. This means:

- The encodings in this description are named to match the encodings of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH.
- The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STCLRH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STCLRH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLRH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STCLRLH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STCLRLH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDCLRLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH gives the operational pseudocode for this instruction.

4.2.94 STEOR, STEORL

Atomic exclusive OR on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEOR` has no memory ordering semantics.
- `STEORL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDEOR, LDEORA, LDEORAL, LDEORL. This means:

- The encodings in this description are named to match the encodings of LDEOR, LDEORA, LDEORAL, LDEORL.
- The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

32-bit LDEOR alias `(size == 10 && R == 0)`

```
STEOR  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEOR  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEOR<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDEORL alias `(size == 10 && R == 1)`

```
STEORL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEORL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEORL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDEOR alias `(size == 11 && R == 0)`

```
STEOR  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEOR  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEOR<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDEORL alias `(size == 11 && R == 1)`

```
STEORL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEORL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEORL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDEOR, LDEORA, LDEORAL, LDEORL gives the operational pseudocode for this instruction.

4.2.95 STEORB, STEORLB

Atomic exclusive OR on byte in memory, without return, atomically loads an 8-bit byte from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

- `STEORB` has no memory ordering semantics.
- `STEORLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDEORB, LDEORAB, LDEORALB, LDEORLB. This means:

- The encodings in this description are named to match the encodings of LDEORB, LDEORAB, LDEORALB, LDEORLB.
- The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering `(R == 0)`

```
STEORB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEORB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEORB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release `(R == 1)`

```
STEORLB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEORLB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEORLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDEORB, LDEORAB, LDEORALB, LDEORLB gives the operational pseudocode for this instruction.

4.2.96 STEORH, STEORLH

Atomic exclusive OR on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs an exclusive OR with the value held in a register on it, and stores the result back to memory.

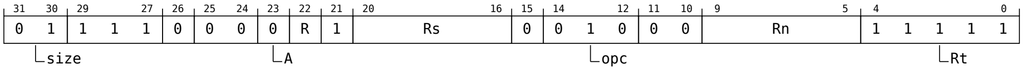
- `STEORH` has no memory ordering semantics.
- `STEORLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDEORH, LDEORAH, LDEORALH, LDEORLH. This means:

- The encodings in this description are named to match the encodings of LDEORH, LDEORAH, LDEORALH, LDEORLH.
- The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)



No memory ordering (R == 0)

```
STEORH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEORH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEORH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STEORLH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STEORLH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDEORLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDEORH, LDEORAH, LDEORALH, LDEORLH gives the operational pseudocode for this instruction.

4.2.97 STLLR

Store LORelease Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(FEAT_LOR)

31	30	29 ... 24	23	22	21	20 ... 16	15	14 ... 10	9 ... 5	4 ... 0
1	x	0 0 1 0 0 0	1	0	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

32-bit (size == 10)

```
STLLR  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLLR  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STLLR  <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLLR  <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

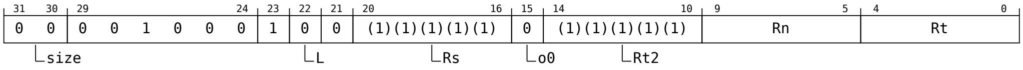
```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.98 STLLRB

Store LORelease Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(FEAT_LOR)


31 30 29 24 23 22 21 20 16 15 14 10 9 5 4 0
0 0 0 0 1 0 0 0 1 0 0 (1)(1)(1)(1)(1) 0 (1)(1)(1)(1)(1) Rn Rt
size
L
Rs
o0
Rt2

```
STLLRB  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLLRB  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.99 STLLRH

Store LORelease Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load LOAcquire, Store LORelease*. For information about memory accesses, see *Load/Store addressing modes*.

No offset

(FEAT_LOR)

31	30	29					24	23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	1	0	0	0	1	0	0	(1)(1)(1)(1)(1)	0	(1)(1)(1)(1)(1)	Rn	Rt
size									L		Rs	o0	Rt2		

```
STLLRH  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLLRH  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.100 STLR

Store-Release Register stores a 32-bit word or a 64-bit doubleword to a memory location, from a register. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29					24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
1	x	0	0	1	0	0	0	1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size									L		Rs	o0	Rt2		

32-bit (size == 10)

```
STLR  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STLR  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STLR  <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STLR  <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.101 STLRB

Store-Release Register Byte stores a byte from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 ... 24	23	22	21	20 ... 16	15	14 ... 10	9 ... 5	4 ... 0
0	0	0 0 1 0 0 0	1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size				L		Rs	o0	Rt2		

```
STLRB  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLRB  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.102 STLRH

Store-Release Register Halfword stores a halfword from a 32-bit register to a memory location. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29					24	23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	1	0	0	0	1	0	0	(1)(1)(1)(1)(1)	1	(1)(1)(1)(1)(1)	Rn	Rt
size									L		Rs	o0	Rt2		

```
STLRH  <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLRH  <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '0' then AccType_LIMITEDORDERED else AccType_ORDERED;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = elsize;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = X[t];
        Mem[address, dbytes, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, dbytes, acctype];
        X[t] = ZeroExtend(data, regsize);
```

4.2.103 STLXP

Store-Release Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords to a memory location if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	sz	0 0 1 0 0 0	0	0	1	Rs	1	Rt2	Rn	Rt
				L			o0			

32-bit (sz == 0)

```
STLXP  <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STLXP  <Ws>, <Wt1>, <Wt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (sz == 1)

```
STLXP  <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STLXP  <Ws>, <Xt1>, <Xt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXP*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

field.

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

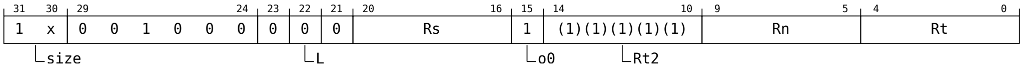
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.104 STLXR

Store-Release Exclusive Register stores a 32-bit word or a 64-bit doubleword to memory if the PE has exclusive access to the memory address, from two registers, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



31	30	29					24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0	0	1	0	0	0	0	0	0	Rs	1	(1)(1)(1)(1)(1)	Rn	Rt
size									L			o0	Rt2		

32-bit (size == 10)

```
STLXR  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STLXR  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STLXR  <Ws>, <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STLXR  <Ws>, <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXR*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.

- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

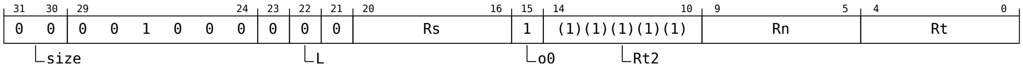
If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.105 STLXRB

Store-Release Exclusive Register Byte stores a byte from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



31 30	29 … 24	23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 0	0 0 1 0 0 0	0	0	0	Rs	1	(1)(1)(1)(1)(1)	Rn	Rt
size			L			o0	Rt2		

```
STLXRB  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLXRB  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXRB*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

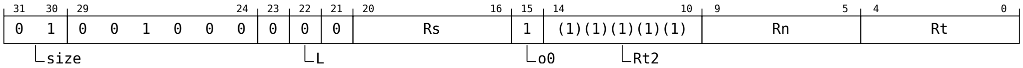
Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
```

```
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.106 STLXRH

Store-Release Exclusive Register Halfword stores a halfword from a 32-bit register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic. The instruction also has memory ordering semantics as described in *Load-Acquire, Store-Release*. For information about memory accesses, see *Load/Store addressing modes*.



```
STLXRH  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STLXRH  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STLXRH*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.107 STNP

Store Pair of Registers, with non-temporal hint, calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*. For information about Non-temporal pair instructions, see *Load/Store Non-temporal pair*.

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	0	0	0	imm7	Rt2	Rn	Rt
opc									L				

32-bit (opc == 00)

```
STNP  <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')

STNP  <Wt1>, <Wt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
STNP  <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')

STNP  <Xt1>, <Xt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_STREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc<0> == '1' then UNDEFINED;
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        X[t]  = data1;
        X[t2] = data2;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.108 STP

Store Pair of Registers calculates an address from a base register value and an immediate offset, and stores two 32-bit words or two 64-bit doublewords to the calculated address, from two registers. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	0	1	0	imm7	Rt2	Rn	Rt
	opc								L				

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
STP <Wt1>, <Wt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
STP <Xt1>, <Xt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

Pre-index

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	1	1	0	imm7	Rt2	Rn	Rt
	opc								L				

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
STP <Wt1>, <Wt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
STP <Xt1>, <Xt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

Signed offset

31	30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
x	0	1	0	1	0	0	1	0	0	imm7	Rt2	Rn	Rt
	opc								L				

32-bit (opc == 00)

```
STP <Wt1>, <Wt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STP <Wt1>, <Wt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 10)

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STP <Xt1>, <Xt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STP*.

Assembler Symbols

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_NORMAL;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if L:opc<0> == '01' || opc == '11' then UNDEFINED;
boolean signed = (opc<0> != '0');
integer scale = 2 + UInt(opc<1>);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

boolean wb_unknown = FALSE;

if memop == MemOp_LOAD && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;         // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;     // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;    // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        if rt_unknown && t == n then
            data1 = bits(datasize) UNKNOWN;
        else
            data1 = X[t];
        if rt_unknown && t2 == n then
            data2 = bits(datasize) UNKNOWN;
        else
            data2 = X[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        if signed then
            X[t]  = SignExtend(data1, 64);
            X[t2] = SignExtend(data2, 64);
        else
            X[t]  = data1;
            X[t2] = data2;

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.109 STR (immediate)

Store Register (immediate) stores a word or a doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>], #<simm> // (PSTATE.C64 == '0')

STR <Wt>, [<Cn|CSP>], #<simm> // (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>], #<simm> // (PSTATE.C64 == '0')

STR <Xt>, [<Cn|CSP>], #<simm> // (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
1	x	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size								opc							

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, #<simm>]! // (PSTATE.C64 == '0')

STR <Wt>, [<Cn|CSP>, #<simm>]! // (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, #<simm>]! // (PSTATE.C64 == '0')

STR <Xt>, [<Cn|CSP>, #<simm>]! // (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 10	9 5	4 0
1	x	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size								opc				

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>{, #<pimm>}] // (PSTATE.C64 == '0')

STR <Wt>, [<Cn|CSP>{, #<pimm>}] // (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>{, #<pimm>}] // (PSTATE.C64 == '0')

STR <Xt>, [<Cn|CSP>{, #<pimm>}] // (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
```

```

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.110 STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29 27	26	25 24	23 22	21	20 16	15 13	12	11 10	9 5	4 0
1	x	1 1 1	0	0 0	0 0	1	Rm	option	S	1 0	Rn	Rt
size					opc							

32-bit (size == 10)

```
STR <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '0')

STR <Wt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STR <Xt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '0')

STR <Xt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] // (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL.

Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#3

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
```

```
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.111 STRB (immediate)

Store Register Byte (immediate) stores the least significant byte of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
	size								opc						

```
STRB  <Wt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
STRB  <Wt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
	size								opc						

```
STRB  <Wt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
STRB  <Wt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 10	9 5	4 0
0	0	1	1	1	0	0	1	0	0	imm12	Rn	Rt
	size								opc			

```
STRB  <Wt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
STRB  <Wt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STRB (immediate)*.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address,

encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
```

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

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.112 STRB (register)

Store Register Byte (register) calculates an address from a base register value and an offset register value, and stores a byte from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29		27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

Extended register (option != 011)

```
STRB  <Wt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '0')
```

```
STRB  <Wt>, [<Cn|CSP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '1')
```

Shifted register (option == 011)

```
STRB  <Wt>, [<Xn|SP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '0')
```

```
STRB  <Wt>, [<Cn|CSP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend specifier, encoded in"option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

<amount> Is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.113 STRH (immediate)

Store Register Halfword (immediate) stores the least significant halfword of a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset. For information about memory accesses, see *Load/Store addressing modes*.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	0	1	Rn	Rt
size								opc							

```
STRH  <Wt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
STRH  <Wt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	1	1	1	1	0	0	0	0	0	0	imm9	1	1	Rn	Rt
size								opc							

```
STRH  <Wt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
STRH  <Wt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31	30	29		27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
0	1	1	1	1	0	0	1	0	0	imm12	Rn	Rt
size								opc				

```
STRH  <Wt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
STRH  <Wt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STRH (immediate)*.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address,

encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> Is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
```

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

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.114 STRH (register)

Store Register Halfword (register) calculates an address from a base register value and an offset register value, and stores a halfword from a 32-bit register to the calculated address. For information about memory accesses, see *Load/Store addressing modes*.

The instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an offset register value. The offset can be optionally shifted and extended.

31	30	29		27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	0	0	1	Rm	option	S	1	0	Rn	Rt
size								opc									

```
STRH  <Wt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
STRH  <Wt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> Is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> Is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#1

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
```

```
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.115 STSET, STSETL

Atomic bit set on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSET` has no memory ordering semantics.
- `STSETL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSET, LDSETA, LDSETAL, LDSETL. This means:

- The encodings in this description are named to match the encodings of LDSET, LDSETA, LDSETAL, LDSETL.
- The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

32-bit LDSET alias `(size == 10 && R == 0)`

```
STSET  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSET  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSET<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDSETL alias `(size == 10 && R == 1)`

```
STSETL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSETL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSETL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDSET alias `(size == 11 && R == 0)`

```
STSET  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSET  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSET<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDSETL alias `(size == 11 && R == 1)`

```
STSETL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSETL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSETL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSET, LDSETA, LDSETAL, LDSETL gives the operational pseudocode for this instruction.

4.2.116 STSETB, STSETLB

Atomic bit set on byte in memory, without return, atomically loads an 8-bit byte from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETB` has no memory ordering semantics.
- `STSETLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSETB, LDSETAB, LDSETALB, LDSETLB. This means:

- The encodings in this description are named to match the encodings of LDSETB, LDSETAB, LDSETALB, LDSETLB.
- The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering `(R == 0)`

```
STSETB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STSETB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSETB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release `(R == 1)`

```
STSETLB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STSETLB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSETLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSETB, LDSETAB, LDSETALB, LDSETLB gives the operational pseudocode for this instruction.

4.2.117 STSETH, STSETLH

Atomic bit set on halfword in memory, without return, atomically loads a 16-bit halfword from memory, performs a bitwise OR with the value held in a register on it, and stores the result back to memory.

- `STSETH` has no memory ordering semantics.
- `STSETLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSETH, LDSETAH, LDSETALH, LDSETLH. This means:

- The encodings in this description are named to match the encodings of LDSETH, LDSETAH, LDSETALH, LDSETLH.
- The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STSETH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STSETH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSETH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STSETLH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STSETLH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSETLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSETH, LDSETAH, LDSETALH, LDSETLH gives the operational pseudocode for this instruction.

4.2.118 STSMAX, STSMAXL

Atomic signed maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAX` has no memory ordering semantics.
- `STSMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL. This means:

- The encodings in this description are named to match the encodings of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL.
- The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

32-bit LDSMAX alias `(size == 10 && R == 0)`

```
STSMAX  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAX  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAX<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDSMAXL alias `(size == 10 && R == 1)`

```
STSMAXL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAXL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAXL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDSMAX alias `(size == 11 && R == 0)`

```
STSMAX  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAX  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAX<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDSMAXL alias `(size == 11 && R == 1)`

```
STSMAXL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAXL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAXL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL gives the operational pseudocode for this instruction.

4.2.119 STSMAXB, STSMAXLB

Atomic signed maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXB` has no memory ordering semantics.
- `STSMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB. This means:

- The encodings in this description are named to match the encodings of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB.
- The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering `(R == 0)`

```
STSMAXB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAXB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAXB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release `(R == 1)`

```
STSMAXLB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAXLB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAXLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB gives the operational pseudocode for this instruction.

4.2.120 STSMAXH, STSMAXLH

Atomic signed maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as signed numbers.

- `STSMAXH` has no memory ordering semantics.
- `STSMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH. This means:

- The encodings in this description are named to match the encodings of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH.
- The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STSMAXH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAXH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAXH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STSMAXLH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMAXLH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMAXLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH gives the operational pseudocode for this instruction.

4.2.121 STSMIN, STSMINL

Atomic signed minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMIN` has no memory ordering semantics.
- `STSMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMIN, LDSMINA, LDSMINAL, LDSMINL. This means:

- The encodings in this description are named to match the encodings of LDSMIN, LDSMINA, LDSMINAL, LDSMINL.
- The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

32-bit LDSMIN alias `(size == 10 && R == 0)`

```
STSMIN  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMIN  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMIN<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDSMINL alias `(size == 10 && R == 1)`

```
STSMINL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMINL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMINL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDSMIN alias `(size == 11 && R == 0)`

```
STSMIN  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMIN  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMIN<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDSMINL alias `(size == 11 && R == 1)`

```
STSMINL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMINL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMINL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSMIN, LDSMINA, LDSMINAL, LDSMINL gives the operational pseudocode for this instruction.

4.2.122 STSMINB, STSMINLB

Atomic signed minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINB` has no memory ordering semantics.
- `STSMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB. This means:

- The encodings in this description are named to match the encodings of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB.
- The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STSMINB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMINB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMINB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STSMINLB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STSMINLB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMINLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB gives the operational pseudocode for this instruction.

4.2.123 STSMINH, STSMINLH

Atomic signed minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as signed numbers.

- `STSMINH` has no memory ordering semantics.
- `STSMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH. This means:

- The encodings in this description are named to match the encodings of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH.
- The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

31	30	29	28	27	26	25	24	23	22	21	20-16	15	14	13	12	11	10	9-5	4	3	2	1	0
0	1	1	1	1	0	0	0	0	R	1	Rs	0	1	0	1	0	0	Rn	1	1	1	1	1
size								A					opc						Rt				

No memory ordering `(R == 0)`

```
STSMINH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STSMINH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMINH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release `(R == 1)`

```
STSMINLH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STSMINLH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDSMINLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH gives the operational pseudocode for this instruction.

4.2.124 STTR

Store Register (unprivileged) stores a word or doubleword from a register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	x	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size								opc							

32-bit (size == 10)

```
STTR  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

STTR  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STTR  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

STTR  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.125 STTRB

Store Register Byte (unprivileged) stores a byte from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
0	0	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size								opc							

```
STTRB  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STTRB  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.126 STTRH

Store Register Halfword (unprivileged) stores a halfword from a 32-bit register to memory. The address that is used for the store is calculated from a base register and an immediate offset.

Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the *Effective value* of PSTATE.UAO is 0 and either:

- The instruction is executed at EL1.
- The instruction is executed at EL2 when the *Effective value* of *HCR_EL2*.{E2H, TGE} is {1, 1}.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	0	0	0	imm9	1	0	Rn	Rt
size								opc							

```
STTRH  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STTRH  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);

unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.127 STUMAX, STUMAXL

Atomic unsigned maximum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAX` has no memory ordering semantics.
- `STUMAXL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL. This means:

- The encodings in this description are named to match the encodings of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL.
- The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

32-bit LDUMAX alias `(size == 10 && R == 0)`

```
STUMAX  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMAX  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAX<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDUMAXL alias `(size == 10 && R == 1)`

```
STUMAXL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMAXL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAXL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDUMAX alias `(size == 11 && R == 0)`

```
STUMAX  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMAX  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAX<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDUMAXL alias `(size == 11 && R == 1)`

```
STUMAXL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMAXL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAXL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL gives the operational pseudocode for this instruction.

4.2.128 STUMAXB, STUMAXLB

Atomic unsigned maximum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXB` has no memory ordering semantics.
- `STUMAXLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB. This means:

- The encodings in this description are named to match the encodings of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB.
- The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STUMAXB <Ws>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
STUMAXB <Ws>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAXB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STUMAXLB <Ws>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
STUMAXLB <Ws>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAXLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB gives the operational pseudocode for this instruction.

4.2.129 STUMAXH, STUMAXLH

Atomic unsigned maximum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the larger value back to memory, treating the values as unsigned numbers.

- `STUMAXH` has no memory ordering semantics.
- `STUMAXLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses see *Load/Store addressing modes*.

This is an alias of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH. This means:

- The encodings in this description are named to match the encodings of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH.
- The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STUMAXH <Ws>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
STUMAXH <Ws>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAXH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STUMAXLH <Ws>, [<Xn|SP>] // (PSTATE.C64 == '0')
```

```
STUMAXLH <Ws>, [<Cn|CSP>] // (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMAXLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH gives the operational pseudocode for this instruction.

4.2.130 STUMIN, STUMINL

Atomic unsigned minimum on word or doubleword in memory, without return, atomically loads a 32-bit word or 64-bit doubleword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMIN` has no memory ordering semantics.
- `STUMINL` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMIN, LDUMINA, LDUMINAL, LDUMINL. This means:

- The encodings in this description are named to match the encodings of LDUMIN, LDUMINA, LDUMINAL, LDUMINL.
- The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

32-bit LDUMIN alias `(size == 10 && R == 0)`

```
STUMIN  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STUMIN  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMIN<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

32-bit LDUMINL alias `(size == 10 && R == 1)`

```
STUMINL  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STUMINL  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMINL<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDUMIN alias `(size == 11 && R == 0)`

```
STUMIN  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STUMIN  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMIN<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

64-bit LDUMINL alias `(size == 11 && R == 1)`

```
STUMINL  <Xs>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STUMINL  <Xs>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMINL<Xs>, XZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDUMIN, LDUMINA, LDUMINAL, LDUMINL gives the operational pseudocode for this instruction.

4.2.131 STUMINB, STUMINLB

Atomic unsigned minimum on byte in memory, without return, atomically loads an 8-bit byte from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINB` has no memory ordering semantics.
- `STUMINLB` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB. This means:

- The encodings in this description are named to match the encodings of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB.
- The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering `(R == 0)`

```
STUMINB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMINB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMINB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release `(R == 1)`

```
STUMINLB  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMINLB  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMINLB<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB gives the operational pseudocode for this instruction.

4.2.132 STUMINH, STUMINLH

Atomic unsigned minimum on halfword in memory, without return, atomically loads a 16-bit halfword from memory, compares it against the value held in a register, and stores the smaller value back to memory, treating the values as unsigned numbers.

- `STUMINH` has no memory ordering semantics.
- `STUMINLH` stores to memory with release semantics, as described in *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

This is an alias of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH. This means:

- The encodings in this description are named to match the encodings of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH.
- The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

Integer
(FEAT_LSE)

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

No memory ordering (R == 0)

```
STUMINH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMINH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMINH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Release (R == 1)

```
STUMINLH  <Ws>, [<Xn|SP>] //  (PSTATE.C64 == '0')

STUMINLH  <Ws>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

is equivalent to

```
LDUMINLH<Ws>, WZR, <Addressing_Mode>
```

and is always the preferred disassembly.

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

The description of LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH gives the operational pseudocode for this instruction.

4.2.133 STUR

Store Register (unscaled) calculates an address from a base register value and an immediate offset, and stores a 32-bit word or a 64-bit doubleword to the calculated address, from a register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 ... 12	11	10	9 ... 5	4 ... 0
1	x	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

32-bit (size == 10)

```
STUR  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

STUR  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STUR  <Xt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

STUR  <Xt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.134 STURB

Store Register Byte (unscaled) calculates an address from a base register value and an immediate offset, and stores a byte to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	0	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

```
STURB  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STURB  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.135 STURH

Store Register Halfword (unscaled) calculates an address from a base register value and an immediate offset, and stores a halfword to the calculated address, from a 32-bit register. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
0	1	1	1	1	0	0	0	0	0	0	imm9	0	0	Rn	Rt
size								opc							

```
STURH  <Wt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STURH  <Wt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(size);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_NORMAL;
MemOp memop;
boolean signed;
integer regsize;

if opc<1> == '0' then
    // store or zero-extending load
    memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
    regsize = if size == '11' then 64 else 32;
    signed = FALSE;
else
    if size == '11' then
        memop = MemOp_PREFETCH;
        if opc<0> == '1' then UNDEFINED;
    else
        // sign-extending load
        memop = MemOp_LOAD;
        if size == '10' && opc<0> == '1' then UNDEFINED;
        regsize = if opc<0> == '1' then 32 else 64;
        signed = TRUE;

integer datasize = 8 << scale;
```

Operation

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

boolean wb_unknown = FALSE;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE && wback && n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base;

base = BaseReg[n, memop == MemOp_PREFETCH];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        else
            data = X[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        if signed then
            X[t] = SignExtend(data, regsize);
        else
            X[t] = ZeroExtend(data, regsize);

    when MemOp_PREFETCH
        address = VAddress(base);
        Prefetch(address, t<4:0>);

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.2.136 STXP

Store Exclusive Pair of registers stores two 32-bit words or two 64-bit doublewords from two registers to a memory location if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. A 32-bit pair requires the address to be doubleword aligned and is single-copy atomic at doubleword granularity. A 64-bit pair requires the address to be quadword aligned and, if the Store-Exclusive succeeds, it causes a single-copy atomic update of the 128-bit memory location being updated. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	sz	0 0 1 0 0 0	0	0	1	Rs	0	Rt2	Rn	Rt
				L			o0			

32-bit (sz == 0)

```
STXP  <Ws>, <Wt1>, <Wt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STXP  <Ws>, <Wt1>, <Wt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (sz == 1)

```
STXP  <Ws>, <Xt1>, <Xt2>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STXP  <Ws>, <Xt1>, <Xt2>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = TRUE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 32 << UInt(sz);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXP*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Xt1> Is the 64-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Xt2> Is the 64-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

<Wt1> Is the 32-bit name of the first general-purpose register to be transferred, encoded in the "Rt" field.

<Wt2> Is the 32-bit name of the second general-purpose register to be transferred, encoded in the "Rt2" field.

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

field.

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.137 STXR

Store Exclusive Register stores a 32-bit word or a 64-bit doubleword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. For information about memory accesses, see *Load/Store addressing modes*.

31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
1	x	0 0 1 0 0 0	0	0	0	Rs	0	(1)(1)(1)(1)(1)	Rn	Rt
size				L			o0	Rt2		

32-bit (size == 10)

```
STXR  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STXR  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11)

```
STXR  <Ws>, <Xt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')

STXR  <Ws>, <Xt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXR*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

Accessing an address that is not aligned to the size of the data being accessed causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

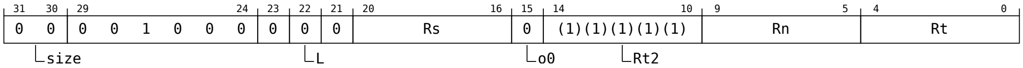
Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.138 STXRB

Store Exclusive Register Byte stores a byte from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses, see *Load/Store addressing modes*.



31	30	29 24	23	22	21	20 16	15	14 10	9 5	4 0
0	0	0 0 1 0 0 0	0	0	0	Rs	0	(1)(1)(1)(1)(1)	Rn	Rt
size				L			o0	Rt2		

```
STXRB  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STXRB  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *STXRB*.

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

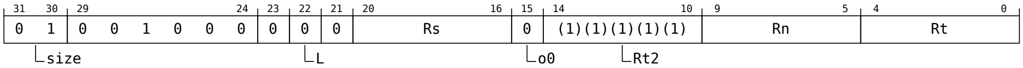
```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
```

```
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.139 STXRH

Store Exclusive Register Halfword stores a halfword from a register to memory if the PE has exclusive access to the memory address, and returns a status value of 0 if the store was successful, or of 1 if no store was performed. See *Synchronization and semaphores*. The memory access is atomic.

For information about memory accesses, see *Load/Store addressing modes*.



```
STXRH  <Ws>, <Wt>, [<Xn|SP>{,#0}] //  (PSTATE.C64 == '0')
```

```
STXRH  <Ws>, <Wt>, [<Cn|CSP>{,#0}] //  (PSTATE.C64 == '1')
```

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2); // ignored by load/store single register
integer s = UInt(Rs);   // ignored by all loads and store-release

AccType acctype = if o0 == '1' then AccType_ORDEREDATOMIC else AccType_ATOMIC;
boolean pair = FALSE;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer elsize = 8 << UInt(size);
integer regsize = if elsize == 64 then 64 else 32;
integer datasize = if pair then elsize * 2 else elsize;
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Ws> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If AArch64.ExclusiveMonitorsPass() returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If AArch64.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
bits(datasize) data;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if memop == MemOp_LOAD && pair && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if memop == MemOp_STORE then
    if s == t || (pair && s == t2) then
        Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
            when Constraint_NONE       rt_unknown = FALSE;   // store original value
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();
    if s == n && n != 31 then
        Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
        assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
            when Constraint_NONE       rn_unknown = FALSE;   // address is original base
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

VirtualAddress base;
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

bits(64) address = VAddress(base);

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        if rt_unknown then
            data = bits(datasize) UNKNOWN;
        elsif pair then
            bits(datasize DIV 2) el1 = X[t];
            bits(datasize DIV 2) el2 = X[t2];
            data = if BigEndian() then el1 : el2 else el2 : el1;
        else
            data = X[t];

        bit status = '1';
        // Check whether the Exclusives monitors are set to include the
        // physical memory locations corresponding to virtual address
        // range [address, address+dbytes-1].
        if AArch64.ExclusiveMonitorsPass(address, dbytes) then
            // This atomic write will be rejected if it does not refer
            // to the same physical locations after address translation.
            Mem[address, dbytes, acctype] = data;
            status = ExclusiveMonitorsStatus();
        X[s] = ZeroExtend(status, 32);

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        // Tell the Exclusives monitors to record a sequence of one or more atomic
        // memory reads from virtual address range [address, address+dbytes-1].
        // The Exclusives monitor will only be set if all the reads are from the
        // same dbytes-aligned physical address, to allow for the possibility of
        // an atomicity break if the translation is changed between reads.
        AArch64.SetExclusiveMonitors(address, dbytes);

        if pair then
            if rt_unknown then
                // ConstrainedUNPREDICTABLE case
                X[t]  = bits(datasize) UNKNOWN;        // In this case t = t2
            elsif elsize == 32 then
                // 32-bit load exclusive pair (atomic)
                data = Mem[address, dbytes, acctype];
                if BigEndian() then
                    X[t]  = data<datasize-1:elsize>;
                    X[t2] = data<elsize-1:0>;
                else
                    X[t]  = data<elsize-1:0>;
                    X[t2] = data<datasize-1:elsize>;
            else // elsize == 64
                // 64-bit load exclusive pair (not atomic),
                // but must be 128-bit aligned
                if address != Align(address, dbytes) then
                    iswrite = FALSE;
                    secondstage = FALSE;
                    AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));
                X[t]  = Mem[address + 0, 8, acctype];
                X[t2] = Mem[address + 8, 8, acctype];
        else
            data = Mem[address, dbytes, acctype];
            X[t] = ZeroExtend(data, regsize);
```

4.2.140 SWP, SWPA, SWPAL, SWPL

Swap word or doubleword in memory atomically loads a 32-bit word or 64-bit doubleword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not one of WZR or XZR, `SWPA` and `SWPAL` load from memory with acquire semantics.
- `SWPL` and `SWPAL` store to memory with release semantics.
- `SWP` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

Integer

(FEAT_LSE)

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

32-bit SWP (size == 10 && A == 0 && R == 0)

```
SWP  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWP  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit SWPA (size == 10 && A == 1 && R == 0)

```
SWPA  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPA  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit SWPAL (size == 10 && A == 1 && R == 1)

```
SWPAL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPAL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit SWPL (size == 10 && A == 0 && R == 1)

```
SWPL  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPL  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit SWP (size == 11 && A == 0 && R == 0)

```
SWP  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWP  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit SWPA (size == 11 && A == 1 && R == 0)

```
SWPA  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPA  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit SWPAL (size == 11 && A == 1 && R == 1)

```
SWPAL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPAL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit SWPL (size == 11 && A == 0 && R == 1)

```
SWPL  <Xs>, <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
SWPL  <Xs>, <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

Assembler Symbols

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

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

<Xs> Is the 64-bit name of the general-purpose register to be stored, encoded in the "Rs" field.

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

store_value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, MemAtomicOp_SWP, store_value, ldacctype, stacctype);

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

4.2.141 SWPB, SWPAB, SWPALB, SWPLB

Swap byte in memory atomically loads an 8-bit byte from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `SWPAB` and `SWPALB` load from memory with acquire semantics.
- `SWPLB` and `SWPALB` store to memory with release semantics.
- `SWPB` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

Integer
(FEAT_LSE)

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

SWPAB (A == 1 && R == 0)

```
SWPAB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPAB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

SWPALB (A == 1 && R == 1)

```
SWPALB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPALB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

SWPB (A == 0 && R == 0)

```
SWPB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

SWPLB (A == 0 && R == 1)

```
SWPLB  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPLB  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

store_value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, MemAtomicOp_SWP, store_value, ldacctype, stacctype);

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

4.2.142 SWPH, SWPAH, SWPALH, SWPLH

Swap halfword in memory atomically loads a 16-bit halfword from a memory location, and stores the value held in a register back to the same memory location. The value initially loaded from memory is returned in the destination register.

- If the destination register is not WZR, `SWPAH` and `SWPALH` load from memory with acquire semantics.
- `SWPLH` and `SWPALH` store to memory with release semantics.
- `SWPH` has no memory ordering requirements.

For more information about memory ordering semantics, see *Load-Acquire, Store-Release*.

For information about memory accesses, see *Load/Store addressing modes*.

Integer
(FEAT_LSE)

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

SWPAH (A == 1 && R == 0)

```
SWPAH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPAH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

SWPALH (A == 1 && R == 1)

```
SWPALH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPALH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

SWPH (A == 0 && R == 0)

```
SWPH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

SWPLH (A == 0 && R == 1)

```
SWPLH  <Ws>, <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '0')

SWPLH  <Ws>, <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
if !HaveAtomicExt() then UNDEFINED;

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

integer datasize = 8 << UInt(size);
integer regsize = if datasize == 64 then 64 else 32;
AccType ldacctype = if A == '1' && Rt != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = if R == '1' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

Operation

```
bits(64) address;
bits(datasize) data;
bits(datasize) store_value;

store_value = X[s];

VirtualAddress base = BaseReg[n];
data = MemAtomic(base, MemAtomicOp_SWP, store_value, ldacctype, stacctype);

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

4.3 Modified SIMD&FP instructions

4.3.1 LD1 (multiple structures)

Load multiple single-element structures to one, two, three, or four registers. This instruction loads multiple single-element structures from memory and writes the result to one, two, three, or four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21					16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	x	x	1	x	size	Rn	Rt
									L							opcode						

One register (opcode == 0111)

```
LD1 { <Vt>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1 { <Vt>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

Two registers (opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1 { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

Three registers (opcode == 0110)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

Four registers (opcode == 0010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1 { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	1	0	Rm	x	x	1	x	size	Rn	Rt
									L			opcode						

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
LD1 { <Vt>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD1 { <Vt>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
LD1 { <Vt>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1 { <Vt>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
LD1 { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Two registers, register offset (Rm != 11111 && opcode == 1010)

```
LD1  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

Three registers, immediate offset (Rm == 11111 && opcode == 0110)

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Three registers, register offset (Rm != 11111 && opcode == 0110)

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

Four registers, immediate offset (Rm == 11111 && opcode == 0010)

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Four registers, register offset (Rm != 11111 && opcode == 0010)

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
```

```
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.2 LD1 (single structure)

Load one single-element structure to one lane of one register. This instruction loads a single-element structure from memory and writes the result to the specified lane of the SIMD&FP register without affecting the other bits of the register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 0	1	0	0 0 0 0 0	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit (opcode == 000)

```
LD1  { <Vt>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit (opcode == 010 && size == x0)

```
LD1  { <Vt>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit (opcode == 100 && size == 00)

```
LD1  { <Vt>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
LD1  { <Vt>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 1	1	0	Rm	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
LD1  { <Vt>.B }[<index>], [<Xn|SP>], #1 //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.B }[<index>], [<Cn|CSP>], #1 //  (PSTATE.C64 == '1')
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
LD1  { <Vt>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
LD1  { <Vt>.H }[<index>], [<Xn|SP>], #2 //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.H }[<index>], [<Cn|CSP>], #2 //  (PSTATE.C64 == '1')
```

16-bit, register offset (`Rm != 11111 && opcode == 010 && size == x0`)

```
LD1  { <Vt>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (`Rm == 11111 && opcode == 100 && size == 00`)

```
LD1  { <Vt>.S }[<index>], [<Xn|SP>], #4 //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.S }[<index>], [<Cn|CSP>], #4 //  (PSTATE.C64 == '1')
```

32-bit, register offset (`Rm != 11111 && opcode == 100 && size == 00`)

```
LD1  { <Vt>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (`Rm == 11111 && opcode == 100 && S == 0 && size == 01`)

```
LD1  { <Vt>.D }[<index>], [<Xn|SP>], #8 //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.D }[<index>], [<Cn|CSP>], #8 //  (PSTATE.C64 == '1')
```

64-bit, register offset (`Rm != 11111 && opcode == 100 && S == 0 && size == 01`)

```
LD1  { <Vt>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1  { <Vt>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
```

```
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.3 LD1R

Load one single-element structure and Replicate to all lanes (of one register). This instruction loads a single-element structure from memory and replicates the structure to all the lanes of the SIMD&FP register.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 ... 23	22	21	20 ... 16	15 ... 13	12	11 10	9 ... 5	4 ... 0
0	Q	0 0 1 1 0 1 0	1	0	0 0 0 0 0	1 1 0	0	size	Rn	Rt
			L	R		opcode	S			

```
LD1R  { <Vt>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD1R  { <Vt>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 ... 23	22	21	20 ... 16	15 ... 13	12	11 10	9 ... 5	4 ... 0
0	Q	0 0 1 1 0 1 1	1	0	Rm	1 1 0	0	size	Rn	Rt
			L	R		opcode	S			

Immediate offset (Rm == 11111)

```
LD1R  { <Vt>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD1R  { <Vt>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset (Rm != 11111)

```
LD1R  { <Vt>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD1R  { <Vt>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"size":

size	<imm>
00	#1
01	#2
10	#4
11	#8

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);          // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);       // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);           // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);             // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.4 LD2 (multiple structures)

Load multiple 2-element structures to two registers. This instruction loads multiple 2-element structures from memory and writes the result to the two SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 23	22	21 16	15 12	11 10	9 5	4 0
0	Q	0 0 1 1 0 0 0	1	0 0 0 0 0 0	1 0 0 0	size	Rn	Rt
			L		opcode			

```
LD2  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 23	22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0 0 1 1 0 0 1	1	0	Rm	1 0 0 0	size	Rn	Rt
			L			opcode			

Immediate offset `(Rm == 11111)`

```
LD2  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset `(Rm != 11111)`

```
LD2  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1

modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.5 LD2 (single structure)

Load single 2-element structure to one lane of two registers. This instruction loads a 2-element structure from memory and writes the result to the corresponding elements of the two SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 ... 23	22	21	20 ... 16	15 ... 13	12	11 10	9 ... 5	4 ... 0
0	Q	0 0 1 1 0 1 0	1	1	0 0 0 0 0	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit `(opcode == 000)`

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit `(opcode == 010 && size == x0)`

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit `(opcode == 100 && size == 00)`

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit `(opcode == 100 && S == 0 && size == 01)`

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 ... 23	22	21	20 ... 16	15 ... 13	12	11 10	9 ... 5	4 ... 0
0	Q	0 0 1 1 0 1 1	1	1	Rm	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit, immediate offset `(Rm == 11111 && opcode == 000)`

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2 //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Cn|CSP>], #2 //  (PSTATE.C64 == '1')
```

8-bit, register offset `(Rm != 11111 && opcode == 000)`

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.B, <Vt2>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset `(Rm == 11111 && opcode == 010 && size == x0)`

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4 //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Cn|CSP>], #4 //  (PSTATE.C64 == '1')
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.H, <Vt2>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8 //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Cn|CSP>], #8 //  (PSTATE.C64 == '1')
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.S, <Vt2>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16 //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Cn|CSP>], #16 //  (PSTATE.C64 == '1')
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD2  { <Vt>.D, <Vt2>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
```

Copyright © 2019-2022 Arm Limited or its affiliates. All rights reserved.
Non-confidential

```
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.6 LD2R

Load single 2-element structure and Replicate to all lanes of two registers. This instruction loads a 2-element structure from memory and replicates the structure to all the lanes of the two SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 ... 23	22	21	20 ... 16	15 ... 13	12	11 10	9 ... 5	4 ... 0
0	Q	0 0 1 1 0 1 0	1	1	0 0 0 0 0	1 1 0	0	size	Rn	Rt
			L	R		opcode	S			

```
LD2R  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD2R  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 ... 23	22	21	20 ... 16	15 ... 13	12	11 10	9 ... 5	4 ... 0
0	Q	0 0 1 1 0 1 1	1	1	Rm	1 1 0	0	size	Rn	Rt
			L	R		opcode	S			

Immediate offset (Rm == 11111)

```
LD2R  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD2R  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset (Rm != 11111)

```
LD2R  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD2R  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"size":

size	<imm>
00	#2
01	#4
10	#8
11	#16

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

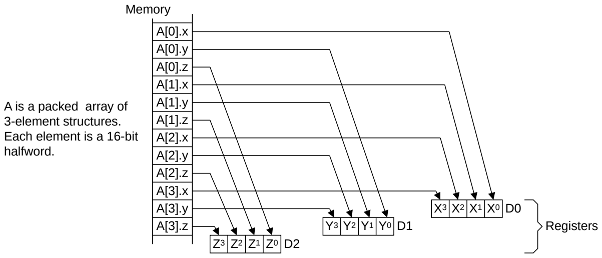
```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
```

```
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.7 LD3 (multiple structures)

Load multiple 3-element structures to three registers. This instruction loads multiple 3-element structures from memory and writes the result to the three SIMD&FP registers, with de-interleaving.

The following figure shows an example of the operation of de-interleaving of a LD3.16 (multiple 3-element structures) instruction:.



Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21					16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	size	Rn	Rt
									L							opcode						

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	1	0	Rm	0	1	0	0	size	Rn	Rt
									L			opcode						

Immediate offset (Rm == 11111)

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset (Rm != 11111)

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
```

```
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.8 LD3 (single structure)

Load single 3-element structure to one lane of three registers). This instruction loads a 3-element structure from memory and writes the result to the corresponding elements of the three SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 … 23	22	21	20 … 16	15 … 13	12	11 10	9 … 5	4 … 0
0	Q	0 0 1 1 0 1 0	1	0	0 0 0 0 0	x x 1	S	size	Rn	Rt
			L	R		opcode				

8-bit `(opcode == 001)`

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit `(opcode == 011 && size == x0)`

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit `(opcode == 101 && size == 00)`

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit `(opcode == 101 && S == 0 && size == 01)`

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 … 23	22	21	20 … 16	15 … 13	12	11 10	9 … 5	4 … 0
0	Q	0 0 1 1 0 1 1	1	0	Rm	x x 1	S	size	Rn	Rt
			L	R		opcode				

8-bit, immediate offset `(Rm == 11111 && opcode == 001)`

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3 //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Cn|CSP>], #3 //  (PSTATE.C64 == '1')
```

8-bit, register offset `(Rm != 11111 && opcode == 001)`

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset `(Rm == 11111 && opcode == 011 && size == x0)`

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6 //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Cn|CSP>], #6 //  (PSTATE.C64 == '1')
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12 //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Cn|CSP>], #12 //  (PSTATE.C64 == '1')
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24 //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Cn|CSP>], #24 //  (PSTATE.C64 == '1')
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
```

```
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

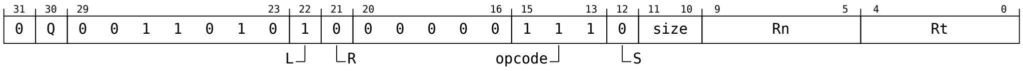
4.3.9 LD3R

Load single 3-element structure and Replicate to all lanes of three registers. This instruction loads a 3-element structure from memory and replicates the structure to all the lanes of the three SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

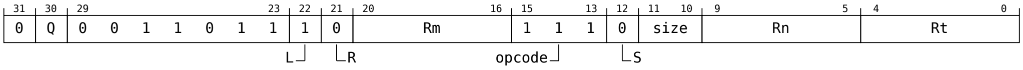


```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index



Immediate offset `(Rm == 11111)`

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset `(Rm != 11111)`

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD3R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"size":

size	<imm>
00	#3
01	#6
10	#12
11	#24

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);          // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);       // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);           // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);             // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.10 LD4 (multiple structures)

Load multiple 4-element structures to four registers. This instruction loads multiple 4-element structures from memory and writes the result to the four SIMD&FP registers, with de-interleaving.

For an example of de-interleaving, see `LD3 (multiple structures)`.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21					16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	size	Rn	Rt
									L							opcode						

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20				16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	1	0	Rm					0	0	0	0	size	Rn	Rt
									L							opcode						

Immediate offset `(Rm == 11111)`

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset `(Rm != 11111)`

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1

modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;
```

```

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.11 LD4 (single structure)

Load single 4-element structure to one lane of four registers. This instruction loads a 4-element structure from memory and writes the result to the corresponding elements of the four SIMD&FP registers without affecting the other bits of the registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21	20				16	15		13	12	11 10	9 5	4 0
0	Q	0	0	1	1	0	1	0	1	1	0	0	0	0	0	x	x	1	S	size	Rn	Rt
									L	R						opcode						

8-bit (opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit (opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit (opcode == 101 && size == 00)

```
LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

LD4 { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

LD4 { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15		13	12	11 10	9 5	4 0
0	Q	0	0	1	1	0	1	1	1	1	Rm	x	x	1	S	size	Rn	Rt
									L	R		opcode						

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4 //  (PSTATE.C64 == '0')

LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Cn|CSP>], #4 //  (PSTATE.C64 == '1')
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')

LD4 { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8 //  (PSTATE.C64 == '0')

LD4 { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Cn|CSP>], #8 //  (PSTATE.C64 == '1')
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
LD4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
LD4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16 //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Cn|CSP>], #16 //  (PSTATE.C64 == '1')
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
LD4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32 //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Cn|CSP>], #32 //  (PSTATE.C64 == '1')
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
LD4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

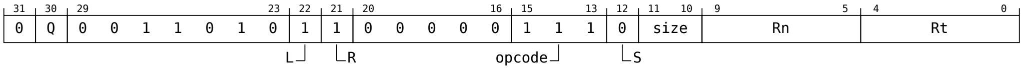
4.3.12 LD4R

Load single 4-element structure and Replicate to all lanes of four registers. This instruction loads a 4-element structure from memory and replicates the structure to all the lanes of the four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

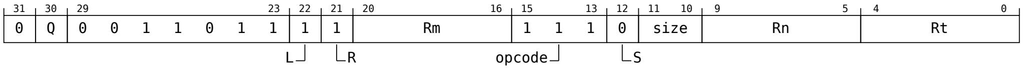


```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index



Immediate offset (Rm == 11111)

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset (Rm != 11111)

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
LD4R  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"size":

size	<imm>
00	#4
01	#8
10	#16
11	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);          // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);       // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);           // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);             // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
```

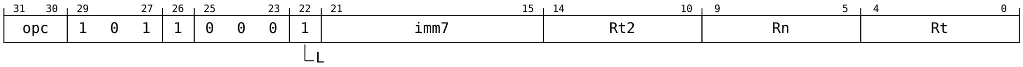
```
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.13 LDNP (SIMD&FP)

Load Pair of SIMD&FP registers, with Non-temporal hint. This instruction loads a pair of SIMD&FP registers from memory, issuing a hint to the memory system that the access is non-temporal. The address that is used for the load is calculated from a base register value and an optional immediate offset.

For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.



32-bit (opc == 00)

```
LDNP  <St1>, <St2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDNP  <St1>, <St2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
LDNP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDNP  <Dt1>, <Dt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
LDNP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDNP  <Qt1>, <Qt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback   = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDNP (SIMD&FP)*.

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.14 LDP (SIMD&FP)

Load Pair of SIMD&FP registers. This instruction loads a pair of SIMD&FP registers from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	1	0	0	1	1	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
LDP  <St1>, <St2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDP  <St1>, <St2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
LDP  <Dt1>, <Dt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDP  <Dt1>, <Dt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
LDP  <Qt1>, <Qt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDP  <Qt1>, <Qt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	1	0	1	1	1	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
LDP  <St1>, <St2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDP  <St1>, <St2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
LDP  <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDP  <Dt1>, <Dt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
LDP  <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDP  <Qt1>, <Qt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	1	0	1	0	1	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
LDP  <St1>, <St2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')

LDP  <St1>, <St2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
LDP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')

LDP  <Dt1>, <Dt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
LDP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')

LDP  <Qt1>, <Qt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback   = FALSE;
boolean postindex = FALSE;
```

For information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see *Architectural Constraints on UNPREDICTABLE behaviors*, and particularly *LDP (SIMD&FP)*.

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.

For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.15 LDR (immediate, SIMD&FP)

Load SIMD&FP Register (immediate offset). This instruction loads an element from memory, and writes the result as a scalar to the SIMD&FP register. The address that is used for the load is calculated from a base register value, a signed immediate offset, and an optional offset that is a multiple of the element size.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 1	0	imm9	0 1	Rn	Rt
				opc					

8-bit (size == 00 && opc == 01)

```
LDR  <Bt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDR  <Bt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 01)

```
LDR  <Ht>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDR  <Ht>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 01)

```
LDR  <St>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDR  <St>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 01)

```
LDR  <Dt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDR  <Dt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 11)

```
LDR  <Qt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')
```

```
LDR  <Qt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 1	0	imm9	1 1	Rn	Rt
				opc					

8-bit (size == 00 && opc == 01)

```
LDR  <Bt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDR  <Bt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 01)

```
LDR  <Ht>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDR  <Ht>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 01)

```
LDR  <St>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDR  <St>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 01)

```
LDR  <Dt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDR  <Dt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 11)

```
LDR  <Qt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')
```

```
LDR  <Qt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29 27	26	25 24	23 22	21 10	9 5	4 0
size	1 1 1	1	0 1	x 1	imm12	Rn	Rt
				opc			

8-bit (size == 00 && opc == 01)

```
LDR  <Bt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDR  <Bt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 01)

```
LDR  <Ht>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDR  <Ht>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 01)

```
LDR  <St>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDR  <St>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 01)

```
LDR  <Dt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDR  <Dt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 11)

```
LDR  <Qt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')
```

```
LDR  <Qt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

VirtualAddress base;

base = BaseReg[n];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.16 LDR (literal, SIMD&FP)

Load SIMD&FP Register (PC-relative literal). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from the PC value and an immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 27	26	25 24	23 5	4 0
opc	0 1 1	1	0 0	imm19	Rt

32-bit (opc == 00)

```
LDR <St>, <label>
```

64-bit (opc == 01)

```
LDR <Dt>, <label>
```

128-bit (opc == 10)

```
LDR <Qt>, <label>
```

```
integer t = UInt(Rt);
integer size;
bits(64) offset;

case opc of
    when '00'
        size = 4;
    when '01'
        size = 8;
    when '10'
        size = 16;
    when '11'
        UNDEFINED;

offset = SignExtend(imm19:'00', 64);
```

Assembler Symbols

<Dt> Is the 64-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be loaded, encoded in the "Rt" field.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, is encoded as "imm19" times 4.

Operation

```
VirtualAddress base = VAFromCapability(PCC);
bits(64) address = VAddress(base) + offset;

bits(size*8) data;

CheckFPAdvSIMDEnabled64();

VACheckAddress(base, address, size, CAP_PERM_LOAD, AccType_VEC);

data = Mem[address, size, AccType_VEC];
V[t] = data;
```

4.3.17 LDR (register, SIMD&FP)

Load SIMD&FP Register (register offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29		27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
size	1	1	1	1	0	0	x	1	1	Rm	option	S	1	0	Rn	Rt
							opc									

8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option != 011)

```
LDR  <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '0')
```

```
LDR  <Bt>, [<Cn|CSP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '1')
```

8-fsreg,LDR-8-fsreg (size == 00 && opc == 01 && option == 011)

```
LDR  <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '0')
```

```
LDR  <Bt>, [<Cn|CSP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '1')
```

16-fsreg,LDR-16-fsreg (size == 01 && opc == 01)

```
LDR  <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
LDR  <Ht>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

32-fsreg,LDR-32-fsreg (size == 10 && opc == 01)

```
LDR  <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
LDR  <St>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

64-fsreg,LDR-64-fsreg (size == 11 && opc == 01)

```
LDR  <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
LDR  <Dt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

128-fsreg,LDR-128-fsreg (size == 00 && opc == 11)

```
LDR  <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')
```

```
LDR  <Qt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

field.

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in"option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

VirtualAddress base;

base = BaseReg[n];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.18 LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled offset). This instruction loads a SIMD&FP register from memory. The address that is used for the load is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 1	0	imm9	0 0	Rn	Rt
				opc					

8-bit (size == 00 && opc == 01)

```
LDUR  <Bt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDUR  <Bt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 01)

```
LDUR  <Ht>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDUR  <Ht>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 01)

```
LDUR  <St>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDUR  <St>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 01)

```
LDUR  <Dt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDUR  <Dt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 11)

```
LDUR  <Qt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')

LDUR  <Qt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

VirtualAddress base;

base = BaseReg[n];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.19 ST1 (multiple structures)

Store multiple single-element structures from one, two, three, or four registers. This instruction stores elements to memory from one, two, three, or four SIMD&FP registers, without interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21					16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	x	x	1	x	size	Rn	Rt
									L									opcode				

One register (opcode == 0111)

```
ST1  { <Vt>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

Two registers (opcode == 1010)

```
ST1  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

Three registers (opcode == 0110)

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

Four registers (opcode == 0010)

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	0	0	Rm	x	x	1	x	size	Rn	Rt
									L					opcode				

One register, immediate offset (Rm == 11111 && opcode == 0111)

```
ST1  { <Vt>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

One register, register offset (Rm != 11111 && opcode == 0111)

```
ST1  { <Vt>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

Two registers, immediate offset (Rm == 11111 && opcode == 1010)

```
ST1  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Two registers, register offset `(Rm != 11111 && opcode == 1010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

Three registers, immediate offset `(Rm == 11111 && opcode == 0110)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Three registers, register offset `(Rm != 11111 && opcode == 0110)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

Four registers, immediate offset `(Rm == 11111 && opcode == 0010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Four registers, register offset `(Rm != 11111 && opcode == 0010)`

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	1D
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the one register, immediate offset variant: is the post-index immediate offset, encoded

in"Q":

Q	<imm>
0	#8
1	#16

For the two registers, immediate offset variant: is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#16
1	#32

For the three registers, immediate offset variant: is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#24
1	#48

For the four registers, immediate offset variant: is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4  (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1  (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3  (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1  (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1  (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2  (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1  (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.20 ST1 (single structure)

Store a single-element structure from one lane of one register. This instruction stores the specified element of a SIMD&FP register to memory.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 0	0	0	0 0 0 0 0	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit (opcode == 000)

```
ST1  { <Vt>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST1  { <Vt>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit (opcode == 010 && size == x0)

```
ST1  { <Vt>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST1  { <Vt>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit (opcode == 100 && size == 00)

```
ST1  { <Vt>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST1  { <Vt>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
ST1  { <Vt>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST1  { <Vt>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 1	0	0	Rm	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST1  { <Vt>.B }[<index>], [<Xn|SP>], #1 //  (PSTATE.C64 == '0')

ST1  { <Vt>.B }[<index>], [<Cn|CSP>], #1 //  (PSTATE.C64 == '1')
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST1  { <Vt>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')

ST1  { <Vt>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST1  { <Vt>.H }[<index>], [<Xn|SP>], #2 //  (PSTATE.C64 == '0')

ST1  { <Vt>.H }[<index>], [<Cn|CSP>], #2 //  (PSTATE.C64 == '1')
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST1  { <Vt>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (`Rm == 11111 && opcode == 100 && size == 00`)

```
ST1  { <Vt>.S }[<index>], [<Xn|SP>], #4 //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.S }[<index>], [<Cn|CSP>], #4 //  (PSTATE.C64 == '1')
```

32-bit, register offset (`Rm != 11111 && opcode == 100 && size == 00`)

```
ST1  { <Vt>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (`Rm == 11111 && opcode == 100 && S == 0 && size == 01`)

```
ST1  { <Vt>.D }[<index>], [<Xn|SP>], #8 //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.D }[<index>], [<Cn|CSP>], #8 //  (PSTATE.C64 == '1')
```

64-bit, register offset (`Rm != 11111 && opcode == 100 && S == 0 && size == 01`)

```
ST1  { <Vt>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST1  { <Vt>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);          // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.21 ST2 (multiple structures)

Store multiple 2-element structures from two registers. This instruction stores multiple 2-element structures from two SIMD&FP registers to memory, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21					16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	size	Rn	Rt
									L							opcode						

```
ST2  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	0	0	Rm	1	0	0	0	size	Rn	Rt
									L			opcode						

Immediate offset (Rm == 11111)

```
ST2  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset (Rm != 11111)

```
ST2  { <Vt>.<T>, <Vt2>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.<T>, <Vt2>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#16
1	#32

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.22 ST2 (single structure)

Store single 2-element structure from one lane of two registers. This instruction stores a 2-element structure to memory from corresponding elements of two SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 0	0	1	0 0 0 0 0	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit (opcode == 000)

```
ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit (opcode == 010 && size == x0)

```
ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit (opcode == 100 && size == 00)

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit (opcode == 100 && S == 0 && size == 01)

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')

ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 1	0	1	Rm	x x 0	S	size	Rn	Rt
			L	R		opcode				

8-bit, immediate offset (Rm == 11111 && opcode == 000)

```
ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], #2 //  (PSTATE.C64 == '0')

ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Cn|CSP>], #2 //  (PSTATE.C64 == '1')
```

8-bit, register offset (Rm != 11111 && opcode == 000)

```
ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')

ST2  { <Vt>.B, <Vt2>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset (Rm == 11111 && opcode == 010 && size == x0)

```
ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], #4 //  (PSTATE.C64 == '0')

ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Cn|CSP>], #4 //  (PSTATE.C64 == '1')
```

16-bit, register offset (Rm != 11111 && opcode == 010 && size == x0)

```
ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.H, <Vt2>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (Rm == 11111 && opcode == 100 && size == 00)

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], #8 //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Cn|CSP>], #8 //  (PSTATE.C64 == '1')
```

32-bit, register offset (Rm != 11111 && opcode == 100 && size == 00)

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.S, <Vt2>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (Rm == 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], #16 //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Cn|CSP>], #16 //  (PSTATE.C64 == '1')
```

64-bit, register offset (Rm != 11111 && opcode == 100 && S == 0 && size == 01)

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST2  { <Vt>.D, <Vt2>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
```

```
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.23 ST3 (multiple structures)

Store multiple 3-element structures from three registers. This instruction stores multiple 3-element structures to memory from three SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21					16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	size	Rn	Rt
									L							opcode						

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	0	0	Rm	0	1	0	0	size	Rn	Rt
									L			opcode						

Immediate offset (Rm == 11111)

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset (Rm != 11111)

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#24
1	#48

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.24 ST3 (single structure)

Store single 3-element structure from one lane of three registers. This instruction stores a 3-element structure to memory from corresponding elements of three SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 0	0	0	0 0 0 0 0	x x 1	S	size	Rn	Rt
			L	R		opcode				

8-bit (opcode == 001)

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit (opcode == 011 && size == x0)

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit (opcode == 101 && size == 00)

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 1	0	0	Rm	x x 1	S	size	Rn	Rt
			L	R		opcode				

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], #3 //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Cn|CSP>], #3 //  (PSTATE.C64 == '1')
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.B, <Vt2>.B, <Vt3>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], #6 //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Cn|CSP>], #6 //  (PSTATE.C64 == '1')
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.H, <Vt2>.H, <Vt3>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], #12 //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Cn|CSP>], #12 //  (PSTATE.C64 == '1')
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.S, <Vt2>.S, <Vt3>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], #24 //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Cn|CSP>], #24 //  (PSTATE.C64 == '1')
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST3  { <Vt>.D, <Vt2>.D, <Vt3>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.25 ST4 (multiple structures)

Store multiple 4-element structures from four registers. This instruction stores multiple 4-element structures to memory from four SIMD&FP registers, with interleaving. Every element of each register is stored.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21					16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	size	Rn	Rt
									L							opcode						

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15			12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	0	0	Rm	0	0	0	0	size	Rn	Rt
									L			opcode						

Immediate offset (Rm == 11111)

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <imm> //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <imm> //  (PSTATE.C64 == '1')
```

Register offset (Rm != 11111)

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.<T>, <Vt2>.<T>, <Vt3>.<T>, <Vt4>.<T>}, [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<T> Is an arrangement specifier, encoded in"size:Q":

size	Q	<T>
00	0	8B
00	1	16B
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> Is the post-index immediate offset, encoded in"Q":

Q	<imm>
0	#32
1	#64

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << UInt(size);
integer elements = datasize DIV esize;

integer rpt;    // number of iterations
integer selem;  // structure elements

case opcode of
    when '0000' rpt = 1; selem = 4;     // LD/ST4 (4 registers)
    when '0010' rpt = 4; selem = 1;     // LD/ST1 (4 registers)
    when '0100' rpt = 1; selem = 3;     // LD/ST3 (3 registers)
    when '0110' rpt = 3; selem = 1;     // LD/ST1 (3 registers)
    when '0111' rpt = 1; selem = 1;     // LD/ST1 (1 register)
    when '1000' rpt = 1; selem = 2;     // LD/ST2 (2 registers)
    when '1010' rpt = 2; selem = 1;     // LD/ST1 (2 registers)
    otherwise UNDEFINED;

// .1D format only permitted with LD1 & ST1
if size:Q == '110' && selem != 1 then UNDEFINED;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(datasize) rval;
integer tt;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if memop == MemOp_LOAD then
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, rpt * elements * selem * ebytes, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
for r = 0 to rpt-1
    for e = 0 to elements-1
        tt = (t + r) MOD 32;
        for s = 0 to selem-1
            rval = V[tt];
            if memop == MemOp_LOAD then
                Elem[rval, e, esize] = Mem[address + offs, ebytes, AccType_VEC];
                V[tt] = rval;
            else // memop == MemOp_STORE
                Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, e, esize];
            offs = offs + ebytes;
            tt = (tt + 1) MOD 32;

if wback then
```

```
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.26 ST4 (single structure)

Store single 4-element structure from one lane of four registers. This instruction stores a 4-element structure to memory from corresponding elements of four SIMD&FP registers.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 2 classes: No offset and Post-index

No offset

31	30	29						23	22	21	20				16	15		13	12	11 10	9 5	4 0
0	Q	0	0	1	1	0	1	0	0	1	0	0	0	0	0	x	x	1	S	size	Rn	Rt
									L	R						opcode						

8-bit (opcode == 001)

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

16-bit (opcode == 011 && size == x0)

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

32-bit (opcode == 101 && size == 00)

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

64-bit (opcode == 101 && S == 0 && size == 01)

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = integer UNKNOWN;
boolean wback = FALSE;
```

Post-index

31	30	29						23	22	21	20 16	15		13	12	11 10	9 5	4 0
0	Q	0	0	1	1	0	1	1	0	1	Rm	x	x	1	S	size	Rn	Rt
									L	R		opcode						

8-bit, immediate offset (Rm == 11111 && opcode == 001)

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], #4 //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Cn|CSP>], #4 //  (PSTATE.C64 == '1')
```

8-bit, register offset (Rm != 11111 && opcode == 001)

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.B, <Vt2>.B, <Vt3>.B, <Vt4>.B }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

16-bit, immediate offset (Rm == 11111 && opcode == 011 && size == x0)

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], #8 //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Cn|CSP>], #8 //  (PSTATE.C64 == '1')
```

16-bit, register offset (Rm != 11111 && opcode == 011 && size == x0)

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.H, <Vt2>.H, <Vt3>.H, <Vt4>.H }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

32-bit, immediate offset (Rm == 11111 && opcode == 101 && size == 00)

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], #16 //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Cn|CSP>], #16 //  (PSTATE.C64 == '1')
```

32-bit, register offset (Rm != 11111 && opcode == 101 && size == 00)

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.S, <Vt2>.S, <Vt3>.S, <Vt4>.S }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

64-bit, immediate offset (Rm == 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], #32 //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Cn|CSP>], #32 //  (PSTATE.C64 == '1')
```

64-bit, register offset (Rm != 11111 && opcode == 101 && S == 0 && size == 01)

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Xn|SP>], <Xm> //  (PSTATE.C64 == '0')
```

```
ST4  { <Vt>.D, <Vt2>.D, <Vt3>.D, <Vt4>.D }[<index>], [<Cn|CSP>], <Xm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
boolean wback = TRUE;
```

Assembler Symbols

<Vt> Is the name of the first or only SIMD&FP register to be transferred, encoded in the "Rt" field.

<Vt2> Is the name of the second SIMD&FP register to be transferred, encoded as "Rt" plus 1 modulo 32.

<Vt3> Is the name of the third SIMD&FP register to be transferred, encoded as "Rt" plus 2 modulo 32.

<Vt4> Is the name of the fourth SIMD&FP register to be transferred, encoded as "Rt" plus 3 modulo 32.

<index> For the 8-bit variant: is the element index, encoded in "Q:S:size".

For the 16-bit variant: is the element index, encoded in "Q:S:size<1>".

For the 32-bit variant: is the element index, encoded in "Q:S".

For the 64-bit variant: is the element index, encoded in "Q".

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Xm> Is the 64-bit name of the general-purpose post-index register, excluding XZR, encoded in the "Rm" field.

Shared Decode

```
integer scale = UInt(opcode<2:1>);
integer selem = UInt(opcode<0>:R) + 1;
boolean replicate = FALSE;
integer index;

case scale of
    when 3
        // load and replicate
        if L == '0' || S == '1' then UNDEFINED;
        scale = UInt(size);
        replicate = TRUE;
    when 0
        index = UInt(Q:S:size);         // B[0-15]
    when 1
        if size<0> == '1' then UNDEFINED;
        index = UInt(Q:S:size<1>);      // H[0-7]
    when 2
        if size<1> == '1' then UNDEFINED;
        if size<0> == '0' then
            index = UInt(Q:S);          // S[0-3]
        else
            if S == '1' then UNDEFINED;
            index = UInt(Q);            // D[0-1]
            scale = 3;

MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = if Q == '1' then 128 else 64;
integer esize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(64) address;
bits(64) offs;
bits(128) rval;
bits(esize) element;
constant integer ebytes = esize DIV 8;

VirtualAddress base = BaseReg[n];
address = VAddress(base);
if replicate || memop == MemOp_LOAD then
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_LOAD, AccType_VEC);
else
    VACheckAddress(base, address, ebytes * selem, CAP_PERM_STORE, AccType_VEC);

offs = Zeros();
if replicate then
    // load and replicate to all elements
    for s = 0 to selem-1
        element = Mem[address + offs, ebytes, AccType_VEC];
        // replicate to fill 128- or 64-bit register
        V[t] = Replicate(element, datasize DIV esize);
        offs = offs + ebytes;
        t = (t + 1) MOD 32;
else
    // load/store one element per register
    for s = 0 to selem-1
        rval = V[t];
        if memop == MemOp_LOAD then
            // insert into one lane of 128-bit register
            Elem[rval, index, esize] = Mem[address + offs, ebytes, AccType_VEC];
            V[t] = rval;
        else // memop == MemOp_STORE
            // extract from one lane of 128-bit register
            Mem[address + offs, ebytes, AccType_VEC] = Elem[rval, index, esize];
        offs = offs + ebytes;
        t = (t + 1) MOD 32;

if wback then
    if m != 31 then
        offs = X[m];
    BaseReg[n] = VAAdd(base, offs);
```

4.3.27 STNP (SIMD&FP)

Store Pair of SIMD&FP registers, with Non-temporal hint. This instruction stores a pair of SIMD&FP registers to memory, issuing a hint to the memory system that the access is non-temporal. The address used for the store is calculated from an address from a base register value and an immediate offset. For information about non-temporal pair instructions, see *Load/Store SIMD and Floating-point Non-temporal pair*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	1	0	0	0	0	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
STNP  <St1>, <St2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STNP  <St1>, <St2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
STNP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STNP  <Dt1>, <Dt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
STNP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STNP  <Qt1>, <Qt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback     = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
AccType acctype = AccType_VECSTREAM;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.28 STP (SIMD&FP)

Store Pair of SIMD&FP registers. This instruction stores a pair of SIMD&FP registers to memory. The address used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Signed offset

Post-index

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	1	0	0	1	0	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
STP  <St1>, <St2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')

STP  <St1>, <St2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
STP  <Dt1>, <Dt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')

STP  <Dt1>, <Dt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
STP  <Qt1>, <Qt2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')

STP  <Qt1>, <Qt2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = TRUE;
```

Pre-index

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	1	0	1	1	0	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
STP  <St1>, <St2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')

STP  <St1>, <St2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
STP  <Dt1>, <Dt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')

STP  <Dt1>, <Dt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
STP  <Qt1>, <Qt2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')

STP  <Qt1>, <Qt2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback  = TRUE;
boolean postindex = FALSE;
```

Signed offset

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	1	0	1	0	0	imm7	Rt2	Rn	Rt
								L				

32-bit (opc == 00)

```
STP  <St1>, <St2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STP  <St1>, <St2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

64-bit (opc == 01)

```
STP  <Dt1>, <Dt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STP  <Dt1>, <Dt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

128-bit (opc == 10)

```
STP  <Qt1>, <Qt2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STP  <Qt1>, <Qt2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback  = FALSE;
boolean postindex = FALSE;
```

Assembler Symbols

<Dt1> Is the 64-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt2> Is the 64-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<Qt1> Is the 128-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt2> Is the 128-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

<St1> Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Rt" field.

<St2> Is the 32-bit name of the second SIMD&FP register to be transferred, encoded in the "Rt2" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<imm> For the 32-bit post-index and 32-bit pre-index variant: is the signed immediate byte offset, a multiple of 4 in the range -256 to 252, encoded in the "imm7" field as <imm>/4.

For the 32-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 4 in the range -256 to 252, defaulting to 0 and encoded in the "imm7" field as <imm>/4.

For the 64-bit post-index and 64-bit pre-index variant: is the signed immediate byte offset, a multiple of 8 in the range -512 to 504, encoded in the "imm7" field as <imm>/8.

For the 64-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 8 in the range -512 to 504, defaulting to 0 and encoded in the "imm7" field as <imm>/8.

For the 128-bit post-index and 128-bit pre-index variant: is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field as <imm>/16.

For the 128-bit signed offset variant: is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0 and encoded in the "imm7" field as <imm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer t2 = UInt(Rt2);
```

```
AccType acctype = AccType_VEC;
MemOp memop = if L == '1' then MemOp_LOAD else MemOp_STORE;
if opc == '11' then UNDEFINED;
integer scale = 2 + UInt(opc);
integer datasize = 8 << scale;
bits(64) offset = LSL(SignExtend(imm7, 64), scale);
```

Operation

```
CheckFPAdvSIMDEnabled64();

bits(datasize) data1;
bits(datasize) data2;
constant integer dbytes = datasize DIV 8;
boolean rt_unknown = FALSE;

if memop == MemOp_LOAD && t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

VirtualAddress base = BaseReg[n];
bits(64) address = VAddress(base);
if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_STORE, acctype);
        data1 = V[t];
        data2 = V[t2];
        Mem[address + 0     , dbytes, acctype] = data1;
        Mem[address + dbytes, dbytes, acctype] = data2;

    when MemOp_LOAD
        VACheckAddress(base, address, dbytes * 2, CAP_PERM_LOAD, acctype);
        data1 = Mem[address + 0     , dbytes, acctype];
        data2 = Mem[address + dbytes, dbytes, acctype];
        if rt_unknown then
            data1 = bits(datasize) UNKNOWN;
            data2 = bits(datasize) UNKNOWN;
        V[t]  = data1;
        V[t2] = data2;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.29 STR (immediate, SIMD&FP)

Store SIMD&FP register (immediate offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

It has encodings from 3 classes: Post-index , Pre-index and Unsigned offset

Post-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 0	0	imm9	0 1	Rn	Rt
				opc					

8-bit (size == 00 && opc == 00)

```
STR  <Bt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')

STR  <Bt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 00)

```
STR  <Ht>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')

STR  <Ht>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 00)

```
STR  <St>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')

STR  <St>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 00)

```
STR  <Dt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')

STR  <Dt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 10)

```
STR  <Qt>, [<Xn|SP>], #<simm> //  (PSTATE.C64 == '0')

STR  <Qt>, [<Cn|CSP>], #<simm> //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = TRUE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Pre-index

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 0	0	imm9	1 1	Rn	Rt
				opc					

8-bit (size == 00 && opc == 00)

```
STR  <Bt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')

STR  <Bt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 00)

```
STR  <Ht>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')

STR  <Ht>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 00)

```
STR  <St>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')

STR  <St>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 00)

```
STR  <Dt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')

STR  <Dt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 10)

```
STR  <Qt>, [<Xn|SP>, #<simm>]! //  (PSTATE.C64 == '0')

STR  <Qt>, [<Cn|CSP>, #<simm>]! //  (PSTATE.C64 == '1')
```

```
boolean wback = TRUE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Unsigned offset

31 30	29		27	26	25	24	23	22	21 10	9 5	4 0
size	1	1	1	1	0	1	x	0	imm12	Rn	Rt
							└opc				

8-bit (size == 00 && opc == 00)

```
STR  <Bt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')

STR  <Bt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 00)

```
STR  <Ht>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')

STR  <Ht>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 00)

```
STR  <St>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')

STR  <St>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 00)

```
STR  <Dt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')

STR  <Dt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 10)

```
STR  <Qt>, [<Xn|SP>{, #<pimm>}] //  (PSTATE.C64 == '0')

STR  <Qt>, [<Cn|CSP>{, #<pimm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = LSL(ZeroExtend(imm12, 64), scale);
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the signed immediate byte offset, in the range -256 to 255, encoded in the "imm9" field.

<pimm> For the 8-bit variant: is the optional positive immediate byte offset, in the range 0 to 4095, defaulting to 0 and encoded in the "imm12" field.

For the 16-bit variant: is the optional positive immediate byte offset, a multiple of 2 in the range 0 to 8190, defaulting to 0 and encoded in the "imm12" field as <pimm>/2.

For the 32-bit variant: is the optional positive immediate byte offset, a multiple of 4 in the range 0 to 16380, defaulting to 0 and encoded in the "imm12" field as <pimm>/4.

For the 64-bit variant: is the optional positive immediate byte offset, a multiple of 8 in the range 0 to 32760, defaulting to 0 and encoded in the "imm12" field as <pimm>/8.

For the 128-bit variant: is the optional positive immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0 and encoded in the "imm12" field as <pimm>/16.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

VirtualAddress base;

base = BaseReg[n];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.30 STR (register, SIMD&FP)

Store SIMD&FP register (register offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an offset register value. The offset can be optionally shifted and extended.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29		27	26	25	24	23	22	21	20 16	15 13	12	11	10	9 5	4 0
size	1	1	1	1	0	0	x	0	1	Rm	option	S	1	0	Rn	Rt
							opc									

8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option != 011)

```
STR  <Bt>, [<Xn|SP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '0')

STR  <Bt>, [<Cn|CSP>, (<Wm>|<Xm>), <extend>{<amount>}] //  (PSTATE.C64 == '1')
```

8-fsreg,STR-8-fsreg (size == 00 && opc == 00 && option == 011)

```
STR  <Bt>, [<Xn|SP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '0')

STR  <Bt>, [<Cn|CSP>, <Xm>{, LSL <amount>}] //  (PSTATE.C64 == '1')
```

16-fsreg,STR-16-fsreg (size == 01 && opc == 00)

```
STR  <Ht>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')

STR  <Ht>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

32-fsreg,STR-32-fsreg (size == 10 && opc == 00)

```
STR  <St>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')

STR  <St>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

64-fsreg,STR-64-fsreg (size == 11 && opc == 00)

```
STR  <Dt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')

STR  <Dt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

128-fsreg,STR-128-fsreg (size == 00 && opc == 10)

```
STR  <Qt>, [<Xn|SP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '0')

STR  <Qt>, [<Cn|CSP>, (<Wm>|<Xm>){, <extend>{<amount>}}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
if option<1> == '0' then UNDEFINED;              // sub-word index
ExtendType extend_type = DecodeRegExtend(option);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

field.

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<Wm> When option<0> is set to 0, is the 32-bit name of the general-purpose index register, encoded in the "Rm" field.

<Xm> When option<0> is set to 1, is the 64-bit name of the general-purpose index register, encoded in the "Rm" field.

<extend> For the 8-bit variant: is the index extend specifier, encoded in"option":

option	<extend>
010	UXTW
110	SXTW
111	SXTX

For the 128-bit, 16-bit, 32-bit and 64-bit variant: is the index extend/shift specifier, defaulting to LSL, and which must be omitted for the LSL option when <amount> is omitted. encoded in"option":

option	<extend>
010	UXTW
011	LSL
110	SXTW
111	SXTX

<amount> For the 8-bit variant: is the index shift amount, it must be #0, encoded in "S" as 0 if omitted, or as 1 if present.

For the 16-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#1

For the 32-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#2

For the 64-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#3

For the 128-bit variant: is the index shift amount, optional only when <extend> is not LSL. Where it is permitted to be optional, it defaults to #0. It is encoded in"S":

S	<amount>
0	#0
1	#4

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
integer m = UInt(Rm);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
bits(64) offset = ExtendReg(m, extend_type, shift);

CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

VirtualAddress base;

base = BaseReg[n];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.3.31 STUR (SIMD&FP)

Store SIMD&FP register (unscaled offset). This instruction stores a single SIMD&FP register to memory. The address that is used for the store is calculated from a base register value and an optional immediate offset.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	1	0 0	x 0	0	imm9	0 0	Rn	Rt
				opc					

8-bit (size == 00 && opc == 00)

```
STUR  <Bt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STUR  <Bt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

16-bit (size == 01 && opc == 00)

```
STUR  <Ht>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STUR  <Ht>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

32-bit (size == 10 && opc == 00)

```
STUR  <St>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STUR  <St>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

64-bit (size == 11 && opc == 00)

```
STUR  <Dt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STUR  <Dt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

128-bit (size == 00 && opc == 10)

```
STUR  <Qt>, [<Xn|SP>{, #<simm>}] //  (PSTATE.C64 == '0')
```

```
STUR  <Qt>, [<Cn|CSP>{, #<simm>}] //  (PSTATE.C64 == '1')
```

```
boolean wback = FALSE;
boolean postindex = FALSE;
integer scale = UInt(opc<1>:size);
if scale > 4 then UNDEFINED;
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

<Cn|CSP> Is the name of the capability register or capability stack pointer holding the base address, encoded in the "Rn" field.

<simm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0 and encoded in the "imm9" field.

Shared Decode

```
integer n = UInt(Rn);
integer t = UInt(Rt);
AccType acctype = AccType_VEC;
MemOp memop = if opc<0> == '1' then MemOp_LOAD else MemOp_STORE;
integer datasize = 8 << scale;
```

Operation

```
CheckFPAdvSIMDEnabled64();
bits(64) address;
bits(datasize) data;

VirtualAddress base;

base = BaseReg[n];
address = VAddress(base);

if ! postindex then
    address = address + offset;

case memop of
    when MemOp_STORE
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_STORE, acctype);
        data = V[t];
        Mem[address, datasize DIV 8, acctype] = data;

    when MemOp_LOAD
        VACheckAddress(base, address, datasize DIV 8, CAP_PERM_LOAD, acctype);
        data = Mem[address, datasize DIV 8, acctype];
        V[t] = data;

if wback then
    base = VAAdd(base,offset);

    BaseReg[n] = base;
```

4.4 New instructions

4.4.1 ADD (extended register)

Add (extended register) adds a Capability register value field and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination Capability register value field. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword. If the result is not representable the destination Capability register tag is cleared. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31										21	20 16	15 13	12 10	9 5	4 0
1	1	0	0	0	0	1	0	1	0	1	Rm	option	imm3	Cn	Cd

```
ADD     <Cd|CSP>, <Cn|CSP>, <Xm>{, <extend>#<amount>}
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
ExtendType extend_type = DecodeRegExtend(option);
integer shift = UInt(imm3);
if shift > 4 then UNDEFINED;
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"option":

option	<extend>
000	UXTB
001	UXTH
010	UXTW
011	UXTX
100	SXTB
101	SXTH
110	SXTW
111	SXTX

<amount> Is the optional unsigned immediate operand, in the range 0 to 4, defaulting to 0, encoded in the "imm3" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64) operand2 = ExtendReg(m, extend_type, shift);
Capability result = CapAdd(operand1, operand2);

if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.2 ADD (immediate)

Add (immediate) copies a capability from the source Capability register to the destination Capability register with an optionally shifted immediate value added to the value field. If the result is not representable the destination Capability register tag is cleared. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31							24	23	22	21 10	9 5	4 0
0	0	0	0	0	0	1	0	0	sh	imm12	Cn	Cd
								A				

```
ADD     <Cd|CSP>, <Cn|CSP>, #<imm>{, LSL <amount>}
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
bits(64) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, 64);
    when '1' imm = ZeroExtend(imm12 : Zeros(12), 64);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<imm> Is the unsigned immediate operand, in the range 0 to 4095, encoded in the "imm12" field.

<amount> Is the index shift amount, encoded in"sh":

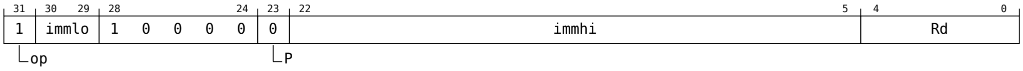
sh	<amount>
0	#0
1	#12

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
integer    operand2 = UInt(imm);

Capability result = CapAdd(operand1, operand2);

if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.3 ADRDP

Form DDC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits to the DDC value with the bottom 12 bits masked out to form a DCC-relative address and writes the result to the destination register. This description only applies in C64.



```
ADRDP   <Cd>, <label>
```

```
integer d = UInt(Rd);
bits(64) imm;

if IsInC64() then
    if P == '1' then
        imm = SignExtend(immhi:immlo:Zeros(12), 64);
    else
        imm = ZeroExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(P:immhi:immlo:Zeros(12), 64);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Rd" field.

<label> Is the program label whose 4KB page address is to be calculated, in the range +/-2GB, encoded in "immhi:immlo".

Operation

```
if IsInC64() then
    Capability addr;
    if  P == '0' then
        if CCTLR[].ADRDPB == '1' then
            addr = C[28];
        else
            addr = DDC[];
    else
        addr = PCC[];

    bits(64) newvalue = CapGetValue(addr) AND NOT(ZeroExtend(Ones(12),64));
    bits(64) offset   = newvalue - CapGetValue(addr) + imm;

    Capability result = CapAdd(addr,offset);

    if CapIsSealed(addr) then
        result = CapWithTagClear(result);

    C[d] = result;
else
    bits(64) addr;
    if CCTLR[].PCCBO == '1' then
        addr = CapGetOffset(PCC[]);
    else
        addr = CapGetValue(PCC[]);

    addr<11:0> = Zeros(12);

    X[d] = addr + imm;
```

4.4.4 ADRP

Form PCC-relative address to 4KB page adds an immediate value that is shifted left by 12 bits to the PCC value with the bottom 12 bits masked out to form a PCC-relative address and writes the result to the destination register. This description only applies in C64.

31	30 29	28 24	23	22 5	4 0
1	immlo	1 0 0 0 0	1	immhi	Rd
op			P		

```
ADRP    <Cd>, <label>
```

```
integer d = UInt(Rd);
bits(64) imm;

if IsInC64() then
    if P == '1' then
        imm = SignExtend(immhi:immlo:Zeros(12), 64);
    else
        imm = ZeroExtend(immhi:immlo:Zeros(12), 64);
else
    imm = SignExtend(P:immhi:immlo:Zeros(12), 64);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Rd" field.

<label> Is the program label whose 4KB page address is to be calculated, in the range +/-2GB, encoded in "immhi:immlo".

Operation

```
if IsInC64() then
    Capability addr;
    if  P == '0' then
        if CCTLR[].ADRDPB == '1' then
            addr = C[28];
        else
            addr = DDC[];
    else
        addr = PCC[];

    bits(64) newvalue = CapGetValue(addr) AND NOT(ZeroExtend(Ones(12),64));
    bits(64) offset   = newvalue - CapGetValue(addr) + imm;

    Capability result = CapAdd(addr,offset);

    if CapIsSealed(addr) then
        result = CapWithTagClear(result);

    C[d] = result;
else
    bits(64) addr;
    if CCTLR[].PCCBO == '1' then
        addr = CapGetOffset(PCC[]);
    else
        addr = CapGetValue(PCC[]);

    addr<11:0> = Zeros(12);

    X[d] = addr + imm;
```

4.4.5 ALIGND

Align Down rounds the value field of the source Capability register down to a two to the power of the immediate value boundary and writes the result to the destination Capability register. If the result is not representable the destination Capability register tag is cleared. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31 21	20 15	14	13 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	imm6	0	0 1 1 0	Cn	Cd
		U			

```
ALIGND  <Cd|CSP>, <Cn|CSP>, #<imm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer align = UInt(imm6);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<imm> Is the unsigned immediate operand, in the range 0 to 63, encoded in the "imm6" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];

bits(64) newvalue = CapGetValue(operand) AND NOT(ZeroExtend(Ones(align),64));
Capability result = CapSetValue(operand, newvalue);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.6 ALIGNU

Align Up rounds the value field of the source Capability register up to a two to the power of the immediate value boundary and writes the result to the destination Capability register. If the result is not representable the destination Capability register tag is cleared. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31 21	20 15	14	13 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	imm6	1	0 1 1 0	Cn	Cd
		U			

```
ALIGNU  <Cd|CSP>, <Cn|CSP>, #<imm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer align = UInt(imm6);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<imm> Is the unsigned immediate operand, in the range 0 to 63, encoded in the "imm6" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];

bits(65) m = ZeroExtend(Ones(align),65);
bits(65) newvalue = (ZeroExtend(CapGetValue(operand),65) + m) AND NOT(m);
Capability result = CapSetValue(operand, newvalue<63:0>);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.7 BICFLGS (immediate)

Bitwise Bit Clear (immediate) on flags field performs a bitwise AND of the flags field of a capability and the complement of an immediate value and writes the result to the flags field of the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31										21	20 13	12	11	10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	imm8	0	0	0	Cn	Cd

```
BICFLGS <Cd|CSP>, <Cn|CSP>, #<imm>
```

```
integer n = UInt(Cn);
integer d = UInt(Cd);
bits(8) mask = imm8;
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];

bits(64) oldvalue = CapGetValue(operand);
bits(8)  newflags = oldvalue<63:56> AND NOT mask;
bits(64) newvalue = newflags : oldvalue<55:0>;

Capability result = CapSetFlags(operand,newvalue);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.8 BICFLGS (register)

Bitwise Bit Clear on flags field performs a bitwise AND of the flags field of a capability and the complement of bits 63 to 56 of a register value and writes the result to the flags field of the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31										21	20–16	15	14	13			10	9–5	4–0
1	1	0	0	0	0	1	0	1	1	0	Rm	0	0	1	0	1	0	Cn	Cd
												opc							

```
BICFLGS <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64)   mask    = X[m];

bits(64) oldvalue = CapGetValue(operand);
bits(8)  newflags = oldvalue<63:56> AND NOT mask<63:56>;
bits(64) newvalue = newflags : oldvalue<55:0>;

Capability result = CapSetFlags(operand,newvalue);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.9 BLR (indirect)

Branch with Link to capability Register calls a subroutine at an address in the source register, setting C30 to PCC+4.

31	30	29									20	19				15	14	13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	0	1	1	0	0	Cn	0	0	0	0	0
																	opc<1>	opc<0>									

```
BLR     <Cn>
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDCALL;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();
Capability target = C[n];

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);

integer linkoffset = 4;
Capability link;

if IsInC64() then
    linkoffset = linkoffset + 1;

link = CapAdd(PCC[], linkoffset);

if CCTLR[].SBL == '1' then
    link = CapSetObjectType(link, CAP_SEAL_TYPE_RB);

C[30] = link;
BranchXToCapability(target, branch_type);
```

4.4.10 BLR (memory indirect)

Unseal load, branch and link loads a capability and an offset, derives, unseals, and branches to the destination Capability register, setting C30 to PCC+4.

31	30	29									20	19 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	1	imm7	1	0	0	Cn	0	0	0	0	1

```
BLR     [<Cn|CSP>, #<imm>]
```

```
integer n = UInt(Cn);
bits(64) offset = SignExtend(imm7:'0000',64);
BranchType branch_type = BranchType_INDCALL;
```

Assembler Symbols

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Cn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

Capability     base;
Capability     target;

if n == 31 then
    CheckSPAlignment();
    base = CSP[];
else
    base = C[n];

integer linkoffset = 4;
Capability link;

if IsInC64() then
    linkoffset = linkoffset + 1;

link = CapAdd(PCC[], linkoffset);

if CCTLR[].SBL == '1' then
    link = CapSetObjectType(link, CAP_SEAL_TYPE_RB);

// When C29 is used, the unsealed capability is written back to C29.
if n == 29 then
    if CapIsTagSet(base) && CapIsSealed(base) &&
       CapGetObjectType(base) == CAP_SEAL_TYPE_LB then
        base = CapUnseal(base);

    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase) + offset;

    VACheckAddress(vabase,addr,CAPABILITY_DBYTES,CAP_PERM_LOAD,AccType_NORMAL);
    target     = MemC[addr,AccType_NORMAL];
    target     = CapSquashPostLoadCap(target,vabase);

    C[29] = base;
    C[30] = link;
else
    boolean        wb_unknown = FALSE;

    if n == 30 then
        Constraint c = ConstrainUnpredictable(Unpredictable_LINKBASEOVERLAPLD);
        assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
        case c of
            when Constraint_UNKNOWN    wb_unknown = TRUE;     // writeback is UNKNOWN
            when Constraint_UNDEF      UNDEFINED;
            when Constraint_NOP        EndOfInstruction();

    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase) + offset;

    VACheckAddress(vabase,addr,CAPABILITY_DBYTES,CAP_PERM_LOAD,AccType_NORMAL);
    target    = MemC[addr,AccType_NORMAL];
    target    = CapSquashPostLoadCap(target,vabase);

    if wb_unknown then
        C[30] = Capability UNKNOWN;
    else
        C[30] = link;

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) &&
   CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);

BranchXToCapability(target, branch_type);
```

4.4.11 BLRR

Branch with Link to capability Register with possible switch to Restricted calls a subroutine at an address in the source register, setting C30 to PCC+4. The PE may switch to Restricted based on the Executive permission in PCC.

31	30	29 … 20	19 … 15	14	13	12 … 10	9 … 5	4	3	2	1	0
1	1	0 0 0 0 1 0 1 1 0 0	0 0 1 0 0	0	1	1 0 0	Cn	0	0	0	1	1
				opc<1>	opc<0>							

```
BLRR    <Cn>
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDCALL;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

Operation

```
if IsInRestricted() then
    UndefinedFault();

CheckCapabilitiesEnabled();

Capability target = C[n];

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);
else
    if CCTLR[].SBL == '1' then
        target = CapWithTagClear(target);

integer linkoffset = 4;
Capability link;

if IsInC64() then
    linkoffset = linkoffset + 1;

link = CapAdd(PCC[], linkoffset);

if CCTLR[].SBL == '1' then
    link =  CapSetObjectType(link, CAP_SEAL_TYPE_RB);

C[30] = link;
BranchXToCapability(target, branch_type);
```

4.4.12 BLRS (capability)

Branch with Link to sealed capability calls a subroutine at an address in the source register, sealing and setting C30 to PCC+4.

31	30	29 … 20	19 … 15	14	13	12 … 10	9 … 5	4	3	2	1	0
1	1	0 0 0 0 1 0 1 1 0 0	0 0 1 0 0	0	1	1 0 0	Cn	0	0	0	1	0
				opc<1>	opc<0>							

```
BLRS    <Cn>
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDCALL;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();
Capability target = C[n];

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);
else
    if CCTLR[].SBL == '1' then
        target = CapWithTagClear(target);

integer linkoffset = 4;
Capability link;

if IsInC64() then
    linkoffset = linkoffset + 1;

link = CapAdd(PCC[], linkoffset);

if CCTLR[].SBL == '1' then
    link = CapSetObjectType(link, CAP_SEAL_TYPE_RB);

C[30] = link;
BranchXToCapability(target, branch_type);
```

4.4.13 BLRS (pair of capabilities)

Branch with Link to sealed capability Register with possible switch to Restricted calls a subroutine at an address in the source register, sealing and setting C30 to PCC+4. The PE may switch to Restricted based on the Executive permission in PCC.

31	30	29 21	20 16	15	14	13	12 10	9 5	4	3	2	1	0
1	1	0 0 0 0 1 0 1 1 0	Cm	1	0	1	0 0 1	Cn	0	0	0	0	0
					opc<1>	opc<0>							

```
BLRS    C29, <Cn>, <Cm>
```

```
integer n = UInt(Cn);
integer m = UInt(Cm);
BranchType branch_type = BranchType_INDCALL;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability sealed_target = C[n];
Capability sealed_data = C[m];

if !IsInRestricted() && !CapCheckPermissions(sealed_target, CAP_PERM_EXECUTIVE) then
    sealed_target = CapWithTagClear(sealed_target);

Capability target;
if CapIsTagSet(sealed_target) && CapIsTagSet(sealed_data)
   && CapIsSealed(sealed_target) && CapIsSealed(sealed_data)
   && UInt(CapGetObjectType(sealed_target)) > CAP_MAX_FIXED_SEAL_TYPE
   && CapGetObjectType(sealed_target) == CapGetObjectType(sealed_data)
   && CapCheckPermissions(sealed_target, CAP_PERM_BRANCH_SEALED_PAIR)
   && CapCheckPermissions(sealed_data, CAP_PERM_BRANCH_SEALED_PAIR)
   && CapCheckPermissions(sealed_target, CAP_PERM_EXECUTE)
   && !CapCheckPermissions(sealed_data, CAP_PERM_EXECUTE) then

    target = CapUnseal(sealed_target);
    C[29]  = CapUnseal(sealed_data);
else
    target = CapWithTagClear(sealed_target);
    C[29]  = sealed_data;

integer linkoffset = 4;
Capability link;

if IsInC64() then
    linkoffset = linkoffset + 1;

link = CapAdd(PCC[], linkoffset);

if CCTLR[].SBL == '1' then
    link = CapSetObjectType(link, CAP_SEAL_TYPE_RB);

C[30] = link;
BranchXToCapability(target, branch_type);
```

4.4.14 BR (indirect)

Branch to capability Register branches unconditionally to an address in a Capability register, with a hint that this is not a subroutine return.

31	30	29 ... 20	19 ... 15	14	13	12 ... 10	9 ... 5	4	3	2	1	0
1	1	0 0 0 0 1 0 1 1 0 0	0 0 1 0 0	0	0	1 0 0	Cn	0	0	0	0	0
				opc<1>	opc<0>							

```
BR      <Cn>
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDIR;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();
Capability target = C[n];

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);

BranchXToCapability(target, branch_type);
```

4.4.15 BR (memory indirect)

Unseal load and branch loads a capability and an offset, derives, unseals, and branches to the destination Capability register.

31	30	29									20	19 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	1	imm7	1	0	0	Cn	0	0	0	0	0

```
BR      [<Cn|CSP>, #<imm>]
```

```
integer n = UInt(Cn);
bits(64) offset = SignExtend(imm7:'0000',64);
BranchType branch_type = BranchType_INDIR;
```

Assembler Symbols

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Cn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

Capability     base;
Capability     target;

if n == 31 then
    CheckSPAlignment();
    base = CSP[];
else
    base = C[n];

// When C29 is used, the unsealed capability is written back to C29.
if n == 29 then
    if CapIsTagSet(base) && CapIsSealed(base) &&
       CapGetObjectType(base) == CAP_SEAL_TYPE_LB then
        base = CapUnseal(base);

    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase) + offset;

    VACheckAddress(vabase,addr,CAPABILITY_DBYTES,CAP_PERM_LOAD,AccType_NORMAL);
    target    = MemC[addr,AccType_NORMAL];
    target    = CapSquashPostLoadCap(target,vabase);

    C[29] = base;
else

    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase) + offset;

    VACheckAddress(vabase,addr,CAPABILITY_DBYTES,CAP_PERM_LOAD,AccType_NORMAL);
    target    = MemC[addr,AccType_NORMAL];
    target    = CapSquashPostLoadCap(target,vabase);

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) &&
   CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);

BranchXToCapability(target, branch_type);
```

4.4.16 BRR

Branch to capability Register with possible switch to Restricted branches unconditionally to an address in the source register, with a hint that this is not a subroutine return. The PE may switch to Restricted based on the Executive permission in PCC.

31	30	29									20	19				15	14	13	12		10	9				5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	0	0	1	0	0			Cn			0	0	0	1	1
																	opc<1>	opc<0>													

```
BRR     <Cn>
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDIR;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

Operation

```
if IsInRestricted() then
    UndefinedFault();

CheckCapabilitiesEnabled();

Capability target = C[n];

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);
else
    if CCTLR[].SBL == '1' then
        target = CapWithTagClear(target);

BranchXToCapability(target, branch_type);
```

4.4.17 BRS (capability)

Branch to sealed capability unseals and branches to an address in the source Capability register.

31	30	29									20	19				15	14	13	12		10	9				5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	0	0	1	0	0	Cn					0	0	0	1	0
																	opc<1>	opc<0>													

```
BRS     <Cn>
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDIR;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();
Capability target = C[n];

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);
else
    if CCTLR[].SBL == '1' then
        target = CapWithTagClear(target);

BranchXToCapability(target, branch_type);
```

4.4.18 BRS (pair of capabilities)

Branch to sealed capability pair checks the capabilities have the correct properties to be used as a sealed pair, unseals the source Capability registers, branches to an address in the first Capability register and writes the second Capability register to C29.

31	30	29								21	20 16	15	14	13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	Cm	1	0	0	0	0	1	Cn	0	0	0	0	0
												opc<1>		opc<0>									

```
BRS     C29, <Cn>, <Cm>
```

```
integer n = UInt(Cn);
integer m = UInt(Cm);
BranchType branch_type = BranchType_INDIR;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability sealed_target = C[n];
Capability sealed_data = C[m];

if !IsInRestricted() && !CapCheckPermissions(sealed_target, CAP_PERM_EXECUTIVE) then
    sealed_target = CapWithTagClear(sealed_target);

Capability target;
if CapIsTagSet(sealed_target) && CapIsTagSet(sealed_data)
   && CapIsSealed(sealed_target) && CapIsSealed(sealed_data)
   && UInt(CapGetObjectType(sealed_target)) > CAP_MAX_FIXED_SEAL_TYPE
   && CapGetObjectType(sealed_target) == CapGetObjectType(sealed_data)
   && CapCheckPermissions(sealed_target, CAP_PERM_BRANCH_SEALED_PAIR)
   && CapCheckPermissions(sealed_data, CAP_PERM_BRANCH_SEALED_PAIR)
   && CapCheckPermissions(sealed_target, CAP_PERM_EXECUTE)
   && !CapCheckPermissions(sealed_data, CAP_PERM_EXECUTE) then

    target = CapUnseal(sealed_target);
    C[29]  = CapUnseal(sealed_data);
else
    target = CapWithTagClear(sealed_target);
    C[29]  = sealed_data;

BranchXToCapability(target, branch_type);
```

4.4.19 BUILD

Build capability from untagged and possibly sealed bit pattern interprets and treats an untagged and possibly sealed bit pattern as a capability, checks this capability against a testing capability and based on the result, writes the built capability to the destination Capability register.

31										21	20 16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Cm	0	0	0	0	0	1	Cn	Cd
													opc<1>	opc<0>					

```
BUILD   <Cd|CSP>, <Cn|CSP>, <Cm|CSP>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the first source register or stack pointer, encoded in the "Cn" field.

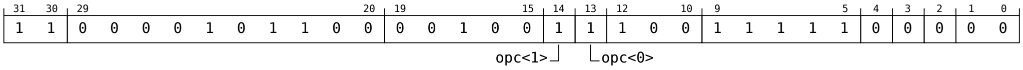
<Cm|CSP> Is the capability name of the second source register or stack pointer, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();
Capability data = if n == 31 then CSP[] else C[n];
Capability key = if m == 31 then CSP[] else C[m];
Capability result;

boolean dataWasSealed = CapIsSealed(data);

if dataWasSealed then
    data = CapUnseal(data);

if !CapIsTagSet(key) || CapIsSealed(key) ||
   !CapIsSubSetOf(data,key) || CapIsBaseAboveLimit(data) then
    if dataWasSealed then
        result = CapWithTagClear(data);
    else
        result = data;
else
    result = CapWithTagSet(data);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.20 BX

Branch Exchange sets PCC to PCC+4 and switches to C64 or A64 depending on the value of PSTATE.C64.



```
BX      #4
```

```
BranchType branch_type = BranchType_DIR;
```

Operation

```
CheckCapabilitiesEnabled();

integer offset = 4;
if !IsInC64() then
    offset = offset + 1;
Capability target = CapAdd(PCC[], offset);

BranchXToCapability(target, branch_type);
```

4.4.21 CAS

Compare and Swap capabilities in memory determines the base register to be used, derives an address from the base register, atomically loads a Capability register from the calculated address in memory, and performs a comparison between this first Capability register with a second Capability register. If the result of the comparison is equal, the second Capability register is atomically stored to the calculated address in memory.

31							24	23	22	21	20 16	15	14				10	9 5	4 0
1	0	1	0	0	0	1	0	1	0	1	Cs	0	1	1	1	1	1	Rn	Ct
									L			R							

```
CAS     <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
CAS     <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
AccType ldacctype = AccType_ATOMICRW;
AccType stacctype = AccType_ATOMICRW;

integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
```

Assembler Symbols

<Cs> Is the capability name of the register to be compared and loaded, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be conditionally stored, encoded in the "Ct" field.

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

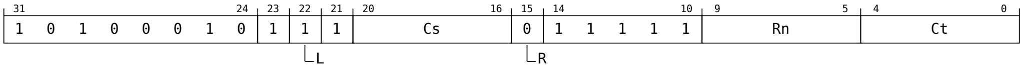
<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability comparecap;
Capability newcap;
Capability data;

comparecap = C[s];
newcap = C[t];
base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(newcap) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(newcap) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

// Both the original VirtualAddress and 64 bit address are passed in
// order to be able to squash permissions and tags correctly.
C[s] = MemAtomicCompareAndSwapC(base,addr,comparecap,newcap,ldacctype,stacctype);
```

4.4.22 CASA

Compare and Swap capabilities in memory with acquire determines the base register to be used, derives an address from the base register, atomically loads a Capability register from the calculated address in memory, and performs a comparison between this first Capability register with a second Capability register. If the result of the comparison is equal, the second Capability register is atomically stored to the calculated address in memory. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release.



```
CASA    <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
CASA    <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
AccType ldacctype = AccType_ORDEREDATOMICRW;
AccType stacctype = AccType_ATOMICRW;

integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
```

Assembler Symbols

<Cs> Is the capability name of the register to be compared and loaded, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be conditionally stored, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability comparecap;
Capability newcap;
Capability data;

comparecap = C[s];
newcap = C[t];
base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(newcap) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(newcap) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

// Both the original VirtualAddress and 64 bit address are passed in
// order to be able to squash permissions and tags correctly.
C[s] = MemAtomicCompareAndSwapC(base,addr,comparecap,newcap,ldacctype,stacctype);
```

4.4.23 CASAL

Compare and Swap capabilities in memory with acquire and release determines the base register to be used, derives an address from the base register, atomically loads a Capability register from the calculated address in memory, and performs a comparison between this first Capability register with a second Capability register. If the result of the comparison is equal, the second Capability register is atomically stored to the calculated address in memory. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. This instruction stores to memory with release semantics.

31 24	23	22	21	20 16	15	14 10	9 5	4 0
1 0 1 0 0 0 1 0	1	1	1	Cs	1	1 1 1 1 1	Rn	Ct
		L			R			

```
CASAL   <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
CASAL   <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
AccType ldacctype = AccType_ORDEREDATOMICRW;
AccType stacctype = AccType_ORDEREDATOMICRW;

integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
```

Assembler Symbols

<Cs> Is the capability name of the register to be compared and loaded, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be conditionally stored, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability comparecap;
Capability newcap;
Capability data;

comparecap = C[s];
newcap = C[t];
base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(newcap) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(newcap) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

// Both the original VirtualAddress and 64 bit address are passed in
// order to be able to squash permissions and tags correctly.
C[s] = MemAtomicCompareAndSwapC(base,addr,comparecap,newcap,ldacctype,stacctype);
```

4.4.24 CASL

Compare and Swap capabilities in memory with release determines the base register to be used, derives an address from the base register, atomically loads a Capability register from the calculated address in memory, and performs a comparison between this first Capability register with a second Capability register. If the result of the comparison is equal, the second Capability register is atomically stored to the calculated address in memory. This instruction stores to memory with release semantics.

31 24	23	22	21	20 16	15	14 10	9 5	4 0
1 0 1 0 0 0 1 0	1	0	1	Cs	1	1 1 1 1 1	Rn	Ct
		L			R			

```
CASL    <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
CASL    <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
AccType ldacctype = AccType_ATOMICRW;
AccType stacctype = AccType_ORDEREDATOMICRW;

integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
```

Assembler Symbols

<Cs> Is the capability name of the register to be compared and loaded, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be conditionally stored, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability comparecap;
Capability newcap;
Capability data;

comparecap = C[s];
newcap = C[t];
base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(newcap) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(newcap) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

// Both the original VirtualAddress and 64 bit address are passed in
// order to be able to squash permissions and tags correctly.
C[s] = MemAtomicCompareAndSwapC(base,addr,comparecap,newcap,ldacctype,stacctype);
```

4.4.25 CFHI

Copy From High copies bits 127 to 64 of the source Capability register to the destination register.

31																15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	1	0	1	0	1	0	0	Cn	Rd
																	opc<1>	opc<0>					

```
CFHI    <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

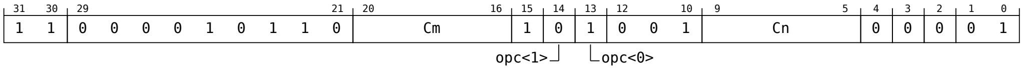
<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64)   result;

result = operand<127:64>;

X[d] = result;
```

4.4.26 CHKEQ

Check for bit equality of two capabilities, setting flags checks if two capabilities are equal. The instruction updates the condition flags based on the result.



```
CHKEQ   <Cn|CSP>, <Cm>
```

```
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cn|CSP> Is the capability name of the first source register or stack pointer, encoded in the "Cn" field.

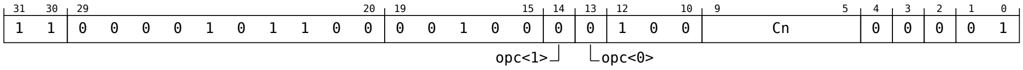
<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
Capability operand2 = C[m];

if operand1 == operand2 then
    PSTATE.<N,Z,C,V> = '0100';
else
    PSTATE.<N,Z,C,V> = '0000';
```

4.4.27 CHKSLD

Check if capability is sealed, setting flags checks if the source Capability register is sealed. The instruction updates the condition flags based on the result.



```
CHKSLD  <Cn|CSP>
```

```
integer n = UInt(Cn);
```

Assembler Symbols

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];

if CapIsSealed(operand) then
    PSTATE.<N,Z,C,V> = '0001';
else
    PSTATE.<N,Z,C,V> = '0000';
```

4.4.28 CHKSS

Check Subset, setting flags checks if a capability is a subset of a testing capability. The instruction updates the condition flags based on the result.

31	30	29 21	20 16	15	14	13	12 10	9 5	4	3	2	1	0
1	1	0 0 0 0 1 0 1 1 0	Cm	1	0	0	0 0 1	Cn	0	0	0	0	1
					opc<1>	opc<0>							

```
CHKSS   <Cn|CSP>, <Cm|CSP>
```

```
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cn|CSP> Is the capability name of the first source register or stack pointer, encoded in the "Cn" field.

<Cm|CSP> Is the capability name of the second source register or stack pointer, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
Capability testingcap = if m == 31 then CSP[] else C[m];

if CapIsSubSetOf(operand1,testingcap) &&
   CapGetTag(operand1) == CapGetTag(testingcap) then
    PSTATE.<N,Z,C,V> = '1000';
else
    PSTATE.<N,Z,C,V> = '0000';
```

4.4.29 CHKSSU

Check Subset, setting flags and conditionally unseal checks if a capability is a subset of a testing capability. If the capability is a valid sealed capability, and the testing capability is a valid unsealed capability, the operation unseals the capability and writes it to the destination Capability register. The instruction updates the condition flags based on the result.

31 21	20 16	15 14	13 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	Cm	1 0	0 0 1 0	Cn	Cd
		opc			

```
CHKSSU  <Cd>, <Cn|CSP>, <Cm|CSP>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the first source register or stack pointer, encoded in the "Cn" field.

<Cm|CSP> Is the capability name of the second source register or stack pointer, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
Capability testingcap = if m == 31 then CSP[] else C[m];
Capability result = operand1;

if CapIsSubSetOf(operand1,testingcap) &&
   CapGetTag(operand1) == CapGetTag(testingcap) then
    if CapIsTagSet(testingcap) && !CapIsSealed(testingcap) &&
       CapIsTagSet(operand1) && CapIsSealed(operand1) then
        result = CapUnseal(operand1);

    PSTATE.<N,Z,C,V> = '1000';
else
    PSTATE.<N,Z,C,V> = '0000';

C[d] = result;
```

4.4.30 CHKTGD

Check if capability has its tag bit set, setting flags checks if the Capability Tag of the source Capability register is set. The instruction updates the condition flags based on the result.

31	30	29 … 20	19 … 15	14	13	12 … 10	9 … 5	4	3	2	1	0
1	1	0 0 0 0 1 0 1 1 0 0	0 0 1 0 0	0	1	1 0 0	Cn	0	0	0	0	1
				opc<1>	opc<0>							

```
CHKTGD  <Cn|CSP>
```

```
integer n = UInt(Cn);
```

Assembler Symbols

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];

if CapIsTagSet(operand) then
    PSTATE.<N,Z,C,V> = '0010';
else
    PSTATE.<N,Z,C,V> = '0000';
```

4.4.31 CLRPERM (immediate)

Clear capability permissions (immediate) clears the Capability Permissions of the source capability based on an immediate value and writes the result to the destination Capability register.

31															16	15 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	1	0	perm	1	0	0	Cn	Cd

```
CLRPERM <Cd|CSP>, <Cn|CSP>, <perm>
```

```
integer n = UInt(Cn);
integer d = UInt(Cd);
bits(3) imm = perm;
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<perm> Is the perm specifier, encoded in"perm":

perm	<perm>
000	#0
001	X
010	W
011	WX
100	R
101	RX
110	RW
111	RWX

Operation

```
CheckCapabilitiesEnabled();

Capability data = if n == 31 then CSP[] else C[n];
Capability result;

bits(64) clr_perms = Zeros(64);
if imm<0> == '1' then
    clr_perms = clr_perms OR CAP_PERM_EXECUTE;
if imm<1> == '1' then
    clr_perms = clr_perms OR CAP_PERM_STORE;
if imm<2> == '1' then
    clr_perms = clr_perms OR CAP_PERM_LOAD;

result = CapClearPerms(data, clr_perms);

if CapIsSealed(data) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.32 CLRPERM (register)

Clear capability Permissions (scalar) clears the Capability Permissions of the source capability using a mask and writes the result to the destination Capability register.

31 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	Rm	1	0	1	0 0 0	Cn	Cd

```
CLRPERM <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

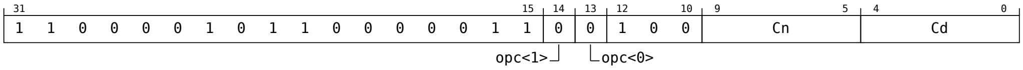
<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability data = if n == 31 then CSP[] else C[n];
bits(64)   mask = X[m];
Capability result;

result = CapClearPerms(data, mask);

if CapIsSealed(data) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.33 CLRTAG

Clear capability Tag clears the Capability Tag of the source capability and writes the result to the destination Capability register



```
CLRTAG  <Cd|CSP>, <Cn|CSP>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
Capability result = CapWithTagClear(operand);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.34 CMP

Compare capabilities if the Capability Tag of the first source Capability register is not the same as the Capability Tag of the second source Capability register subtracts the Capability Tag of the first source Capability register from the Capability Tag of the second source Capability register and discards the result otherwise subtracts the Value field of the first source Capability register from the Value field of the second source Capability register and discards the result. The instruction updates the condition flags based on the result.

This is an alias of SUBS. This means:

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

31 … 21	20 … 16	15 … 10	9 … 5	4 … 0
1 1 0 0 0 0 1 0 1 1 1	Cm	1 0 0 1 1 0	Cn	Rd

```
CMP      <Cn>, <Cm>
```

is equivalent to

```
SUBSXZR, <Cn>, <Cm>
```

and is always the preferred disassembly.

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

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

4.4.35 CPY

Copy Capability register copies a capability from the source Capability register to the destination Capability register.

This instruction is used by the alias MOV.

31 … 15	14	13	12 … 10	9 … 5	4 … 0
1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 1 1	1	0	1 0 0	Cn	Cd
	opc<1>	opc<0>			

```
CPY     <Cd|CSP>, <Cn|CSP>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability result = if n == 31 then CSP[] else C[n];
if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.36 CPYTYPE

Set capability value to the Capability ObjectType of another capability writes the ObjectType from the second capability to the Capability Value of the first capability and writes the result to the destination Capability register. If the first capability is sealed, the destination Capability Tag is cleared.

31–21	20–16	15	14	13	12–10	9–5	4–0
1 1 0 0 0 0 1 0 1 1 0	Cm	0	0	1	0 0 1	Cn	Cd
			opc<1>	opc<0>			

```
CPYTYPE <Cd>, <Cn>, <Cm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability key  = C[n];
Capability data = C[m];
Capability result;

if CapIsSealed(data) then
    result = CapSetValue(key, CapGetObjectType(data));
else
    result = CapSetValue(key, CAP_NO_SEALING);

if CapIsSealed(key) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.37 CPYVALUE

Set capability value to Capability Value of another capability writes the Capability Value from the second capability to the Capability Value of the first capability and writes the result to the destination Capability register. If the first capability is sealed, the destination Capability Tag is cleared.

31 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	Cm	0	1	1	0 0 1	Cn	Cd
			opc<1>	opc<0>			

```
CPYVALUE <Cd>, <Cn>, <Cm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = C[n];
Capability operand2 = C[m];
Capability result;

result = CapSetValue(operand1,CapGetValue(operand2));

if CapIsSealed(operand1) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.38 CSEAL

Conditionally Seal capability seals a capability using a sealing capability if the ObjectType extracted from the Value field of the sealing capability allows this operation. This is intended to be used with BUILD.

31 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	Cm	0	1	0	0 0 1	Cn	Cd
			opc<1>	opc<0>			

```
CSEAL   <Cd|CSP>, <Cn|CSP>, <Cm|CSP>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the first source register or stack pointer, encoded in the "Cn" field.

<Cm|CSP> Is the capability name of the second source register or stack pointer, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
Capability sealingcap = if m == 31 then CSP[] else C[m];

bits(64)   otype = CapGetValue(sealingcap);
Capability result = operand1;

if otype == CAP_NO_SEALING then
    PSTATE.<N,Z,C,V> = '0001';
elsif CapIsTagSet(operand1) && CapIsTagSet(sealingcap) &&
    !CapIsSealed(operand1) && !CapIsSealed(sealingcap) &&
    CapCheckPermissions(sealingcap, CAP_PERM_SEAL) &&
    CapIsInBounds(sealingcap) &&
    UInt(otype) <= CAP_MAX_OBJECT_TYPE then

    result = CapSetObjectType(operand1,otype);
    PSTATE.<N,Z,C,V> = '0001';
else
    PSTATE.<N,Z,C,V> = '0000';

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.39 CSEL

Conditional Select writes, in the destination capability register, the value of the first source capability register if the condition is TRUE, and otherwise writes the value of the second source capability register.

31										21	20 16	15 12	11	10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Cm	cond	1	1	Cn	Cd

```
CSEL    <Cd>, <Cn>, <Cm>, <cond>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

<cond> Is one of the standard conditions, encoded in"cond":

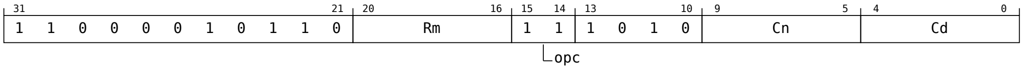
cond	<cond>
0000	EQ
0001	NE
0010	CS
0011	CC
0100	MI
0101	PL
0110	VS
0111	VC
1000	HI
1001	LS
1010	GE
1011	LT
1100	GT
1101	LE
1110	AL
1111	NV

Operation

```
CheckCapabilitiesEnabled();

Capability result;
if ConditionHolds(cond) then
    result = C[n];
else
    result = C[m];
C[d] = result;
```

4.4.40 CTHI

Copy To High copies the source register to bits 127 to 64 of the destination Capability register and clears the Capability Tag of the destination Capability register.



```
CTHI    <Cd|CSP>, <Cn>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability result = C[n];

result<127:64> = X[m];

if d == 31 then
    CSP[] = CapWithTagClear(result);
else
    C[d] = CapWithTagClear(result);
```

4.4.41 CVT (to capability)

Convert pointer to capability offset from a capability derives the Capability Value from the source 64-bit register and Capability register, and writes the result to the destination Capability register.

31										21	20 16	15	14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	Rm	0	0	0	1	1	0	Cn	Cd

```
CVT     <Cd>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64)   operand2 = X[m];
Capability  result;

if CCTLR[].DDCBO == '1' then
    result = CapSetOffset(operand1,operand2);
else
    result = CapSetValue(operand1,operand2);

if CapIsSealed(operand1) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.42 CVT (to pointer)

Convert capability to pointer, setting flags derives an address from the source Capability registers and writes the result to the destination register. The instruction updates the condition flags based on the result.

31										21	20 16	15					10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Cm	1	1	0	0	0	0	Cn	Rd

```
CVT     <Xd>, <Cn|CSP>, <Cm>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the first source register or stack pointer, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
Capability operand2 = C[m];
bits(64)   result;

if CapIsTagSet(operand1) then
    if CCTLR[].DDCBO == '1' then
        result = CapGetValue(operand1) - CapGetBase(operand2);
    else
        result = CapGetValue(operand1);

    if result == 0 then
        PSTATE.<N,Z,C,V> = '0110';
    else
        PSTATE.<N,Z,C,V> = '0010';
else
    result = Zeros(64);
    PSTATE.<N,Z,C,V> = '0000';

X[d] = result;
```

4.4.43 CVTD (to capability)

Convert pointer to capability offset from DDC derives a Capability Value from a 64-bit register and DDC, and writes the result to the destination Capability register.

31																15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	1	1	0	0	1	0	0	Rn	Cd
																	opc<1>	opc<0>					

```
CVTD    <Cd>, <Xn>
```

```
integer d = UInt(Cd);
integer n = UInt(Rn);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

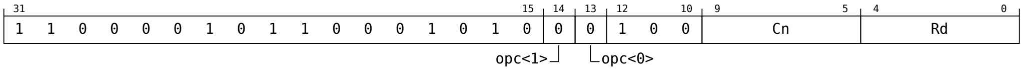
<Xn> Is the 64-bit name of the source general-purpose register, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = DDC[];
bits(64)   operand2 = X[n];
Capability  result;

if CCTLR[].DDCBO == '1' then
    result = CapSetOffset(operand1,operand2);
else
    result = CapSetValue(operand1,operand2);

if CapIsSealed(operand1) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.44 CVTD (to pointer)

Convert capability to pointer offset from DDC, setting flags derives an address from the source Capability register and DDC, and writes the result to the destination register. The instruction updates the condition flags based on the result.



```
CVTD    <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

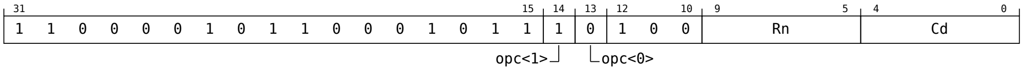
<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
Capability operand2 = DDC[];
bits(64)   result;

if CapIsTagSet(operand1) then
    if CCTLR[].DDCBO == '1' then
        result = CapGetValue(operand1) - CapGetBase(operand2);
    else
        result = CapGetValue(operand1);

    if result == 0 then
        PSTATE.<N,Z,C,V> = '0110';
    else
        PSTATE.<N,Z,C,V> = '0010';
else
    result = Zeros(64);
    PSTATE.<N,Z,C,V> = '0000';

X[d] = result;
```

4.4.45 CVTDZ

Convert pointer to capability offset from DDC, with null capability from zero semantics derives a Capability Value from a 64-bit register and DDC, and writes the result to the destination Capability register. This instruction sets the destination Capability register to zero based on the result.



```
CVTDZ   <Cd>, <Xn>
```

```
integer d = UInt(Cd);
integer n = UInt(Rn);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Xn> Is the 64-bit name of the source general-purpose register, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = DDC[];
bits(64)   operand2 = X[n];
Capability  result;

if operand2 == 0 then
    result = CapNull();
else
    if CCTLR[].DDCBO == '1' then
        result = CapSetOffset(operand1,operand2);
    else
        result = CapSetValue(operand1,operand2);

if CapIsSealed(operand1) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.46 CVTP (to capability)

Convert pointer to capability offset from PCC derives a Capability Value from a 64-bit register and PCC, and writes the result to the destination Capability register.

31 … 15	14	13	12 … 10	9 … 5	4 … 0
1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 1 1	0	1	1 0 0	Rn	Cd
	opc<1>	opc<0>			

```
CVTP    <Cd>, <Xn>
```

```
integer d = UInt(Cd);
integer n = UInt(Rn);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

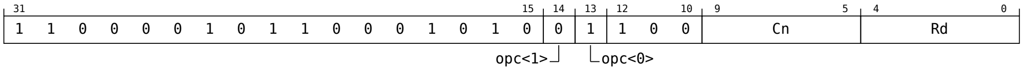
<Xn> Is the 64-bit name of the source general-purpose register, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = PCC[];
bits(64)   operand2 = X[n];
Capability  result;

if CCTLR[].PCCBO == '1' then
    result = CapSetOffset(operand1,operand2);
else
    result = CapSetValue(operand1,operand2);

if CapIsSealed(operand1) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.47 CVTP (to pointer)

Convert capability to pointer offset from PCC, setting flags derives an address from the source Capability register and PCC, and writes the result to the destination register. The instruction updates the condition flags based on the result.



```
CVTP    <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

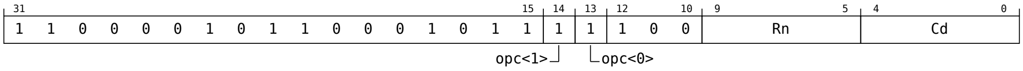
<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
Capability operand2 = PCC[];
bits(64)   result;

if CapIsTagSet(operand1) then
    if CCTLR[].PCCBO == '1' then
        result = CapGetValue(operand1) - CapGetBase(operand2);
    else
        result = CapGetValue(operand1);

    if result == 0 then
        PSTATE.<N,Z,C,V> = '0110';
    else
        PSTATE.<N,Z,C,V> = '0010';
else
    result = Zeros(64);
    PSTATE.<N,Z,C,V> = '0000';

X[d] = result;
```

4.4.48 CVTPZ

Convert pointer to capability offset from PCC, with null capability from zero semantics derives a Capability Value from a 64-bit register and PCC, and writes the result to the destination Capability register. This instruction sets the destination Capability register to zero based on the result.



```
CVTPZ   <Cd>, <Xn>
```

```
integer d = UInt(Cd);
integer n = UInt(Rn);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Xn> Is the 64-bit name of the source general-purpose register, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = PCC[];
bits(64)   operand2 = X[n];
Capability  result;

if operand2 == 0 then
    result = CapNull();
else
    if CCTLR[].PCCBO == '1' then
        result = CapSetOffset(operand1,operand2);
    else
        result = CapSetValue(operand1,operand2);

if CapIsSealed(operand1) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.49 CVTZ

Convert pointer to capability offset from a capability, with null capability from zero semantics derives the Capability Value from the source 64-bit register and Capability register, and writes the result to the destination Capability register. This instruction sets the destination Capability register to zero based on the result.

31										21	20 16	15	14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	Rm	0	1	0	1	1	0	Cn	Cd

```
CVTZ    <Cd>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64)   operand2 = X[m];
Capability  result;

if operand2 == 0 then
    result = CapNull();
else
    if CCTLR[].DDCBO == '1' then
        result = CapSetOffset(operand1,operand2);
    else
        result = CapSetValue(operand1,operand2);

if CapIsSealed(operand1) then
    C[d] = CapWithTagClear(result);
else
    C[d] = result;
```

4.4.50 EORFLGS (immediate)

Bitwise Exclusive OR (immediate) on flags field performs a bitwise XOR of the flags field of a capability and an immediate value and writes the result to the flags field of the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31										21	20 13	12 11	10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	imm8	1 0	0	Cn	Cd

```
EORFLGS <Cd|CSP>, <Cn|CSP>, #<imm>
```

```
integer n = UInt(Cn);
integer d = UInt(Cd);
bits(8) mask = imm8;
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

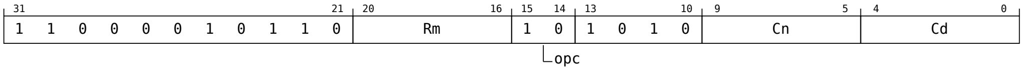
<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];

bits(64) oldvalue = CapGetValue(operand);
bits(8)  newflags = oldvalue<63:56> EOR mask;
bits(64) newvalue = newflags : oldvalue<55:0>;

Capability result = CapSetFlags(operand,newvalue);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.51 EORFLGS (register)

Bitwise Exclusive OR (register) on flags field performs a bitwise XOR of the flags field of a capability and bits 63 to 56 of a register value and writes the result to the flags field of the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.



31 … 21	20 … 16	15	14	13 … 10	9 … 5	4 … 0
1 1 0 0 0 0 1 0 1 1 0	Rm	1	0	1 0 1 0	Cn	Cd
		opc				

```
EORFLGS <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64)   mask    = X[m];

bits(64) oldvalue = CapGetValue(operand);
bits(8)  newflags = oldvalue<63:56> EOR mask<63:56>;
bits(64) newvalue = newflags : oldvalue<55:0>;

Capability result = CapSetFlags(operand,newvalue);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.52 GCBASE

Get the Base field of a capability calculates the base field of a capability and writes it to the destination register.

31															16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0	Cn	Rd
																opc<2:1>		opc<0>					

```
GCBASE  <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

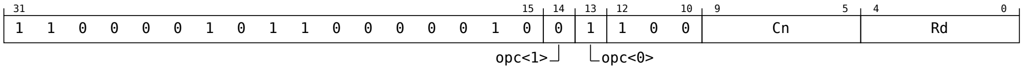
<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(CAP_BOUND_NUM_BITS) result;

(result, - , - ) = CapGetBounds(operand);

X[d] = result<63:0>;
```

4.4.53 GCFLGS

Get the Flags field of a capability gets the Flags field of a capability and writes the result to the destination register.



```
GCFLGS  <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) value  = CapGetValue(operand);
bits(64) result = value<63:56>:Zeros(56);

X[d] = result;
```

4.4.54 GCLEN

Get the Length of a capability calculates the length of a capability from the limit and the base of that capability and writes the result to the destination register.

31															16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	1	1	0	0	Cn	Rd
																opc<2:1>		opc<0>					

```
GCLEN   <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;

bits(65) length = CapGetLength(operand);
if length<64> == '1' then
    result = Ones(64);
else
    result = length<63:0>;

X[d] = result;
```

4.4.55 GCLIM

Get the Limit of a capability calculates the limit of a capability and writes the result to the destination register.

31 15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 1 0	0	0	1 0 0	Cn	Rd
	opc<1>	opc<0>			

```
GCLIM   <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;
bits(CAP_BOUND_NUM_BITS) limit;

( - , limit , - ) = CapGetBounds(operand);
if limit<64> == '1' then
    result = Ones(64);
else
    result = limit<63:0>;

X[d] = result;
```

4.4.56 GCOFF

Get the offset of a capability calculates the Offset of a capability from the Value field and the base of that capability and writes the result to the destination register.

31															16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	1	1	1	0	0	Cn	Rd
																opc<2:1>		opc<0>					

```
GCOFF    <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;

result = CapGetOffset(operand);

X[d] = result;
```

4.4.57 GCPERM

Get the Permissions field of a capability gets the Permissions field of a capability and writes the result to the destination register.

31 16	15 14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 0	1 1	0	1 0 0	Cn	Rd
	opc<2:1>	opc<0>			

```
GCPERM  <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;

result = ZeroExtend(CapGetPermissions(operand),64);

X[d] = result;
```

4.4.58 GCSEAL

Get the sealed status of a capability writes zero to the the destination register if the ObjectType field of the source Capability register is zero and writes one otherwise.

31															16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	1	1	0	0	Cn	Rd
																opc<2:1>		opc<0>					

```
GCSEAL  <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

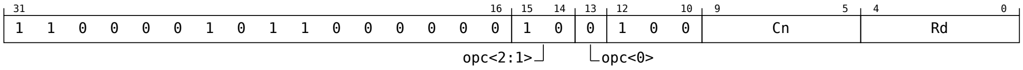
<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;

if CapIsSealed(operand) then
    result = 1<63:0>;
else
    result = 0<63:0>;

X[d] = result;
```

4.4.59 GCTAG

Get the Tag field of a capability gets the Tag field of the source Capability register and writes the result to the destination register.



```
GCTAG   <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

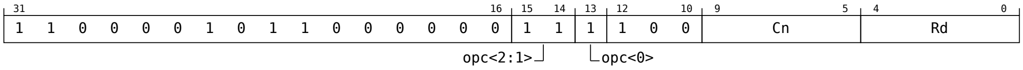
<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;

if CapIsTagSet(operand) then
    result = 1<63:0>;
else
    result = 0<63:0>;

X[d] = result;
```

4.4.60 GCTYPE

Get the ObjectType field of a capability gets the ObjectType field of a capability and writes the result to the destination register.



```
GCTYPE  <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;

result = CapGetObjectType(operand);

X[d] = result;
```

4.4.61 GCVALUE

Get the Value field of a capability gets the range of the Value field of a capability and writes the result to the destination register.

31															16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	1	0	1	0	0	Cn	Rd
																opc<2:1>		opc<0>					

```
GCVALUE <Xd>, <Cn|CSP>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64) result;

result = CapGetValue(operand);

X[d] = result;
```

4.4.62 LDAPR

Load-Acquire RCpc capability determines the base register to be used, derives an address from the base register, loads a capability from memory, and writes it to the destination Capability register. The instruction has memory ordering semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release, except that:

* There is no ordering requirement, separate from the requirements of a Load-AcquirePC or a Store-Release, created by having a Store-Release followed by a Load-AcquirePC instruction. * The reading of a value written by a Store-Release by a Load-AcquirePC instruction by the same observer does not make the write of the Store-Release globally observed.

This difference in memory ordering is not described in the pseudocode. For information about memory accesses, see Load/Store addressing modes.

31							24	23	22	21	20			17	16	15					10	9				5	4				0
1	0	1	0	0	0	1	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	Rn					Ct				

```
LDAPR   <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDAPR   <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;

base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
Capability data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.63 LDAR (capability, alternate base)

Load-Acquire capability via alternate base determines the base register to be used, derives an address from the base register, loads a capability from memory, and writes it to the destination Capability register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20			17	16	15	14			11	10	9 5	4 0
0	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	0	1	1	1	1	1	Rn	Ct
									L														

```
LDAR    <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '0')
```

```
LDAR    <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;

base = AltBaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
Capability data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.64 LDAR (capability, normal base)

Load-Acquire capability determines the base register to be used, derives an address from the base register, loads a capability from memory, and writes it to the destination Capability register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20			17	16	15	14			11	10	9 5	4 0
0	1	0	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	Rn	Ct
									L														

```
LDAR    <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDAR    <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;

base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
Capability data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.65 LDAR (integer)

Load-Acquire Register via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a register from memory, zero-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20			17	16	15	14				10	9 5	4 0
0	1	0	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	Rn	Rt
									L														

```
LDAR    <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '0')
```

```
LDAR    <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
datasize=32;
regsize=32;
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

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

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

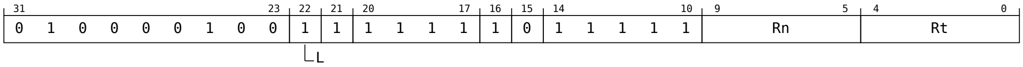
<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress address;

base = AltBaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_LOAD, acctype);
bits(datasize) data = Mem[addr, datasize DIV 8, acctype];

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

4.4.66 LDARB

Load-Acquire Register Byte via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a byte from memory, zero-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.



```
LDARB   <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '0')
```

```
LDARB   <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
datasize=8;
regsize=32;
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress address;

base = AltBaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_LOAD, acctype);
bits(datasize) data = Mem[addr, datasize DIV 8, acctype];

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

4.4.67 LDAXP

Load-Acquire Exclusive Pair of capabilities determines the base register to be used, derives an address from the base register, loads two capabilities from memory, and writes the result to two Capability registers. A 256-bit pair requires the address to be 256-bit aligned. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20				16	15	14 10	9 5	4 0
0	0	1	0	0	0	1	0	0	1	1	1	1	1	1	1	1	Ct2	Rn	Ct
									L										

```
LDAXP   <Ct>, <Ct2>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDAXP   <Ct>, <Ct2>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_ORDEREDATOMIC;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, acctype);

AArch64.SetExclusiveMonitors(addr, CAPABILITY_DBYTES*2);

if addr != Align(addr, CAPABILITY_DBYTES*2) then
    boolean iswrite = FALSE;
    boolean secondstage = FALSE;
    AArch64.Abort(addr, AArch64.AlignmentFault(acctype, iswrite, secondstage));

Capability data1 = MemC[addr, acctype];
Capability data2 = MemC[addr + CAPABILITY_DBYTES, acctype];

if rt_unknown then
    C[t] = Capability UNKNOWN;
    C[t2] = Capability UNKNOWN;
else
    C[t] = CapSquashPostLoadCap(data1, base);
    C[t2] = CapSquashPostLoadCap(data2, base);
```

4.4.68 LDAXR

Load-Acquire Exclusive capability determines the base register to be used, derives an address from the base register, loads two capabilities from memory, and writes the result to two Capability registers. The memory access is atomic. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20				16	15	14				10	9 … 5	4 … 0
0	0	1	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	Rn	Ct
									L														

```
LDAXR   <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDAXR   <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
AccType acctype = AccType_ORDEREDATOMIC;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

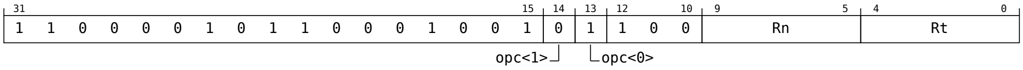
<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;

base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);

AArch64.SetExclusiveMonitors(addr, CAPABILITY_DBYTES);

Capability data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.69 LDCT

Load capability tags loads 4 Capability Tags from memory and writes them to the destination register.



```
LDCT    <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDCT    <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
```

Assembler Symbols

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

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

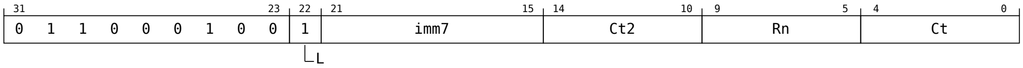
<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = BaseReg[n];
integer count = 4;

bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES*count, CAP_PERM_LOAD, AccType_NORMAL);
bits(64) data = Zeros(64);

if addr != Align(addr, CAPABILITY_DBYTES*count) then
    boolean iswrite = FALSE;
    boolean secondstage = FALSE;
    AArch64.Abort(addr, AArch64.AlignmentFault(AccType_NORMAL, iswrite, secondstage));

for i = 0 to count-1
    bits(1) tag = AArch64.CapabilityTag(addr, AccType_NORMAL);
    data<i> = tag;
    addr = addr + CAPABILITY_DBYTES;

if !VACheckPerm(base, CAP_PERM_LOAD_CAP) then
    data = Zeros(64);

X[t] = data;
```

4.4.70 LDNP

Load Pair of capabilities, with non-temporal hint determines the base register to be used, derives an address from the base register and an immediate offset, loads two capabilities from memory, and writes them to two Capability registers. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair. For information about memory accesses, see Load/Store addressing modes.



```
LDNP    <Ct>, <Ct2>, [<Xn|SP>, #<imm>] //  (PSTATE.C64 == '0')
```

```
LDNP    <Ct>, <Ct2>, [<Cn|CSP>, #<imm>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_STREAM;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, acctype);
Capability data1 = MemC[addr, acctype];
Capability data2 = MemC[addr + CAPABILITY_DBYTES, acctype];

if rt_unknown then
    C[t] = Capability UNKNOWN;
    C[t2] = Capability UNKNOWN;
else
    C[t] = CapSquashPostLoadCap(data1, base);
    C[t2] = CapSquashPostLoadCap(data2, base);
```

4.4.71 LDP (post-indexed)

Load Pair of capabilities (immediate post-index) calculates an address from the source Capability register and an immediate offset, loads two capabilities from memory, and writes them to two Capability registers. For information about memory accesses, see Load/Store addressing modes.

31 23	22	21 15	14 10	9 5	4 0
0 0 1 0 0 0 1 0 1	1	imm7	Ct2	Rn	Ct
	L				

```
LDP     <Ct>, <Ct2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDP     <Ct>, <Ct2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_NORMAL;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

boolean wback = TRUE;
boolean wb_unknown = FALSE;
if (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, acctype);
Capability data1 = MemC[addr, acctype];
Capability data2 = MemC[addr + CAPABILITY_DBYTES, acctype];

if rt_unknown then
    C[t] = Capability UNKNOWN;
    C[t2] = Capability UNKNOWN;
else
    C[t] = CapSquashPostLoadCap(data1, base);
    C[t2] = CapSquashPostLoadCap(data2, base);

```

```
if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);
    BaseReg[n] = base;
```

4.4.72 LDP (pre-indexed)

Load Pair of capabilities (immediate pre-index) calculates an address from the source Capability register and an immediate offset, loads two capabilities from memory, and writes them to two Capability registers. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21–15	14–10	9–5	4–0
0	1	1	0	0	0	1	0	1	1	imm7	Ct2	Rn	Ct
									L				

```
LDP     <Ct>, <Ct2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDP     <Ct>, <Ct2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_NORMAL;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

boolean wback = TRUE;
boolean wb_unknown = FALSE;
if (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;        // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, acctype);
Capability data1 = MemC[addr, acctype];
Capability data2 = MemC[addr + CAPABILITY_DBYTES, acctype];

if rt_unknown then
    C[t] = Capability UNKNOWN;
    C[t2] = Capability UNKNOWN;
else
    C[t] = CapSquashPostLoadCap(data1, base);
    C[t2] = CapSquashPostLoadCap(data2, base);

```

```
if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);
    BaseReg[n] = base;
```

4.4.73 LDP (signed offset)

Load Pair of capabilities (signed offset) calculates an address from the source Capability register and an immediate offset, loads two capabilities from memory, and writes them to two Capability registers. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21 15	14 10	9 5	4 0
0	1	0	0	0	0	1	0	1	1	imm7	Ct2	Rn	Ct
									L				

```
LDP     <Ct>, <Ct2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDP     <Ct>, <Ct2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_NORMAL;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, acctype);
Capability data1 = MemC[addr, acctype];
Capability data2 = MemC[addr + CAPABILITY_DBYTES, acctype];

if rt_unknown then
    C[t] = Capability UNKNOWN;
    C[t2] = Capability UNKNOWN;
else
    C[t] = CapSquashPostLoadCap(data1, base);
    C[t2] = CapSquashPostLoadCap(data2, base);
```

4.4.74 LDPBLR

Load Pair of capabilities and Branch with Link calculates an address from the source Capability register, loads from memory two capabilities, a target capability and a data capability. The instruction writes the data capability to the destination Capability register and branches to the target capability, setting C30 to PCC+4.

31																15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	0	0	0	1	1	0	0	Cn	Ct
																	opc<1>	opc<0>					

```
LDPBLR  <Ct>, [<Cn|CSP>]
```

```
integer t = UInt(Ct);
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDCALL;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability base;
Capability data;
Capability target;
if n == 31 then
    CheckSPAlignment();
    base = CSP[];
else
    base = C[n];

boolean wb_unknown = FALSE;
integer linkoffset = 4;
Capability link;

if IsInC64() then
    linkoffset = linkoffset + 1;

link = CapAdd(PCC[], linkoffset);

if CCTLR[].SBL == '1' then
    link = CapSetObjectType(link, CAP_SEAL_TYPE_RB);

if t == 30 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LINKTRANSFEROVERLAPLD);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    wb_unknown = TRUE;    // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if t == 29 then
    if CapIsTagSet(base) && CapIsSealed(base) &&
       CapGetObjectType(base) == CAP_SEAL_TYPE_LPB then
        base = CapUnseal(base);

    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase);
    VACheckAddress(vabase, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, AccType_NORMAL);

    data   = MemC[addr, AccType_NORMAL];
    target = MemC[addr + CAPABILITY_DBYTES, AccType_NORMAL];
    data = CapSquashPostLoadCap(data, vabase);
    target = CapSquashPostLoadCap(target, vabase);

    C[30] = link;
    C[29] = data;
else
    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase);
    VACheckAddress(vabase, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, AccType_NORMAL);

    data   = MemC[addr, AccType_NORMAL];
    target = MemC[addr + CAPABILITY_DBYTES, AccType_NORMAL];
    data = CapSquashPostLoadCap(data, vabase);
    target = CapSquashPostLoadCap(target, vabase);

    if wb_unknown then
        C[30] = Capability UNKNOWN;
        C[t]  = Capability UNKNOWN;
    else
        C[30] = link;
        C[t] = data;

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) &&
   CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);

BranchXToCapability(target, branch_type);
```

4.4.75 LDPBR

Load Pair of capabilities and Branch calculates an address from the source Capability register, loads from memory two capabilities, a target capability and a data capability. The instruction writes the data capability to the destination Capability register and branches to the target capability.

31																15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	0	0	0	0	1	0	0	Cn	Ct
																	opc<1>	opc<0>					

```
LDPBR   <Ct>, [<Cn|CSP>]
```

```
integer t = UInt(Ct);
integer n = UInt(Cn);
BranchType branch_type = BranchType_INDIR;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();

Capability base;
Capability data;
Capability target;
if n == 31 then
    CheckSPAlignment();
    base = CSP[];
else
    base = C[n];

if t == 29 then
    if CapIsTagSet(base) && CapIsSealed(base) &&
       CapGetObjectType(base) == CAP_SEAL_TYPE_LPB then
        base = CapUnseal(base);

    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase);
    VACheckAddress(vabase, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, AccType_NORMAL);

    data   = MemC[addr, AccType_NORMAL];
    target = MemC[addr + CAPABILITY_DBYTES, AccType_NORMAL];
    data = CapSquashPostLoadCap(data, vabase);
    target = CapSquashPostLoadCap(target, vabase);

    C[29] = data;
else
    VirtualAddress vabase = VAFromCapability(base);
    bits(64) addr = VAddress(vabase);
    VACheckAddress(vabase, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, AccType_NORMAL);

    data   = MemC[addr, AccType_NORMAL];
    target = MemC[addr + CAPABILITY_DBYTES, AccType_NORMAL];
    data = CapSquashPostLoadCap(data, vabase);
    target = CapSquashPostLoadCap(target, vabase);

    C[t] = data;

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) &&
   CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);

BranchXToCapability(target, branch_type);
```

4.4.76 LDR (literal)

Load capability (literal) calculates an address from the PCC value and an immediate offset, loads a capability from memory, and writes it to a Capability register. For information about memory accesses, see Load/Store addressing modes.

31									22	21 … 5	4 … 0
1	0	0	0	0	0	1	0	0	0	imm17	Ct

```
LDR     <Ct>, <label>
```

```
integer t = UInt(Ct);
bits(64) offset = SignExtend(imm17:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<label> Is the program label from which the data is to be loaded. Its offset from the address of this instruction, in the range +/-1MB, encoded in the "imm17" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = VAFromCapability(PCC);
bits(64) address = Align(VAddress(base) + offset, CAPABILITY_DBYTES);
Capability data;

VACheckAddress(base, address, CAPABILITY_DBYTES, CAP_PERM_LOAD, AccType_NORMAL);

data = MemC[address, AccType_NORMAL];
data = CapSquashPostLoadCap(data, base);
C[t] = data;
```

4.4.77 LDR (post-indexed)

Load capability (immediate post-indexed) loads a capability from memory and writes it to a Capability register. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.

31							24	23	22	21	20 … 12	11	10	9 … 5	4 … 0
1	0	1	0	0	0	1	0	0	1	0	imm9	0	1	Rn	Ct
								opc<1>	opc<0>						

```
LDR     <Ct>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
LDR     <Ct>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

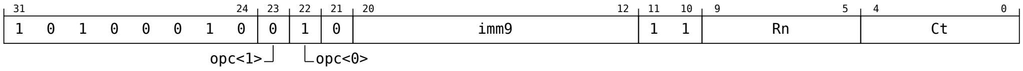
<imm> Is the signed immediate byte offset, a multiple of 16 in the range -4096 to 4080, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

boolean wback = TRUE;
boolean wb_unknown = FALSE;
if n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base);

VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);
C[t] = data;

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);
    BaseReg[n] = base;
```

4.4.78 LDR (pre-indexed)

Load capability (immediate pre-indexed) loads a capability from memory and writes it to a Capability register. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.



```
LDR     <Ct>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
LDR     <Ct>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -4096 to 4080, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

boolean wback = TRUE;
boolean wb_unknown = FALSE;
if n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPLD);
    assert c IN {Constraint_WBSUPPRESS, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_WBSUPPRESS wback = FALSE;       // writeback is suppressed
        when Constraint_UNKNOWN    wb_unknown = TRUE;   // writeback is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);
C[t] = data;

if wback then
    if wb_unknown then
        base = VirtualAddress UNKNOWN;
    else
        base = VAAdd(base,offset);
    BaseReg[n] = base;
```

4.4.79 LDR (register offset, capability, alternate base)

Load capability (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a capability from memory, and writes it to the destination Capability register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. The offset register can optionally be shifted and extended. For information about memory accesses, see Load/Store addressing modes.

31 … 21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
1 1 0 0 0 0 1 0 1 1 1	Rm		1	sz	S	1	1	Rn	Ct
		sign				L			

```
LDR     <Ct>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Ct>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = LOG2_CAPABILITY_DBYTES;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> Is the index shift amount, encoded in"S":

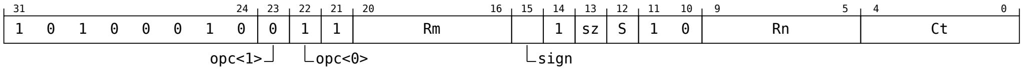
S	<amount>
0	[absent]
1	#4

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
Capability data;

bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, CAPABILITY_DBYTES,  CAP_PERM_LOAD, AccType_NORMAL);
data = MemC[addr, AccType_NORMAL];
data = CapSquashPostLoadCap(data, base);
C[t] = data;
```

4.4.80 LDR (register offset, capability, normal base)

Load capability (register) determines the base register to be used, derives an address from the base register and an offset register, loads a capability from memory, and writes it to the destination Capability register. The offset register can optionally be shifted and extended. For information about memory accesses, see Load/Store addressing modes.



```
LDR    <Ct>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDR    <Ct>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = LOG2_CAPABILITY_DBYTES;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> Is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#4

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = BaseReg[n];
Capability data;

bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, CAPABILITY_DBYTES,  CAP_PERM_LOAD, AccType_NORMAL);
data = MemC[addr, AccType_NORMAL];
data = CapSquashPostLoadCap(data, base);
C[t] = data;
```

4.4.81 LDR (register offset, integer)

Load Register (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a word from memory, and writes the result to the destination register. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	1	1	Rm		1	sz	S	0	1	Rn	Rt
									L			sign				opc			

```
LDR     <Xt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Xt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 3;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 64;
```

Word

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	1	1	Rm		1	sz	S	0	0	Rn	Rt
									L			sign				opc			

```
LDR     <Wt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Wt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 2;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> For the doubleword variant: is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#3

For the word variant: is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#2

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr =  VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8,  CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.82 LDR (register offset, SIMD&FP)

Load SIMD&FP Register (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a SIMD&FP register from memory, and writes the result to the destination SIMD&FP register. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	1	1	Rm		1	sz	S	1	1	Rn	Rt
									L			sign				opc			

```
LDR     <St>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDR     <St>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 2;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

64-bit

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	1	1	Rm		1	sz	S	1	0	Rn	Rt
									L			sign				opc			

```
LDR     <Dt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Dt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 3;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> For the 32-bit variant: is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#2

For the 64-bit variant: is the index shift amount, encoded in"S":

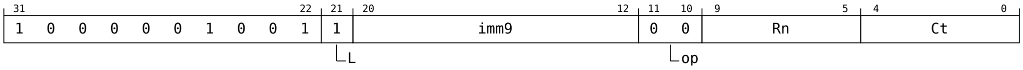
S	<amount>
0	[absent]
1	#3

Operation

```
CheckCapabilitiesEnabled();
CheckFPAdvSIMDEnabled64();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr =  VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8,  CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

V[t] = data;
```

4.4.83 LDR (unsigned offset, capability, alternate base)

Load capability (unsigned offset) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a capability from memory, and writes the result to the destination Capability register. For information about memory accesses, see Load/Store addressing modes. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register.



```
LDR     <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

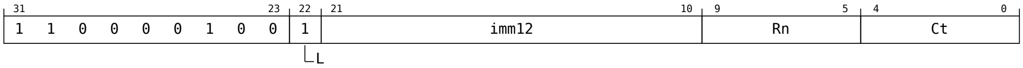
<imm> Is the optional unsigned immediate byte offset, a multiple of 16 in the range 0 to 8176, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, CAPABILITY_DBYTES,  CAP_PERM_LOAD, AccType_NORMAL);
Capability data = MemC[addr, AccType_NORMAL];
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.84 LDR (unsigned offset, capability, normal base)

Load capability (unsigned offset) determines the base register to be used, derives an address from the base register and an immediate offset, loads a capability from memory, and writes the result to the destination Capability register. For information about memory accesses, see Load/Store addressing modes.



```
LDR     <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm12:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0, encoded in the "imm12" field.

Operation

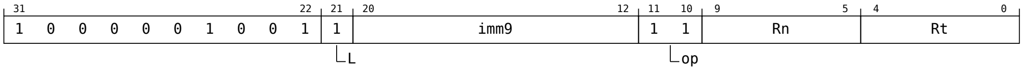
```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);
C[t] = data;
```

4.4.85 LDR (unsigned offset, integer)

Load Register (unsigned offset) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

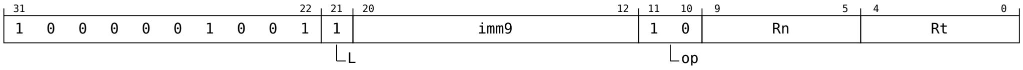


```
LDR     <Xt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Xt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9:'000', 64);
datasize = 64;
regsize  = 64;
```

Word



```
LDR     <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDR     <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9:'00', 64);
datasize = 32;
regsize  = 32;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> For the doubleword variant: is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 4088, defaulting to 0, encoded in the "imm9" field.

For the word variant: is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 2044, defaulting to 0, encoded in the "imm9" field.

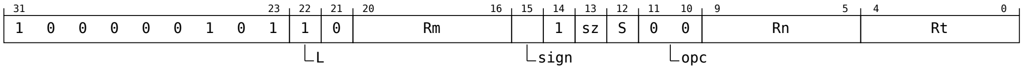
Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

```

```
VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.86 LDRB (register offset)

Load Register Byte (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a byte from memory, zero-extends it, and writes the result to the destination register. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
LDRB    <Wt>, [<Cn|CSP>, <R><m>, <extend>] //  (PSTATE.C64 == '0')
```

```
LDRB    <Wt>, [<Xn|SP>, <R><m>, <extend>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 0;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

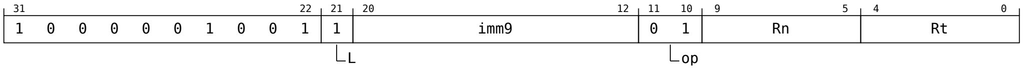
sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr =  VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8,  CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.87 LDRB (unsigned offset)

Load Register Byte (unsigned offset) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a byte from memory, zero-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
LDRB    <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDRB    <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9, 64);
datasize = 8;
regsize  = 32;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, in the range 0 to 511, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.88 LDRH

Load Register Halfword (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a halfword from memory, zero-extends it, and writes the result to the destination register. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	0	0	0	0	1	0	1	1	0	Rm			1	sz	S	1	1	Rn		Rt	
									L				sign				opc					

```
LDRH    <Wt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDRH    <Wt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 1;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> Is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#1

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr =  VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8,  CAP_PERM_LOAD, AccType_NORMAL);
```

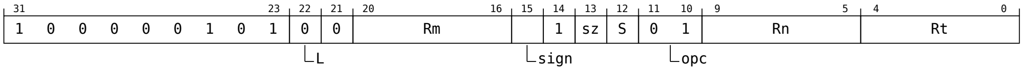
```
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.89 LDRSB

Load Register Signed Byte (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a byte from memory, sign-extends it, and writes the result to the destination register. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword



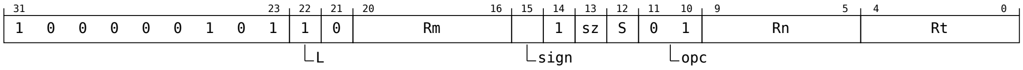
31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	0	0	Rm		1	sz	S	0	1	Rn	Rt
									L			sign				opc			

```
LDRSB   <Xt>, [<Cn|CSP>, <R><m>, <extend>] //  (PSTATE.C64 == '0')
```

```
LDRSB   <Xt>, [<Xn|SP>, <R><m>, <extend>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 0;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 64;
```

Word



31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	1	0	Rm		1	sz	S	0	1	Rn	Rt
									L			sign				opc			

```
LDRSB   <Wt>, [<Cn|CSP>, <R><m>, <extend>] //  (PSTATE.C64 == '0')
```

```
LDRSB   <Wt>, [<Xn|SP>, <R><m>, <extend>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 0;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr =  VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8,  CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.90 LDRSH

Load Register Signed Halfword (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, loads a halfword from memory, sign-extends it, and writes the result to the destination register. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	0	0	Rm		1	sz	S	1	0	Rn	Rt
									L				sign			opc			

```
LDRSH   <Xt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDRSH   <Xt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 1;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 64;
```

Word

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	1	0	Rm		1	sz	S	1	0	Rn	Rt
									L				sign			opc			

```
LDRSH   <Wt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
LDRSH   <Wt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 1;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> Is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#1

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr =  VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8,  CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

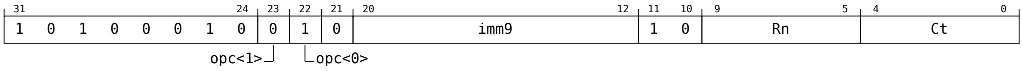
4.4.91 LDTR

Load capability (unprivileged) determines the base register to be used, derives an address from the base register and an immediate offset, loads a capability from memory, and writes the result to the destination Capability register. For information about memory accesses, see Load/Store addressing modes. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

* The instruction is executed at EL1. * The instruction is executed at EL2 when the Effective value of both HCR_EL2.E2H and HCR_EL2.TGE are 1.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed.

In all cases the memory access operates with the capability restrictions as determined by the Exception level at which the instruction is executed.



```
LDTR    <Ct>, [<Xn|SP>, #<imm>] //  (PSTATE.C64 == '0')
```

```
LDTR    <Ct>, [<Cn|CSP>, #<imm>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

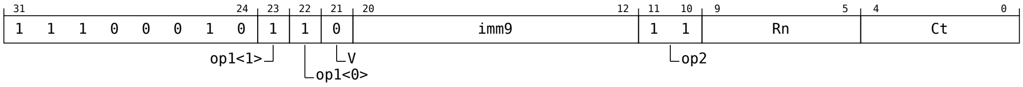
<imm> Is the signed immediate byte offset, a multiple of 16 in the range -4096 to 4080, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);
C[t] = data;
```

4.4.92 LDUR (capability, alternate base)

Load capability (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a capability from memory, and writes the result to the destination Capability register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
LDUR    <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, CAPABILITY_DBYTES,  CAP_PERM_LOAD, AccType_NORMAL);
Capability data = MemC[addr, AccType_NORMAL];
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.93 LDUR (capability, normal base)

Load capability (unscaled) determines the base register to be used, derives an address from the base register and an immediate offset, loads a capability from memory, and writes the result to the destination Capability register. For information about memory accesses, see Load/Store addressing modes.

31							24	23	22	21	20 12	11	10	9 5	4 0
1	0	1	0	0	0	1	0	0	1	0	imm9	0	0	Rn	Ct
								opc<1>		opc<0>					

```
LDUR    <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

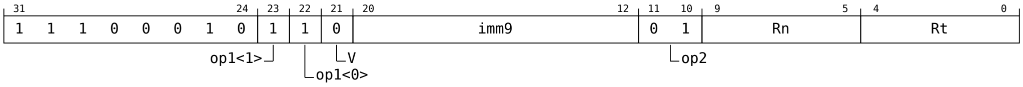
```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);
data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);
C[t] = data;
```

4.4.94 LDUR (integer)

Load Register (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a 32-bit word or 64-bit doubleword from memory, zero-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

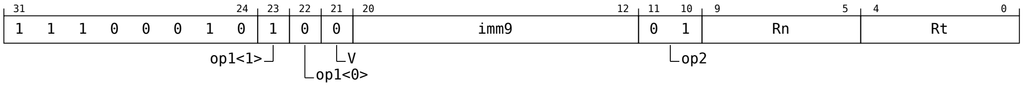


```
LDUR    <Xt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Xt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 64;
regsize  = 64;
```

Word



```
LDUR    <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 32;
regsize  = 32;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.95 LDUR (SIMD&FP)

Load SIMD&FP Register (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a SIMD&FP register from memory, and writes the result to the destination SIMD&FP register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

8-bit

31 24	23	22	21	20 12	11 10	9 5	4 0
1 1 1 0 0 0 1 0	0	0	1	imm9	0 1	Rn	Rt
	op1<1>	op1<0>	V		op2		

```
LDUR    <Bt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Bt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 8;
```

16-bit

31 24	23	22	21	20 12	11 10	9 5	4 0
1 1 1 0 0 0 1 0	0	1	1	imm9	0 1	Rn	Rt
	op1<1>	op1<0>	V		op2		

```
LDUR    <Ht>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Ht>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 16;
```

32-bit

31 24	23	22	21	20 12	11 10	9 5	4 0
1 1 1 0 0 0 1 0	1	0	1	imm9	0 1	Rn	Rt
	op1<1>	op1<0>	V		op2		

```
LDUR    <St>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <St>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 32;
```

64-bit

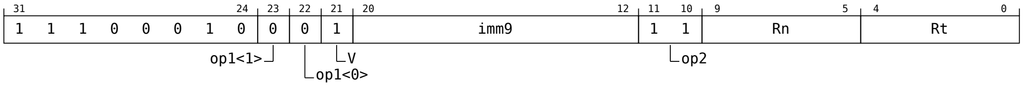
31 24	23	22	21	20 12	11 10	9 5	4 0
1 1 1 0 0 0 1 0	1	1	1	imm9	0 1	Rn	Rt
	op1<1>	op1<0>	V		op2		

```
LDUR    <Dt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Dt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 64;
```

128-bit



```
LDUR    <Qt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDUR    <Qt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 128;
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

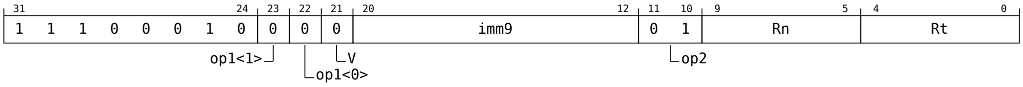
<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();
CheckFPAdvSIMDEnabled64();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

V[t] = data;
```

4.4.96 LDURB

Load Register Byte (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a byte from memory, zero-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
LDURB   <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDURB   <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 8;
regsize  = 32;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

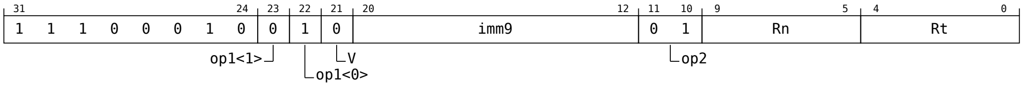
<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.97 LDURH

Load Register Halfword (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a halfword from memory, zero-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
LDURH   <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDURH   <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 16;
regsize  = 32;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

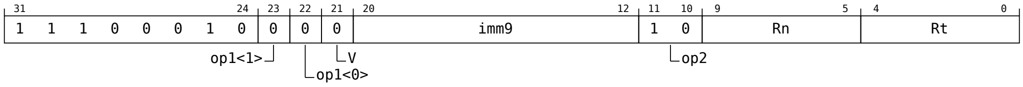
```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.98 LDURSB

Load Register Signed Byte (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset,loads a byte from memory, sign-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

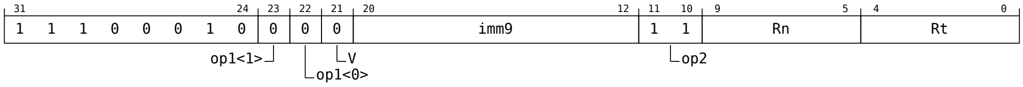


```
LDURSB  <Xt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDURSB  <Xt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 8;
regsize  = 64;
```

Word



```
LDURSB  <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDURSB  <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 8;
regsize  = 32;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

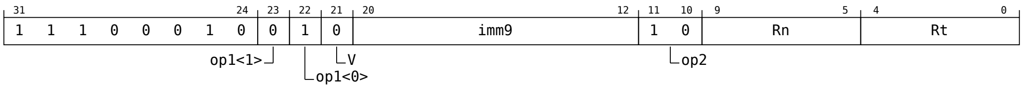
```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.99 LDURSH

Load Register Signed Halfword (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a halfword from memory, sign-extends it, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

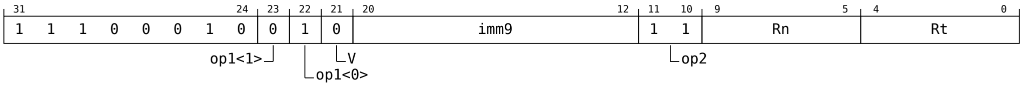


```
LDURSH  <Xt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDURSH  <Xt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 16;
regsize  = 64;
```

Word



```
LDURSH  <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDURSH  <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 16;
regsize  = 32;
```

Assembler Symbols

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

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

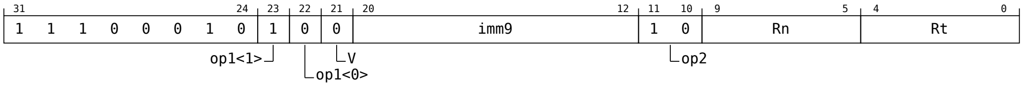
<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.100 LDURSW

Load Register Signed Word (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, loads a word from memory, sign-extends it to form a 64-bit value, and writes the result to the destination register. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
LDURSW  <Xt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
LDURSW  <Xt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 32;
regsize  = 64;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8 , CAP_PERM_LOAD, AccType_NORMAL);
bits(datasize) data = Mem[addr, datasize DIV 8, AccType_NORMAL];

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

4.4.101 LDXP

Load Exclusive Pair of capabilities determines the base register to be used, derives an address from the base register, loads two capabilities from memory, and writes the result to two Capability registers. A 256-bit pair requires the address to be 256-bit aligned. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses, see Load/Store addressing modes.

31 23	22	21	20 16	15	14 10	9 5	4 0
0 0 1 0 0 0 1 0 0	1	1	1 1 1 1 1	0	Ct2	Rn	Ct
	L						

```
LDXP    <Ct>, <Ct2>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDXP    <Ct>, <Ct2>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_ATOMIC;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
boolean rt_unknown = FALSE;

if t == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_LDPOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // result is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES*2, CAP_PERM_LOAD, acctype);

AArch64.SetExclusiveMonitors(addr, CAPABILITY_DBYTES*2);

if addr != Align(addr, CAPABILITY_DBYTES*2) then
    boolean iswrite = FALSE;
    boolean secondstage = FALSE;
    AArch64.Abort(addr, AArch64.AlignmentFault(acctype, iswrite, secondstage));

Capability data1 = MemC[addr, acctype];
Capability data2 = MemC[addr + CAPABILITY_DBYTES, acctype];

if rt_unknown then
    C[t] = Capability UNKNOWN;
    C[t2] = Capability UNKNOWN;
else
    C[t] = CapSquashPostLoadCap(data1, base);
    C[t2] = CapSquashPostLoadCap(data2, base);
```

4.4.102 LDXR

Load Exclusive capability determines the base register to be used, derives an address from the base register, loads a capability from memory, and writes the result to the destination Capability register. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses, see Load/Store addressing modes.

31 23	22	21	20 16	15	14 10	9 5	4 0
0 0 1 0 0 0 1 0 0	1	0	1 1 1 1 1	0	1 1 1 1 1	Rn	Ct
	L						

```
LDXR    <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
LDXR    <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
AccType acctype = AccType_ATOMIC;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;

base = BaseReg[n];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, acctype);

AArch64.SetExclusiveMonitors(addr, CAPABILITY_DBYTES);

Capability data = MemC[addr, acctype];
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.103 MOV

Move between registers

This is an alias of CPY. This means:

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

31																15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1	1	0	1	0	0	Cn	Cd
																	opc<1>	opc<0>					

```
MOV     <Cd|CSP>, <Cn|CSP>
```

is equivalent to

```
CPY<Cd|CSP>, <Cn|CSP>
```

and is always the preferred disassembly.

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

Operation

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

4.4.104 MRS

Move System Register to Capability register allows the PE to read a capability from an AArch64 System register into the destination Capability register

31										21	20	19	18 16	15 12	11 8	7 5	4 0
1	1	0	0	0	0	1	0	1	0	0	1	o0	op1	CRn	CRm	op2	Ct
											L						

```
MRS     <Ct>, (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>)
```

```
integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
integer sys_op2 = UInt(op2);
integer t = UInt(Ct);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<op0> Is the op0 specifier, encoded in"o0":

o0	<op0>
0	2
1	3

<op1> Is the unsigned immediate operand, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is the name Cn, with n in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is the name Cm, with m in the range 0 to 15, encoded in the "CRm" field.

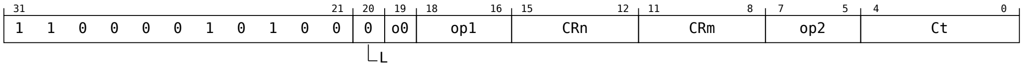
<op2> Is the unsigned immediate operand, in the range 0 to 7, encoded in the "op2" field.

Operation

```
CheckCapabilitiesEnabled();

C[t] = AArch64.CapSysRegRead(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2);
```

4.4.105 MSR

Move Capability register to System Register allows the PE to write a capability to an AArch64 System register from a capability general-purpose register.



```
MSR    (<systemreg>|S<op0>_<op1>_<Cn>_<Cm>_<op2>), <Ct>
```

```
integer sys_op0 = 2 + UInt(o0);
integer sys_op1 = UInt(op1);
integer sys_crn = UInt(CRn);
integer sys_crm = UInt(CRm);
integer sys_op2 = UInt(op2);
integer t = UInt(Ct);
```

Assembler Symbols

<op0> Is the op0 specifier, encoded in"o0":

o0	<op0>
0	2
1	3

<op1> Is the unsigned immediate operand, in the range 0 to 7, encoded in the "op1" field.

<Cn> Is the name Cn, with n in the range 0 to 15, encoded in the "CRn" field.

<Cm> Is the name Cm, with m in the range 0 to 15, encoded in the "CRm" field.

<op2> Is the unsigned immediate operand, in the range 0 to 7, encoded in the "op2" field.

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

Operation

```
CheckCapabilitiesEnabled();

AArch64.CapSysRegWrite(sys_op0, sys_op1, sys_crn, sys_crm, sys_op2, C[t]);
```

4.4.106 ORRFLGS (immediate)

Bitwise OR (immediate) on flags field performs a bitwise OR of the flags field of a capability and an immediate value and writes the result to the flags field of the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31										21	20 13	12	11	10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	imm8	0	1	0	Cn	Cd

```
ORRFLGS <Cd|CSP>, <Cn|CSP>, #<imm>
```

```
integer n = UInt(Cn);
integer d = UInt(Cd);
bits(8) mask = imm8;
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

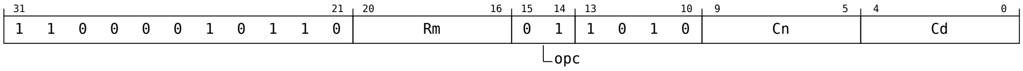
<imm> Is the unsigned immediate operand, in the range 0 to 255, encoded in the "imm8" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];

bits(64) oldvalue = CapGetValue(operand);
bits(8)  newflags = oldvalue<63:56> OR mask;
bits(64) newvalue = newflags : oldvalue<55:0>;

Capability result = CapSetFlags(operand,newvalue);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.107 ORRFLGS (register)

Bitwise OR on flags field performs a bitwise OR of the flags field of a capability and bits 63 to 56 of a register value and writes the result to the flags field of the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.



```
ORRFLGS <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

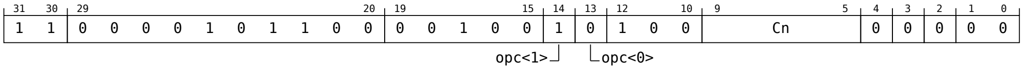
<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand = if n == 31 then CSP[] else C[n];
bits(64)   mask    = X[m];

bits(64) oldvalue = CapGetValue(operand);
bits(8)  newflags = oldvalue<63:56> OR mask<63:56>;
bits(64) newvalue = newflags : oldvalue<55:0>;

Capability result = CapSetFlags(operand,newvalue);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.108 RET

Return from subroutine branches unconditionally to an address in the source Capability register, with a hint that this is a subroutine return.



```
RET    { <Cn>}
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_RET;
```

Assembler Symbols

<Cn> Is the optional capability name of the first source register, defaulting to C30 in C64, encoded in the "Cn" field. To avoid confusion with RET {<Xn>} disassemblers should not omit <Cn>.

Operation

```
CheckCapabilitiesEnabled();
Capability target = C[n];

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);

BranchXToCapability(target, branch_type);
```

4.4.109 RETR

Return from subroutine with possible switch to Restricted branches unconditionally to an address in the source Capability register, with a hint that this is a subroutine return. The PE may switch to Restricted based on the Executive permission in PCC.

31 30	29 … 20	19 … 15	14	13	12 … 10	9 … 5	4	3	2	1	0
1 1	0 0 0 0 1 0 1 1 0 0	0 0 1 0 0	1	0	1 0 0	Cn	0	0	0	1	1
			opc<1>	opc<0>							

```
RETR   { <Cn>}
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_RET;
```

Assembler Symbols

<Cn> Is the optional capability name of the first source register, defaulting to C30 encoded in the "Cn" field.

Operation

```
if IsInRestricted() then
    UndefinedFault();

CheckCapabilitiesEnabled();

Capability target = C[n];

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);
else
    if CCTLR[].SBL == '1' then
        target = CapWithTagClear(target);

BranchXToCapability(target, branch_type);
```

4.4.110 RETS (capability)

Return to sealed capability unseals and branches to an address in the source Capability register with a hint that this is a return.

31	30	29									20	19				15	14	13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	1	0	1	0	0	Cn	0	0	0	1	0
																	opc<1>	opc<0>									

```
RETS   { <Cn>}
```

```
integer n = UInt(Cn);
BranchType branch_type = BranchType_RET;
```

Assembler Symbols

<Cn> Is the optional capability name of the first source register, defaulting to C30 encoded in the "Cn" field.

Operation

```
CheckCapabilitiesEnabled();
Capability target = C[n];

if !IsInRestricted() && !CapCheckPermissions(target, CAP_PERM_EXECUTIVE) then
    target = CapWithTagClear(target);

if CapIsTagSet(target) && CapIsSealed(target) && CapGetObjectType(target) == CAP_SEAL_TYPE_RB then
    target = CapUnseal(target);
else
    if CCTLR[].SBL == '1' then
        target = CapWithTagClear(target);

BranchXToCapability(target, branch_type);
```

4.4.111 RETS (pair of capabilities)

Return to sealed capability pair checks the capabilities have the correct properties to be used as a sealed pair, unseals the source Capability registers, branches to an address in the first Capability register and writes the second Capability register to C29, with a hint that this is a return.

31	30	29 21	20 16	15	14	13	12 10	9 5	4	3	2	1	0
1	1	0 0 0 0 1 0 1 1 0	Cm	1	1	0	0 0 1	Cn	0	0	0	0	0
					opc<1>	opc<0>							

```
RETS    C29, <Cn>, <Cm>
```

```
integer n = UInt(Cn);
integer m = UInt(Cm);
BranchType branch_type = BranchType_RET;
```

Assembler Symbols

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability sealed_target = C[n];
Capability sealed_data = C[m];

if !IsInRestricted() && !CapCheckPermissions(sealed_target, CAP_PERM_EXECUTIVE) then
    sealed_target = CapWithTagClear(sealed_target);

Capability target;
if CapIsTagSet(sealed_target) && CapIsTagSet(sealed_data)
   && CapIsSealed(sealed_target) && CapIsSealed(sealed_data)
   && UInt(CapGetObjectType(sealed_target)) > CAP_MAX_FIXED_SEAL_TYPE
   && CapGetObjectType(sealed_target) == CapGetObjectType(sealed_data)
   && CapCheckPermissions(sealed_target, CAP_PERM_BRANCH_SEALED_PAIR)
   && CapCheckPermissions(sealed_data, CAP_PERM_BRANCH_SEALED_PAIR)
   && CapCheckPermissions(sealed_target, CAP_PERM_EXECUTE)
   && !CapCheckPermissions(sealed_data, CAP_PERM_EXECUTE) then

    target = CapUnseal(sealed_target);
    C[29]  = CapUnseal(sealed_data);
else
    target = CapWithTagClear(sealed_target);
    C[29]  = sealed_data;

BranchXToCapability(target, branch_type);
```

4.4.112 RRLEN

Round Representable Length generates a length, writing it to the destination register. Together with a Capability Value masked as per RRMASK, the length can be used with SCBNDSE to set representable bounds.

31 … 15	14	13	12 … 10	9 … 5	4 … 0
1 1 0 0 0 0 1 0 1 1 0 0 0 1 1 1 0	0	0	1 0 0	Rn	Rd
	opc<1>	opc<0>			

```
RRLEN   <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the source general-purpose register, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

bits(64) request = X[n];

bits(64) mask = CapGetRepresentableMask(request);

X[d] = (request + NOT(mask)) AND mask;
```

4.4.113 RRMASK

Round Representable Mask generates a mask, writing it to the destination register. Together with a length obtained from RRLEN, the mask can be used on a Capablity Value to set representable bounds with SCBNDSE.

31 15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0 0 0 1 1 1 0	0	1	1 0 0	Rn	Rd
	opc<1>	opc<0>			

```
RRMASK  <Xd>, <Xn>
```

```
integer d = UInt(Rd);
integer n = UInt(Rn);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Xn> Is the 64-bit name of the source general-purpose register, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

bits(64) request = X[n];

bits(64) mask = CapGetRepresentableMask(request);

X[d] = mask;
```

4.4.114 SCBNDS (immediate)

Set Bounds (immediate) derives Capability Bounds using the source Capability register and a length from an immediate offset and writes the result to the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared

It has encodings from 2 classes: Scaled and Unscaled

Scaled

31										21	20 15	14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	imm6	1	1	1	1	0	Cn	Cd
												S						

```
SCBNDS  <Cd|CSP>, <Cn|CSP>, #<imm>, LSL #4
```

```
integer n = UInt(Cn);
integer d = UInt(Cd);
bits(65) length = if S == '1' then ZeroExtend(imm6:'0000',65) else ZeroExtend(imm6,65);
```

Unscaled

31										21	20 15	14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	imm6	0	1	1	1	0	Cn	Cd
												S						

```
SCBNDS  <Cd|CSP>, <Cn|CSP>, #<imm>
```

```
integer n = UInt(Cn);
integer d = UInt(Cd);
bits(65) length = if S == '1' then ZeroExtend(imm6:'0000',65) else ZeroExtend(imm6,65);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<imm> Is the unsigned immediate operand, in the range 0 to 63, encoded in the "imm6" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand =  if n == 31 then CSP[] else C[n];

Capability result = CapSetBounds(operand, length, TRUE);

if CapIsSealed(operand) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.115 SCBNDS (register)

Set Bounds derives Capability Bounds using the source Capability register and a length from a 64-bit register and writes the result to the destination Capability register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared

31 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	Rm	0	0	0	0 0 0	Cn	Cd
			opc<1>	opc<0>			

```
SCBNDS  <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64) xm = X[m];
bits(65) length = ZeroExtend(xm,65);

Capability result = CapSetBounds(operand1, length, FALSE);

if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.116 SCBNDSE

Set Bounds Exact derives Capability Bounds using the source Capability register and a length from a 64-bit register and writes the result to the destination Capability register. If the bounds cannot be set exactly, this instruction clears the Capability Tag. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared

31										21	20 16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Rm	0	0	1	0	0	0	Cn	Cd
													opc<1>	opc<0>					

```
SCBNDSE <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64) xm = X[m];
bits(65) length = ZeroExtend(xm,65);

Capability result = CapSetBounds(operand1, length, TRUE);

if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.117 SCFLGS

Set the Flags field of a capability writes the source Capability register to the destination Capability register with the Flags field set to a value based on a 64-bit general-purpose register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31 … 21	20 … 16	15 … 10	9 … 5	4 … 0
1 1 0 0 0 0 1 0 1 1 0	Rm	1 1 1 0 0 0	Cn	Cd

```
SCFLGS  <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

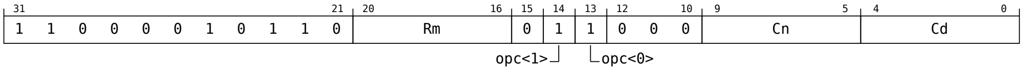
<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64)   newflags = X[m];

bits(64) oldvalue = CapGetValue(operand1);
bits(64) newvalue = newflags<63:56>: oldvalue<55:0>;

Capability result = CapSetFlags(operand1,newvalue);

if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.118 SCOFF

Set the offset field of a capability writes the source Capability register to the destination Capability register with the offset set to a value based on a 64-bit general-purpose register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.



```
SCOFF   <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64)   newoffset = X[m];
Capability result;

result = CapSetOffset(operand1,newoffset);
if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.119 SCTAG

Set the Capability Tag field writes the source Capability register to the destination Capability register with the Tag field set to a value based on a 64-bit general-purpose register.

31										21	20 16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Rm	1	0	0	0	0	0	Cn	Cd

```
SCTAG   <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64)   newtag = X[m];
Capability result;

if newtag<0> == '1' && CapIsSystemAccessEnabled() && !IsTagSettingDisabled() then
    result = CapWithTagSet(operand1);
else
    result = CapWithTagClear(operand1);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.120 SCVALUE

Set value field of a capability writes the source Capability register to the destination Capability register with the Value field set to a value based on a 64-bit general-purpose register. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31 21	20 16	15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	Rm	0	1	0	0 0 0	Cn	Cd
			opc<1>	opc<0>			

```
SCVALUE <Cd|CSP>, <Cn|CSP>, <Xm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Rm);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<Xm> Is the 64-bit name of the source general-purpose register, encoded in the "Rm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
bits(64)   newvalue = X[m];
Capability result;

result = CapSetValue(operand1,newvalue);
if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.121 SEAL (capability)

Seal capability seals a capability with a sealing capability, by setting the ObjectType of the capability to the Capability Value of the sealing capability, and writes the result to the destination Capability register.

31 21	20 16	15	14	13 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0	Cm	0	0	0 0 1 0	Cn	Cd
		opc				

```
SEAL    <Cd>, <Cn>, <Cm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();
bits(64) otype = CapGetValue(C[m]);
Capability  c  = CapSetObjectType(C[n], otype);

if CapIsTagSet(C[n]) && CapIsTagSet(C[m]) &&
    !CapIsSealed(C[n]) && !CapIsSealed(C[m]) &&
    CapCheckPermissions(C[m], CAP_PERM_SEAL) &&
    CapIsInBounds(C[m]) &&
    UInt(otype) <= CAP_MAX_OBJECT_TYPE then

    C[d] = c;
else
    C[d] = CapWithTagClear(c);
```

4.4.122 SEAL (immediate)

Seal capability (immediate) seals a capability by setting the ObjectType of that capability to nonzero, and writes the result to the destination Capability register. An operand of rb seals for use with a register based branch, lpb for a load pair and branch and lb for a load and branch.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	1	0	form	1	0	0	Cn	Cd

```
SEAL    <Cd>, <Cn>, <form>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
bits(64) f = ZeroExtend(form,64);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

<form> Is the form specifier, encoded in"form":

form	<form>
00	RESERVED
01	rb
10	lpb
11	lb

Operation

```
if f == 0 then
    UNDEFINED;

CheckCapabilitiesEnabled();

Capability c = CapSetObjectType(C[n], f);

if CapIsTagSet(C[n]) && !CapIsSealed(C[n]) then
    C[d] = c;
else
    C[d] = CapWithTagClear(c);
```

4.4.123 STCT

Store capability tags stores four Capability Tags to memory. The address that is used for the store is calculated from a base register.

31 15	14	13	12 10	9 5	4 0
1 1 0 0 0 0 1 0 1 1 0 0 0 1 0 0 1	0	0	1 0 0	Rn	Rt
	opc<1>	opc<0>			

```
STCT    <Xt>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STCT    <Xt>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
if PSTATE.EL == EL0 then
    UNDEFINED;

CheckCapabilitiesEnabled();

VirtualAddress base = BaseReg[n];
integer count = 4;
boolean willabort = FALSE;
boolean iswrite;
boolean secondstage;
bits(64) data = X[t];

bits(64) addr = VAddress(base);

if addr != Align(addr, CAPABILITY_DBYTES*count) then
    iswrite = TRUE;
    secondstage = FALSE;
    willabort = TRUE;

for i = 0 to count-1
    bits(1) tag;
    if CapIsSystemAccessEnabled() && !IsTagSettingDisabled() then
        tag = data<i>;
    else
        tag = '0';

    bits(64) cap_required = CAP_PERM_STORE;
    if tag == '1' then
        cap_required = cap_required OR CAP_PERM_STORE_CAP;

    VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, AccType_NORMAL);

    if willabort == TRUE then
        AArch64.Abort(addr, AArch64.AlignmentFault(AccType_NORMAL, iswrite, secondstage));

    AArch64.CapabilityTag[addr, AccType_NORMAL] = tag;
    addr = addr + CAPABILITY_DBYTES;
```

4.4.124 STLR (capability, alternate base)

Store-Release capability via alternate base determines the base register to be used, derives an address from the base register, and stores a capability to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20			17	16	15	14			11	10	9 ... 5	4 ... 0
0	1	0	0	0	0	1	0	0	0	0	1	1	1	1	1	0	1	1	1	1	1	Rn	Ct
									L														

```
STLR    <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '0')
```

```
STLR    <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;

base = AltBaseReg[n];
data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);

MemC[addr, acctype] = data;
```

4.4.125 STLR (capability, normal base)

Store-Release capability determines the base register to be used, derives an address from the base register, and stores a capability to the calculated address in memory. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

31 23	22	21	20 17	16	15	14 11	10	9 5	4 0
0 1 0 0 0 0 1 0 0	0	0	1 1 1 1	1	1	1 1 1 1	1	Rn	Ct
	L								

```
STLR    <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STLR    <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;

base = BaseReg[n];
data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);

MemC[addr, acctype] = data;
```

4.4.126 STLR (integer)

Store-Release Register via alternate base determines the base register to be used, derives an address from the base register and an offset register, and stores a register to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

31 … 23	22	21	20 … 17	16	15	14 … 10	9 … 5	4 … 0
0 1 0 0 0 0 1 0 0	0	1	1 1 1 1	1	1	1 1 1 1 1	Rn	Rt
	L							

```
STLR    <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '0')
```

```
STLR    <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
datasize=32;
regsize=32;
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

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

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress address;
bits(datasize) data;

base = AltBaseReg[n];
data = X[t];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, acctype);

Mem[addr, datasize DIV 8, acctype] = data;
```

4.4.127 STLRB

Store-Release Register Byte via alternate base determines the base register to be used, derives an address from the base register and an offset register, and stores a byte to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21	20			17	16	15	14				10	9 5	4 0
0	1	0	0	0	0	1	0	0	0	1	1	1	1	1	1	0	1	1	1	1	1	Rn	Rt
									L														

```
STLRB   <Wt>, [<Cn|CSP>] //  (PSTATE.C64 == '0')
```

```
STLRB   <Wt>, [<Xn|SP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
datasize=8;
regsize=32;
AccType acctype = AccType_ORDERED;
```

Assembler Symbols

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

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

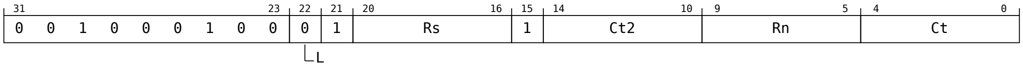
<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress address;
bits(datasize) data;

base = AltBaseReg[n];
data = X[t];
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, acctype);

Mem[addr, datasize DIV 8, acctype] = data;
```

4.4.128 STLXP

Store-Release Exclusive Pair of capabilities determines the base register to be used, derives an address from the base register, and stores two capabilities to the calculated address in memory. A 256-bit pair requires the address to be 256-bit aligned. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.



```
STLXP   <Ws>, <Ct>, <Ct2>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STLXP   <Ws>, <Ct>, <Ct2>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
integer s = UInt(Rs);
AccType acctype = AccType_ORDEREDATOMIC;
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field.

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data1;
Capability data2;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || s == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE       rt_unknown = FALSE;   // store original value
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE       rn_unknown = FALSE;   // address is original base
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if rt_unknown then
    data1 = Capability UNKNOWN;
    data2 = Capability UNKNOWN;
else
    data1 = C[t];
    data2 = C[t2];

```

```
if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];
bits(64) cap_required1 = CAP_PERM_STORE;
bits(64) cap_required2 = CAP_PERM_STORE;

if CapIsTagSet(data1) then
    cap_required1 = cap_required1 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data1) then
        cap_required1 = cap_required1 OR CAP_PERM_STORE_LOCAL;

if CapIsTagSet(data2) then
    cap_required2 = cap_required2 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data2) then
        cap_required2 = cap_required2 OR CAP_PERM_STORE_LOCAL;

bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required1, acctype);
VACheckAddress(base, addr + CAPABILITY_DBYTES<63:0>, CAPABILITY_DBYTES, cap_required2, acctype);

bit status = '1';
if AArch64.ExclusiveMonitorsPass(addr, CAPABILITY_DBYTES*2) then
    MemCP(addr, acctype, data1, data2);
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

4.4.129 STLXR

Store-Release Exclusive capability determines the base register to be used, derives an address from the base register, and stores a capability to the calculated address in memory. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. The instruction also has memory ordering semantics as described in Load-Acquire, Store-Release. For information about memory accesses, see Load/Store addressing modes.

31 23	22	21	20 16	15	14 10	9 5	4 0
0 0 1 0 0 0 1 0 0	0	0	Rs	1	1 1 1 1 1	Rn	Ct
	L						

```
STLXR   <Ws>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STLXR   <Ws>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
integer s = UInt(Rs);
AccType acctype = AccType_ORDEREDATOMIC;
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field.

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

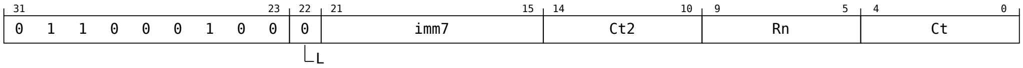
Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE       rt_unknown = FALSE;   // store original value
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE       rn_unknown = FALSE;   // address is original base
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

if rt_unknown then
    data = Capability UNKNOWN;
else
    data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
```

```
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);

bit status = '1';
if AArch64.ExclusiveMonitorsPass(addr, CAPABILITY_DBYTES) then
    MemC[addr, acctype] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

4.4.130 STNP

Store Pair of capabilities, with non-temporal hint determines the base register to be used, derives an address from the base register and an immediate offset, and stores two capabilities to memory from two Capability registers. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about Non-temporal pair instructions, see Load/Store Non-temporal pair. For information about memory accesses, see Load/Store addressing modes.



```
STNP    <Ct>, <Ct2>, [<Xn|SP>, #<imm>] //  (PSTATE.C64 == '0')
```

```
STNP    <Ct>, <Ct2>, [<Cn|CSP>, #<imm>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_STREAM;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data1;
Capability data2;

base = BaseReg[n];
bits(64) addr1 = VAddress(base) + offset;
bits(64) addr2 = addr1 + CAPABILITY_DBYTES<63:0>;

data1 = C[t];
data2 = C[t2];

bits(64) cap_required1 = CAP_PERM_STORE;
bits(64) cap_required2 = CAP_PERM_STORE;

if CapIsTagSet(data1) then
    cap_required1 = cap_required1 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data1) then
        cap_required1 = cap_required1 OR CAP_PERM_STORE_LOCAL;

if CapIsTagSet(data2) then
    cap_required2 = cap_required2 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data2) then
        cap_required2 = cap_required2 OR CAP_PERM_STORE_LOCAL;

VACheckAddress(base, addr1, CAPABILITY_DBYTES, cap_required1, acctype);
MemC[addr1, acctype] = data1;
VACheckAddress(base, addr2, CAPABILITY_DBYTES, cap_required2, acctype);
MemC[addr2, acctype] = data2;
```

4.4.131 STP (post-indexed)

Store Pair of capabilities (immediate post-index) determines the base register to be used, derives an address from the base register, and stores two capabilities to memory from two Capability registers. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.

31 … 23	22	21 … 15	14 … 10	9 … 5	4 … 0
0 0 1 0 0 0 1 0 1	0	imm7	Ct2	Rn	Ct
	L				

```
STP     <Ct>, <Ct2>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
STP     <Ct>, <Ct2>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_NORMAL;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

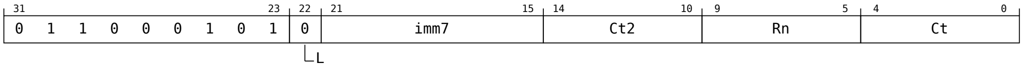
<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data1;
Capability data2;

boolean rt_unknown = FALSE;
if (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr1 = VAddress(base);
bits(64) addr2 = addr1 + CAPABILITY_DBYTES<63:0>;

if rt_unknown && t == n then
    data1 = Capability UNKNOWN;
else
    data1 = C[t];

if rt_unknown && t2 == n then
    data2 = Capability UNKNOWN;
else
    data2 = C[t2];

bits(64) cap_required1 = CAP_PERM_STORE;
bits(64) cap_required2 = CAP_PERM_STORE;

if CapIsTagSet(data1) then
    cap_required1 = cap_required1 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data1) then
        cap_required1 = cap_required1 OR CAP_PERM_STORE_LOCAL;

if CapIsTagSet(data2) then
    cap_required2 = cap_required2 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data2) then
        cap_required2 = cap_required2 OR CAP_PERM_STORE_LOCAL;

VACheckAddress(base, addr1, CAPABILITY_DBYTES, cap_required1, acctype);
MemC[addr1, acctype] = data1;
VACheckAddress(base, addr2, CAPABILITY_DBYTES, cap_required2, acctype);
MemC[addr2, acctype] = data2;

BaseReg[n] = VAAdd(base, offset);
```

4.4.132 STP (pre-indexed)

Store Pair of capabilities (immediate pre-index) determines the base register to be used, derives an address from the base register, and stores two capabilities to memory from two Capability registers. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.



```
STP     <Ct>, <Ct2>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
STP     <Ct>, <Ct2>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_NORMAL;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

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

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data1;
Capability data2;

boolean rt_unknown = FALSE;
if (t == n || t2 == n) && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;   // value stored is pre-writeback
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr1 = VAddress(base) + offset;
bits(64) addr2 = addr1 + CAPABILITY_DBYTES<63:0>;

if rt_unknown && t == n then
    data1 = Capability UNKNOWN;
else
    data1 = C[t];

if rt_unknown && t2 == n then
    data2 = Capability UNKNOWN;
else
    data2 = C[t2];

bits(64) cap_required1 = CAP_PERM_STORE;
bits(64) cap_required2 = CAP_PERM_STORE;

if CapIsTagSet(data1) then
    cap_required1 = cap_required1 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data1) then
        cap_required1 = cap_required1 OR CAP_PERM_STORE_LOCAL;

if CapIsTagSet(data2) then
    cap_required2 = cap_required2 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data2) then
        cap_required2 = cap_required2 OR CAP_PERM_STORE_LOCAL;

VACheckAddress(base, addr1, CAPABILITY_DBYTES, cap_required1, acctype);
MemC[addr1, acctype] = data1;
VACheckAddress(base, addr2, CAPABILITY_DBYTES, cap_required2, acctype);
MemC[addr2, acctype] = data2;

BaseReg[n] = VAAdd(base, offset);
```

4.4.133 STP (signed offset)

Store Pair of capabilities (signed offset) determines the base register to be used, derives an address from the base register, and stores two capabilities to memory from two Capability registers. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21 … 15	14 … 10	9 … 5	4 … 0
0	1	0	0	0	0	1	0	1	0	imm7	Ct2	Rn	Ct
									L				

```
STP     <Ct>, <Ct2>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STP     <Ct>, <Ct2>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
AccType acctype = AccType_NORMAL;
bits(64) offset = SignExtend(imm7:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

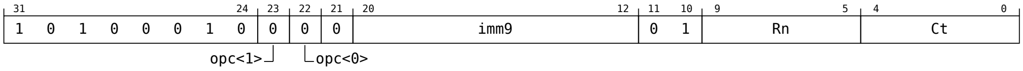
<imm> Is the optional signed immediate byte offset, a multiple of 16 in the range -1024 to 1008, defaulting to 0, encoded in the "imm7" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data1;
Capability data2;

base = BaseReg[n];
bits(64) addr1 = VAddress(base) + offset;
bits(64) addr2 = addr1 + CAPABILITY_DBYTES<63:0>;

data1 = C[t];
data2 = C[t2];

bits(64) cap_required1 = CAP_PERM_STORE;
bits(64) cap_required2 = CAP_PERM_STORE;

if CapIsTagSet(data1) then
    cap_required1 = cap_required1 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data1) then
        cap_required1 = cap_required1 OR CAP_PERM_STORE_LOCAL;

if CapIsTagSet(data2) then
    cap_required2 = cap_required2 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data2) then
        cap_required2 = cap_required2 OR CAP_PERM_STORE_LOCAL;

VACheckAddress(base, addr1, CAPABILITY_DBYTES, cap_required1, acctype);
MemC[addr1, acctype] = data1;
VACheckAddress(base, addr2, CAPABILITY_DBYTES, cap_required2, acctype);
MemC[addr2, acctype] = data2;
```

4.4.134 STR (post-indexed)

Store capability (immediate post-indexed) determines the base register to be used, derives an address from the base register, and stores a capability to memory from a Capability register. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.



```
STR     <Ct>, [<Xn|SP>], #<imm> //  (PSTATE.C64 == '0')
```

```
STR     <Ct>, [<Cn|CSP>], #<imm> //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

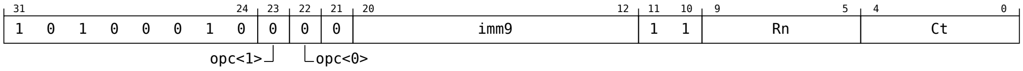
<imm> Is the signed immediate byte offset, a multiple of 16 in the range -4096 to 4080, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

boolean rt_unknown = FALSE;
if n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE       rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN    rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base);
if rt_unknown then
    data = Capability UNKNOWN;
else
    data = C[t];
bits(64) cap_required = CAP_PERM_STORE;

if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);
MemC[addr, acctype] = data;

BaseReg[n] = VAAdd(base,offset);
```

4.4.135 STR (pre-indexed)

Store capability (immediate pre-index) determines the base register to be used, derives an address from the base register, and stores a capability to memory from a Capability register. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.



```
STR     <Ct>, [<Xn|SP>, #<imm>]! //  (PSTATE.C64 == '0')
```

```
STR     <Ct>, [<Cn|CSP>, #<imm>]! //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the signed immediate byte offset, a multiple of 16 in the range -4096 to 4080, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

boolean rt_unknown = FALSE;
if n == t && n != 31 then
    c = ConstrainUnpredictable(Unpredictable_WBOVERLAPST);
    assert c IN {Constraint_NONE, Constraint_UNKNOWN, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_NONE        rt_unknown = FALSE;  // value stored is original value
        when Constraint_UNKNOWN     rt_unknown = TRUE;   // value stored is UNKNOWN
        when Constraint_UNDEF       UNDEFINED;
        when Constraint_NOP         EndOfInstruction();

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;
if rt_unknown then
    data = Capability UNKNOWN;
else
    data = C[t];
bits(64) cap_required = CAP_PERM_STORE;

if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);
MemC[addr, acctype] = data;

BaseReg[n] = VAAdd(base,offset);
```

4.4.136 STR (register offset, capability, alternate base)

Store capability (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, and stores a capability to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. The offset register can optionally be shifted and extended. For information about memory accesses, see Load/Store addressing modes.

31										21	20 … 16	15	14	13	12	11	10	9 … 5	4 … 0
1	1	0	0	0	0	1	0	1	1	1	Rm		1	sz	S	0	1	Rn	Ct
												sign				L			

```
STR     <Ct>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
STR     <Ct>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = LOG2_CAPABILITY_DBYTES;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> Is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#4

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
Capability data;

data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
```

```
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, AccType_NORMAL);
MemC[addr, AccType_NORMAL] = data;
```

4.4.137 STR (register offset, capability, normal base)

Store capability (register) determines the base register to be used, derives an address from the base register and an offset register, and stores a capability to the calculated address in memory. The offset register can optionally be shifted and extended. For information about memory accesses, see Load/Store addressing modes.

31	30	29	28	27	26	25	24	23	22	21	20–16	15	14	13	12	11	10	9–5	4–0
1	0	1	0	0	0	1	0	0	0	1	Rm		1	sz	S	1	0	Rn	Ct
								opc<1>	opc<0>			sign							

```
STR    <Ct>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
STR    <Ct>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = LOG2_CAPABILITY_DBYTES;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> Is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#4

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = BaseReg[n];
Capability data;

data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, AccType_NORMAL);
```

```
MemC[addr, AccType_NORMAL] = data;
```

4.4.138 STR (register offset, integer)

Store Register (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, and stores a word to the calculated address in memory. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

31								23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	0	0	0	0	1	0	1	0	1	Rm			1	sz	S	0	1	Rn		Rt	
									L				sign				opc					

```
STR     <Xt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
STR     <Xt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 3;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 64;
```

Word

31								23	22	21	20	16	15	14	13	12	11	10	9	5	4	0
1	0	0	0	0	0	1	0	1	0	1	Rm			1	sz	S	0	0	Rn		Rt	
									L				sign				opc					

```
STR     <Wt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
STR     <Wt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 2;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> For the doubleword variant: is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#3

For the word variant: is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#2

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.139 STR (register offset, SIMD&FP)

Store SIMD&FP Register (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, and stores a SIMD&FP register to the calculated address in memory. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: 32-bit and 64-bit

32-bit

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	0	1	Rm		1	sz	S	1	1	Rn	Rt
									L			sign				opc			

```
STR    <St>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
STR    <St>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 2;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

64-bit

31								23	22	21	20 16	15	14	13	12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	1	0	1	Rm		1	sz	S	1	0	Rn	Rt
									L			sign				opc			

```
STR    <Dt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
STR    <Dt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 3;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
```

Assembler Symbols

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> For the 32-bit variant: is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#2

For the 64-bit variant: is the index shift amount, encoded in"S":

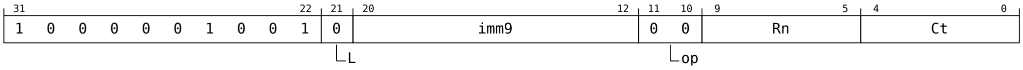
S	<amount>
0	[absent]
1	#3

Operation

```
CheckCapabilitiesEnabled();
CheckFPAdvSIMDEnabled64();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = V[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.140 STR (unsigned offset, capability, alternate base)

Store capability (unsigned offset) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a capability to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
STR     <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STR     <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 16 in the range 0 to 8176, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

Capability data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, AccType_NORMAL);
MemC[addr, AccType_NORMAL] = data;
```

4.4.141 STR (unsigned offset, capability, normal base)

Store capability (unsigned offset) stores a capability to memory from a Capability register. The address to use is derived from a base register value in A64 or capability base register in C64 and a immediate offset scaled by 16. For information about memory accesses, see Load/Store addressing modes.

31								23	22	21–10	9–5	4–0
1	1	0	0	0	0	1	0	0	0	imm12	Rn	Ct
									L			

```
STR     <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STR     <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm12:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional unsigned immediate byte offset, a multiple of 16 in the range 0 to 65520, defaulting to 0, encoded in the "imm12" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;
data = C[t];
bits(64) cap_required = CAP_PERM_STORE;

if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);
MemC[addr, acctype] = data;
```

4.4.142 STR (unsigned offset, integer)

Store Register (unsigned offset) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a 32-bit word or 64-bit doubleword to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

31									22	21	20 12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	0	1	0	imm9	1	1	Rn	Rt
										L			op		

```
STR     <Xt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STR     <Xt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9:'000', 64);
datasize = 64;
regsize  = 64;
```

Word

31									22	21	20 12	11	10	9 5	4 0
1	0	0	0	0	0	1	0	0	1	0	imm9	1	0	Rn	Rt
										L			op		

```
STR     <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STR     <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9:'00', 64);
datasize = 32;
regsize  = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> For the doubleword variant: is the optional unsigned immediate byte offset, a multiple of 8 in the range 0 to 4088, defaulting to 0, encoded in the "imm9" field.

For the word variant: is the optional unsigned immediate byte offset, a multiple of 4 in the range 0 to 2044, defaulting to 0, encoded in the "imm9" field.

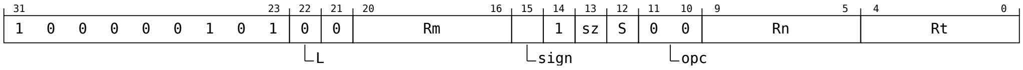
Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

```

```
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.143 STRB (register offset)

Store Register Byte (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, and stores a byte to the calculated address in memory. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
STRB    <Wt>, [<Cn|CSP>, <R><m>, <extend>] //  (PSTATE.C64 == '0')
```

```
STRB    <Wt>, [<Xn|SP>, <R><m>, <extend>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 0;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

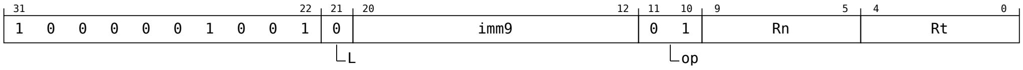
sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.144 STRB (unsigned offset)

Store Register Byte (unsigned offset) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a byte to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
STRB    <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STRB    <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = ZeroExtend(imm9, 64);
datasize = 8;
regsize  = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

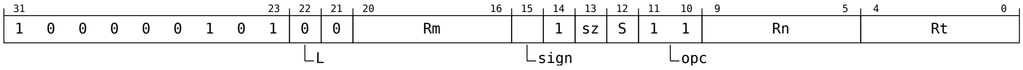
<imm> Is the optional unsigned immediate byte offset, in the range 0 to 511, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.145 STRH

Store Register Halfword (register) via alternate base determines the base register to be used, derives an address from the base register and an offset register, and stores a halfword to the calculated address in memory. The offset register can optionally be shifted and extended. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
STRH    <Wt>, [<Cn|CSP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '0')
```

```
STRH    <Wt>, [<Xn|SP>, <R><m>{, <extend><amount>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
integer m = UInt(Rm);
integer scale = 1;
ExtendType extend_type = DecodeRegExtend(sign:'1':sz);
integer shift = if S == '1' then scale else 0;
integer regsize = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<R> Is a width specifier, encoded in"sz":

sz	<R>
0	W
1	X

<m> Is the number [0-30] of the source general-purpose register or the name ZR (31), encoded in the "Rm" field.

<extend> Is the index extend and shift specifier, encoded in"sign:sz":

sign	sz	<extend>
0	0	UXTW
0	1	LSL
1	0	SXTW
1	1	SXTX

<amount> Is the index shift amount, encoded in"S":

S	<amount>
0	[absent]
1	#1

Operation

```
CheckCapabilitiesEnabled();

bits(64) offset = ExtendReg(m, extend_type, shift);
VirtualAddress base = AltBaseReg[n];
integer datasize = 8 << scale;

bits(64) addr = VAddress(base) + offset;
VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.146 STTR

Store capability (unprivileged) determines the base register to be used, derives an address from the base register and an immediate offset, and stores a capability to the calculated address in memory. For information about memory accesses, see Load/Store addressing modes. Memory accesses made by the instruction behave as if the instruction was executed at EL0 if the Effective value of PSTATE.UAO is 0 and either:

* The instruction is executed at EL1. * The instruction is executed at EL2 when the Effective value of both HCR_EL2.E2H and HCR_EL2.TGE are 1.

Otherwise, the memory access operates with the restrictions determined by the Exception level at which the instruction is executed.

In all cases the memory access operates with the capability restrictions as determined by the Exception level at which the instruction is executed.

31							24	23	22	21	20 12	11	10	9 5	4 0
1	0	1	0	0	0	1	0	0	0	0	imm9	1	0	Rn	Ct
								opc<1>	opc<0>						

```
STTR    <Ct>, [<Xn|SP>, #<imm>] //  (PSTATE.C64 == '0')
```

```
STTR    <Ct>, [<Cn|CSP>, #<imm>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9:'0000', 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

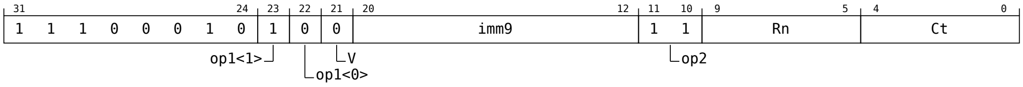
<imm> Is the signed immediate byte offset, a multiple of 16 in the range -4096 to 4080, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
unpriv_at_el1 = PSTATE.EL == EL1;
unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11';

user_access_override = HaveUAOExt() && PSTATE.UAO == '1';
if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
    acctype = AccType_UNPRIV;
else
    acctype = AccType_NORMAL;

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;
data = C[t];
bits(64) cap_required = CAP_PERM_STORE;

if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);
MemC[addr, acctype] = data;
```

4.4.147 STUR (capability, alternate base)

Store capability (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a capability to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
STUR    <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

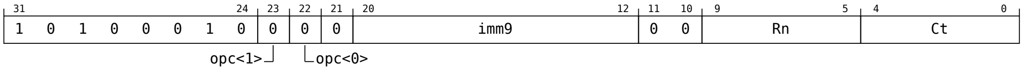
<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

Capability data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, AccType_NORMAL);
MemC[addr, AccType_NORMAL] = data;
```

4.4.148 STUR (capability, normal base)

Store capability (unscaled) determines the base register to be used, derives an address from the base register and an immediate offset, and stores a capability to the calculated address in memory. For information about memory accesses, see Load/Store addressing modes.



```
STUR    <Ct>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Ct>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
```

Assembler Symbols

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

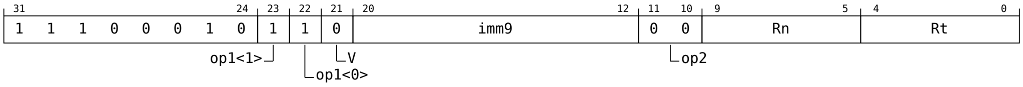
```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;
acctype = AccType_NORMAL;

base = BaseReg[n];
bits(64) addr = VAddress(base) + offset;
data = C[t];
bits(64) cap_required = CAP_PERM_STORE;

if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);
MemC[addr, acctype] = data;
```

4.4.149 STUR (integer)

Store Register (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a 32-bit word or 64-bit doubleword to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 2 classes: Doubleword and Word

Doubleword

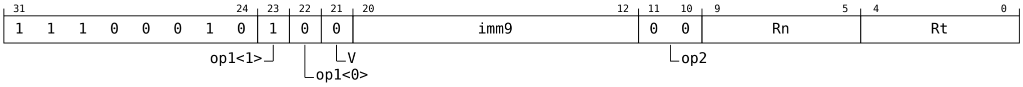


```
STUR    <Xt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Xt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 64;
regsize  = 64;
```

Word



```
STUR    <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 32;
regsize  = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xt> Is the 64-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

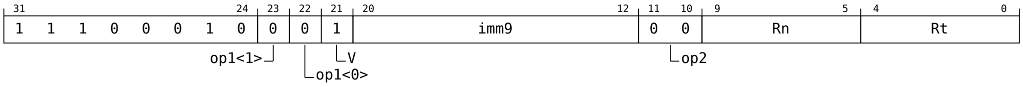
```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.150 STUR (SIMD&FP)

Store SIMD&FP Register (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a SIMD&FP register to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.

It has encodings from 5 classes: 8-bit , 16-bit , 32-bit , 64-bit and 128-bit

8-bit

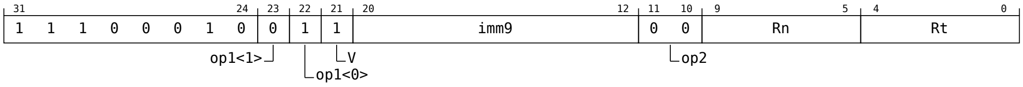


```
STUR    <Bt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Bt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 8;
```

16-bit

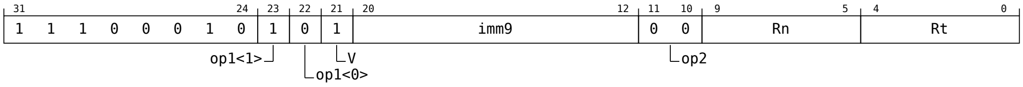


```
STUR    <Ht>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Ht>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 16;
```

32-bit

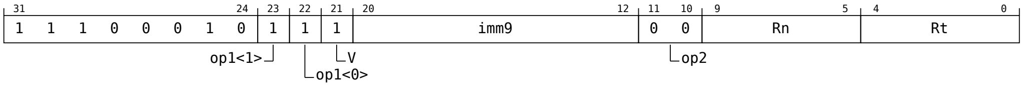


```
STUR    <St>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <St>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 32;
```

64-bit

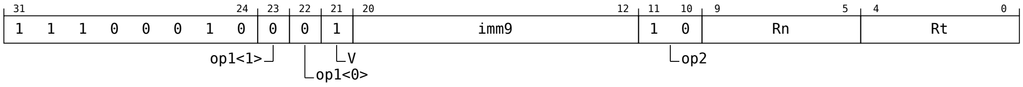


```
STUR    <Dt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Dt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 64;
```

128-bit



```
STUR    <Qt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STUR    <Qt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 128;
```

Assembler Symbols

<Bt> Is the 8-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Dt> Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Ht> Is the 16-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Qt> Is the 128-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<St> Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

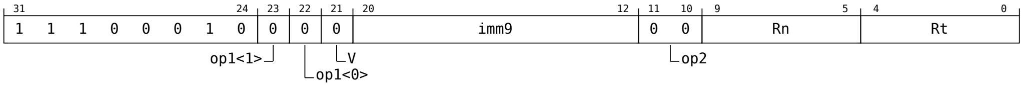
<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();
CheckFPAdvSIMDEnabled64();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = V[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.151 STURB

Store Register Byte (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a byte to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
STURB   <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STURB   <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 8;
regsize  = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

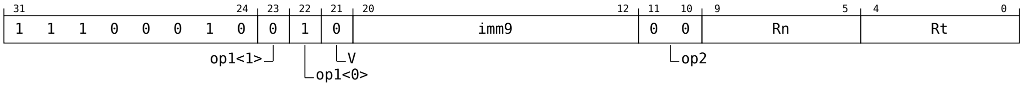
<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.152 STURH

Store Register Halfword (unscaled) via alternate base determines the base register to be used, derives an address from the base register and an immediate offset, and stores a halfword to the calculated address in memory. The base register used by this operation depends on PSTATE.C64: if PSTATE.C64 is 1, the base register is a 64-bit general-purpose register; if PSTATE.C64 is 0, the base register is a capability general-purpose register. For information about memory accesses, see Load/Store addressing modes.



```
STURH   <Wt>, [<Cn|CSP>{, #<imm>}] //  (PSTATE.C64 == '0')
```

```
STURH   <Wt>, [<Xn|SP>{, #<imm>}] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Rt);
integer n = UInt(Rn);
bits(64) offset = SignExtend(imm9, 64);
datasize = 16;
regsize  = 32;
```

Assembler Symbols

<Wt> Is the 32-bit name of the general-purpose register to be transferred, encoded in the "Rt" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

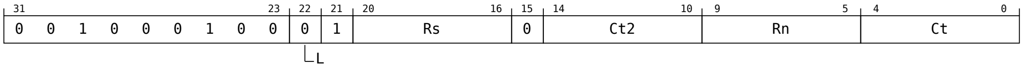
<imm> Is the optional signed immediate byte offset, in the range -256 to 255, defaulting to 0, encoded in the "imm9" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = AltBaseReg[n];
bits(64) addr = VAddress(base) + offset;

VACheckAddress(base, addr, datasize DIV 8, CAP_PERM_STORE, AccType_NORMAL);
bits(datasize) data = X[t];
Mem[addr, datasize DIV 8, AccType_NORMAL] = data;
```

4.4.153 STXP

Store Exclusive Pair of capabilities determines the base register to be used, derives an address from the base register, and stores two capabilities to the calculated address in memory. A 256-bit pair requires the address to be 256-bit aligned. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses, see Load/Store addressing modes.



```
STXP    <Ws>, <Ct>, <Ct2>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STXP    <Ws>, <Ct>, <Ct2>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer t2 = UInt(Ct2);
integer n = UInt(Rn);
integer s = UInt(Rs);
AccType acctype = AccType_ATOMIC;
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field.

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Ct2> Is the capability name of the second transfer register, encoded in the "Ct2" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data1;
Capability data2;
boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;

if s == t || s == t2 then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;    // store UNKNOWN value
        when Constraint_NONE       rt_unknown = FALSE;   // store original value
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;    // address is UNKNOWN
        when Constraint_NONE       rn_unknown = FALSE;   // address is original base
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if rt_unknown then
    data1 = Capability UNKNOWN;
    data2 = Capability UNKNOWN;
else
    data1 = C[t];
    data2 = C[t2];

if rn_unknown then
    base = VirtualAddress UNKNOWN;
```

```
else
    base = BaseReg[n];
bits(64) cap_required1 = CAP_PERM_STORE;
bits(64) cap_required2 = CAP_PERM_STORE;

if CapIsTagSet(data1) then
    cap_required1 = cap_required1 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data1) then
        cap_required1 = cap_required1 OR CAP_PERM_STORE_LOCAL;

if CapIsTagSet(data2) then
    cap_required2 = cap_required2 OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data2) then
        cap_required2 = cap_required2 OR CAP_PERM_STORE_LOCAL;

bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required1, acctype);
VACheckAddress(base, addr + CAPABILITY_DBYTES<63:0>, CAPABILITY_DBYTES, cap_required2, acctype);

bit status = '1';
if AArch64.ExclusiveMonitorsPass(addr, CAPABILITY_DBYTES*2) then
    MemCP(addr, acctype, data1, data2);
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

4.4.154 STXR

Store Exclusive capability determines the base register to be used, derives an address from the base register, and stores a capability to the calculated address in memory. The PE marks the physical address being accessed as an exclusive access. This exclusive access mark is checked by Store Exclusive instructions. See Synchronization and semaphores. For information about memory accesses, see Load/Store addressing modes.

31 … 23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0 0 1 0 0 0 1 0 0	0	0	Rs	0	1 1 1 1 1	Rn	Ct
	L						

```
STXR    <Ws>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
STXR    <Ws>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer n = UInt(Rn);
integer s = UInt(Rs);
AccType acctype = AccType_ATOMIC;
```

Assembler Symbols

<Ws> Is the 32-bit name of the general-purpose register into which the status result of the store exclusive is written, encoded in the "Rs" field.

<Ct> Is the capability name of the transfer register, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base;
Capability data;

boolean rt_unknown = FALSE;
boolean rn_unknown = FALSE;
if s == t then
    Constraint c = ConstrainUnpredictable(Unpredictable_DATAOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rt_unknown = TRUE;     // store UNKNOWN value
        when Constraint_NONE       rt_unknown = FALSE;    // store original value
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();
if s == n && n != 31 then
    Constraint c = ConstrainUnpredictable(Unpredictable_BASEOVERLAP);
    assert c IN {Constraint_UNKNOWN, Constraint_NONE, Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNKNOWN    rn_unknown = TRUE;     // address is UNKNOWN
        when Constraint_NONE       rn_unknown = FALSE;    // address is original base
        when Constraint_UNDEF      UNDEFINED;
        when Constraint_NOP        EndOfInstruction();

if rn_unknown then
    base = VirtualAddress UNKNOWN;
else
    base = BaseReg[n];

if rt_unknown then
    data = Capability UNKNOWN;
else
    data = C[t];
bits(64) cap_required = CAP_PERM_STORE;
if CapIsTagSet(data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
```

```
bits(64) addr = VAddress(base);
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, acctype);

bit status = '1';
if AArch64.ExclusiveMonitorsPass(addr, CAPABILITY_DBYTES) then
    MemC[addr, acctype] = data;
    status = ExclusiveMonitorsStatus();
X[s] = ZeroExtend(status, 32);
```

4.4.155 SUB

Subtract (immediate) copies a capability from the source Capability register to the destination Capability register with an optionally shifted immediate value subtracted from the value field. If the result is not representable the destination Capability register tag is cleared. If the source capability is sealed, the Capability Tag written to the destination Capability register is cleared.

31 … 24	23	22	21 … 10	9 … 5	4 … 0
0 0 0 0 0 0 1 0	1	sh	imm12	Cn	Cd
	A				

```
SUB     <Cd|CSP>, <Cn|CSP>, #<imm>{, LSL <amount>}
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
bits(64) imm;

case sh of
    when '0' imm = ZeroExtend(imm12, 64);
    when '1' imm = ZeroExtend(imm12 : Zeros(12), 64);
```

Assembler Symbols

<Cd|CSP> Is the capability name of the destination register or stack pointer, encoded in the "Cd" field.

<Cn|CSP> Is the capability name of the source register or stack pointer, encoded in the "Cn" field.

<imm> Is the unsigned immediate operand, in the range 0 to 4095, encoded in the "imm12" field.

<amount> Is the index shift amount, encoded in"sh":

sh	<amount>
0	#0
1	#12

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = if n == 31 then CSP[] else C[n];
integer    operand2 = UInt(imm);

Capability result = CapAdd(operand1, -operand2);

if CapIsSealed(operand1) then
    result = CapWithTagClear(result);

if d == 31 then
    CSP[] = result;
else
    C[d] = result;
```

4.4.156 SUBS

Subtract, setting flags if the Capability Tag of the first source Capability register is not the same as the Capability Tag of the second source Capability register subtracts the Capability Tag of the first source Capability register from the Capability Tag of the second source Capability register and writes the result to the destination 64-bit register otherwise subtracts the Value field of the first source Capability register from the Value field of the second source Capability register and writes the result to the destination 64-bit register. The instruction updates the condition flags based on the result.

This instruction is used by the alias CMP.

31										21	20 16	15					10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	Cm	1	0	0	1	1	0	Cn	Rd

```
SUBS    <Xd>, <Cn>, <Cm>
```

```
integer d = UInt(Rd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Xd> Is the 64-bit name of the destination general-purpose register, encoded in the "Rd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

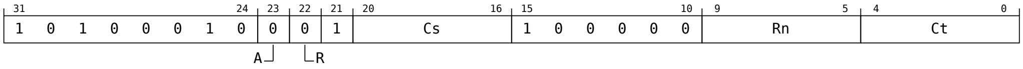
<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

Capability operand1 = C[n];
Capability operand2 = C[m];

boolean tag1 = CapIsTagSet(operand1);
boolean tag2 = CapIsTagSet(operand2);
bits(64) result;
bits(4) nzcv;

if tag1 != tag2 then
    bits(2) interim;
    bits(2) tvalue1 = if tag1 then '01' else '00';
    bits(2) tvalue2 = if tag2 then '01' else '00';
    (interim, nzcv) = AddWithCarry(tvalue1, NOT(tvalue2), '1');
    result = ZeroExtend(interim,64);
else
    bits(64) value1 = CapGetValue(operand1);
    bits(64) value2 = CapGetValue(operand2);
    (result, nzcv) = AddWithCarry(value1, NOT(value2), '1');

PSTATE.<N,Z,C,V> = nzcv;
X[d] = result;
```

4.4.157 SWP

Swap capabilities in memory determines the base register to be used, derives an address from the base register, atomically loads a Capability register from the calculated address in memory, and atomically stores another Capability register back to the same calculated address. The Capability register initially loaded from the calculated address in memory is returned to the destination Capability register.



```
SWP     <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWP     <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
AccType ldacctype = AccType_ATOMICRW;
AccType stacctype = AccType_ATOMICRW;
```

Assembler Symbols

<Cs> Is the capability name of the register to be stored, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be loaded, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

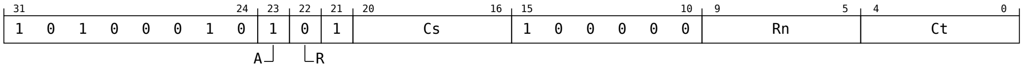
<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = BaseReg[n];
Capability data;
Capability store_data;

bits(64) addr = VAddress(base);
store_data = C[s];
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required =  CAP_PERM_STORE;
if CapIsTagSet(store_data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(store_data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

data = MemAtomicC(addr, MemAtomicOp_SWP, store_data, ldacctype, stacctype);
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.158 SWPA

Swap capabilities in memory with acquire determines the base register to be used, derives an address from the base register, atomically loads a Capability register from the calculated address in memory, and atomically stores another Capability register back to the same calculated address. The Capability register initially loaded from the calculated address in memory is returned to the destination Capability register. This instruction loads from memory with acquire semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release.



```
SWPA    <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPA    <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
AccType ldacctype = if Ct != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = AccType_ATOMICRW;
```

Assembler Symbols

<Cs> Is the capability name of the register to be stored, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be loaded, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = BaseReg[n];
Capability data;
Capability store_data;

bits(64) addr = VAddress(base);
store_data = C[s];
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required =  CAP_PERM_STORE;
if CapIsTagSet(store_data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(store_data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

data = MemAtomicC(addr, MemAtomicOp_SWP, store_data, ldacctype, stacctype);
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.159 SWPAL

Swap capabilities in memory with acquire and release determines the base register to be used, derives an address from the base register, atomically loads a Capability register from the calculated address in memory, and atomically stores another Capability register back to the same calculated address. The Capability register initially loaded from the calculated address in memory is returned to the destination Capability register. This instruction loads from memory with acquire and release semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release.

31 … 24	23	22	21	20 … 16	15 … 10	9 … 5	4 … 0
1 0 1 0 0 0 1 0	1	1	1	Cs	1 0 0 0 0 0	Rn	Ct
	A	R					

```
SWPAL   <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPAL   <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
AccType ldacctype = if Ct != '11111' then AccType_ORDEREDATOMICRW else AccType_ATOMICRW;
AccType stacctype = AccType_ORDEREDATOMICRW;
```

Assembler Symbols

<Cs> Is the capability name of the register to be stored, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be loaded, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = BaseReg[n];
Capability data;
Capability store_data;

bits(64) addr = VAddress(base);
store_data = C[s];
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required =  CAP_PERM_STORE;
if CapIsTagSet(store_data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(store_data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

data = MemAtomicC(addr, MemAtomicOp_SWP, store_data, ldacctype, stacctype);
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.160 SWPL

Swap capabilities in memory with release determines the base register to be used, derives an address from the base register. atomically loads a Capability register from the calculated address in memory, and atomically stores another Capability register back to the same calculated address. The Capability register initially loaded from the calculated address in memory is returned to the destination Capability register. This instruction loads from memory with release semantics as described in Load-Acquire, Load-AcquirePC, and Store-Release.

31 24	23	22	21	20 16	15 10	9 5	4 0
1 0 1 0 0 0 1 0	0	1	1	Cs	1 0 0 0 0 0	Rn	Ct
	A	R					

```
SWPL    <Cs>, <Ct>, [<Xn|SP>] //  (PSTATE.C64 == '0')
```

```
SWPL    <Cs>, <Ct>, [<Cn|CSP>] //  (PSTATE.C64 == '1')
```

```
integer t = UInt(Ct);
integer s = UInt(Cs);
integer n = UInt(Rn);
AccType ldacctype = AccType_ATOMICRW;
AccType stacctype = AccType_ORDEREDATOMICRW;
```

Assembler Symbols

<Cs> Is the capability name of the register to be stored, encoded in the "Cs" field.

<Ct> Is the capability name of the register to be loaded, encoded in the "Ct" field.

<Xn|SP> Is the 64-bit name of the general-purpose base register or stack pointer, encoded in the "Rn" field.

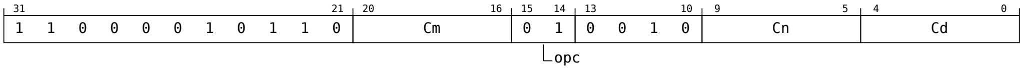
<Cn|CSP> Is the capability name of the base register or stack pointer, encoded in the "Rn" field.

Operation

```
CheckCapabilitiesEnabled();

VirtualAddress base = BaseReg[n];
Capability data;
Capability store_data;

bits(64) addr = VAddress(base);
store_data = C[s];
VACheckAddress(base, addr, CAPABILITY_DBYTES, CAP_PERM_LOAD, ldacctype);
bits(64) cap_required =  CAP_PERM_STORE;
if CapIsTagSet(store_data) then
    cap_required = cap_required OR CAP_PERM_STORE_CAP;
    if CapIsLocal(store_data) then
        cap_required = cap_required OR CAP_PERM_STORE_LOCAL;
VACheckAddress(base, addr, CAPABILITY_DBYTES, cap_required, stacctype);

data = MemAtomicC(addr, MemAtomicOp_SWP, store_data, ldacctype, stacctype);
data = CapSquashPostLoadCap(data, base);

C[t] = data;
```

4.4.161 UNSEAL

Unseal Capability unseals a capability with an unsealing capability, by checking the ObjectType of the capability against the Capability Value of the unsealing capability, and writes the result to the destination Capability register.



```
UNSEAL  <Cd>, <Cn>, <Cm>
```

```
integer d = UInt(Cd);
integer n = UInt(Cn);
integer m = UInt(Cm);
```

Assembler Symbols

<Cd> Is the capability name of the destination register, encoded in the "Cd" field.

<Cn> Is the capability name of the first source register, encoded in the "Cn" field.

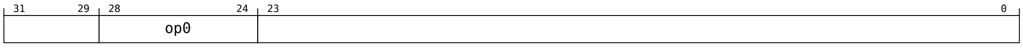
<Cm> Is the capability name of the second source register, encoded in the "Cm" field.

Operation

```
CheckCapabilitiesEnabled();

bits(64) value = CapGetValue(C[m]);
bits(64) otype = CapGetObjectType(C[n]);

Capability c = CapUnseal(C[n]);

if !CapCheckPermissions(C[m], CAP_PERM_GLOBAL) then
    c = CapClearPerms(c, CAP_PERM_GLOBAL);

if  CapIsTagSet(C[n]) && CapIsTagSet(C[m]) &&
    CapIsSealed(C[n]) && !CapIsSealed(C[m]) &&
    CapCheckPermissions(C[m], CAP_PERM_UNSEAL) &&
    CapIsInBounds(C[m]) &&
    otype == value then

    C[d] = c;
else
    C[d] = CapWithTagClear(c);
```

4.5 Index by encoding

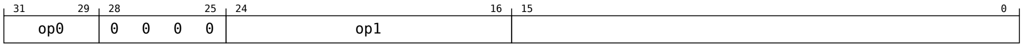
Top-level encodings for A64



op0	Instruction details
0000x	Reserved
00010	Morello encodings
00011	UNALLOCATED
001xx	UNALLOCATED
100xx	Data Processing – Immediate
101xx	Branches, Exception Generating and System instructions
x1x0x	Loads and Stores
x101x	Data Processing – Register
x111x	Data Processing – Scalar Floating-Point and Advanced SIMD

Reserved

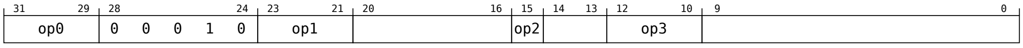
These instructions are under the top-level.



op0	op1	Instruction details
000	000000000	UDF
000	0001xxxxx	UNALLOCATED
!= 000		UNALLOCATED

Morello encodings

These instructions are under the top-level.



op0	op1	op2	op3	Instruction details
000				Morello add/subtract capability
001				morello_load_store_misc_1
010				morello_load_store_misc_2
011				morello_load_store_misc_3
100	00x			LDR (literal)

op0	op1	op2	op3	Instruction details
100	01x			Morello load/store unsigned offset via alternate base
100	1xx			Morello load/store register via alternate base
101				morello_load_store_misc_4
110	0xx			Morello load/store unsigned offset
110	100			Morello get/set system register
110	101			ADD (extended register)
110	11x			morello_misc
111				Morello load/store unscaled immediate via alternate base

Morello add/subtract capability

These instructions are under Morello encodings.

31							24	23	22	21 … 10	9 … 5	4 … 0
0	0	0	0	0	0	1	0	A	sh	imm12	Cn	Cd

A	Instruction Details
0	ADD (immediate)
1	SUB

morello_load_store_misc_1

These instructions are under Morello encodings.

31							24	23	22 … 0
0	0	1	0	0	0	1	0	op0	

op0	Instruction details
0	Morello load/ exclusive
1	Morello load/store pair postindex

Morello load/ exclusive

These instructions are under morello_load_store_misc_1.

31								23	22	21	20 … 16	15	14 … 10	9 … 5	4 … 0
0	0	1	0	0	0	1	0	0	L	op	Rs	o2	Ct2	Rn	Ct

L	op	Rs	o2	Ct2	Instruction Details
0	0		0	11111	STXR
0	0		1	11111	STLXR
0	1		0		STXP
0	1		1		STLXP

L	op	Rs	o2	Ct2	Instruction Details
1	0	11111	0	11111	LDXR
1	0	11111	1	11111	LDAXR
1	1	11111	0		LDXP
1	1	11111	1		LDAXP

Morello load/store pair postindex

These instructions are under morello_load_store_misc_1.

31								23	22	21 15	14 10	9 5	4 0
0	0	1	0	0	0	1	0	1	L	imm7	Ct2	Rn	Ct

L	Instruction Details
0	STP (post-indexed)
1	LDP (post-indexed)

morello_load_store_misc_2

These instructions are under Morello encodings.

31							24	23	22	21	20 16	15	14 0
0	1	0	0	0	0	1	0	op0		op1		op2	

op0	op1	op2	Instruction details
0	0	0	Morello load/store acquire/release capability via alternate base
0	0	1	Morello load/store acquire/release
0	1		Morello load/store acquire/release via alternate base
1			Morello load/store pair

Morello load/store acquire/release capability via alternate base

These instructions are under morello_load_store_misc_2.

31								23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	0	0	1	0	0	L	0	Rs	0	Ct2	Rn	Ct

L	Rs	Ct2	Instruction Details
0	11111	11111	STLR (capability, alternate base)
1	11111	11111	LDAR (capability, alternate base)

Morello load/store acquire/release

These instructions are under morello_load_store_misc_2.

31								23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	0	0	1	0	0	L	0	Rs	1	Ct2	Rn	Ct

L	Rs	Ct2	Instruction Details
0	11111	11111	STLR (capability, normal base)
1	11111	11111	LDAR (capability, normal base)

Morello load/store acquire/release via alternate base

These instructions are under morello_load_store_misc_2.

31								23	22	21	20 16	15	14 10	9 5	4 0
0	1	0	0	0	0	1	0	0	L	1	Rs	op	Rt2	Rn	Rt

L	Rs	op	Rt2	Instruction Details
0	11111	0	11111	STLRB
0	11111	1	11111	STLR (integer)
1	11111	0	11111	LDARB
1	11111	1	11111	LDAR (integer)

Morello load/store pair

These instructions are under morello_load_store_misc_2.

31								23	22	21 15	14 10	9 5	4 0
0	1	0	0	0	0	1	0	1	L	imm7	Ct2	Rn	Ct

L	Instruction Details
0	STP (signed offset)
1	LDP (signed offset)

morello_load_store_misc_3

These instructions are under Morello encodings.

31							24	23	22 0
0	1	1	0	0	0	1	0	op0	

op0	Instruction details
0	Morello load/store pair non-temporal
1	Morello load/store pair preindex

Morello load/store pair non-temporal

These instructions are under morello_load_store_misc_3.

31								23	22	21 15	14 10	9 5	4 0
0	1	1	0	0	0	1	0	0	L	imm7	Ct2	Rn	Ct

L	Instruction Details
0	STNP
1	LDNP

Morello load/store pair preindex

These instructions are under morello_load_store_misc_3.

31–23	22	21–15	14–10	9–5	4–0
0 1 1 0 0 0 1 0 1	L	imm7	Ct2	Rn	Ct

L	Instruction Details
0	STP (pre-indexed)
1	LDP (pre-indexed)

Morello load/store unsigned offset via alternate base

These instructions are under Morello encodings.

31–22	21	20–12	11–10	9–5	4–0
1 0 0 0 0 0 1 0 0 1	L	imm9	op	Rn	Rt

L	op	Instruction Details
0	00	STR (unsigned offset, capability, alternate base)
0	01	STRB (unsigned offset)
0	10	STR (unsigned offset, integer) — word
0	11	STR (unsigned offset, integer) — doubleword
1	00	LDR (unsigned offset, capability, alternate base)
1	01	LDRB (unsigned offset)
1	10	LDR (unsigned offset, integer) — word
1	11	LDR (unsigned offset, integer) — doubleword

Morello load/store register via alternate base

These instructions are under Morello encodings.

31–23	22	21	20–16	15	14	13	12	11–10	9–5	4–0
1 0 0 0 0 0 1 0 1	L	op	Rm	A	1	B	S	opc	Rn	Rt

L	op	opc	Instruction Details
0	0	00	STRB (register offset)
0	0	01	LDRSB — doubleword
0	0	10	LDRSH — doubleword
0	0	11	STRH
0	1	00	STR (register offset, integer) — word
0	1	01	STR (register offset, integer) — doubleword

L	op	opc	Instruction Details
0	1	10	STR (register offset, SIMD&FP) — 64-bit
0	1	11	STR (register offset, SIMD&FP) — 32-bit
1	0	00	LDRB (register offset)
1	0	01	LDRSB — word
1	0	10	LDRSH — word
1	0	11	LDRH
1	1	00	LDR (register offset, integer) — word
1	1	01	LDR (register offset, integer) — doubleword
1	1	10	LDR (register offset, SIMD&FP) — 64-bit
1	1	11	LDR (register offset, SIMD&FP) — 32-bit

morello_load_store_misc_4

These instructions are under Morello encodings.

31							24	23 22	21	20 16	15 10	9 0
1	0	1	0	0	0	1	0		op0		op1	

op0	op1	Instruction details
0	xxxx00	Morello load/store unscaled immediate
0	xxxx01	Morello load/store immediate postindex
0	xxxx10	Morello load/store immediate translated
0	xxxx11	Morello load/store immediate preindex
1	100000	Morello swap
1	110000	LDAPR
1	x11111	Morello compare and swap
1	x1xx10	Morello load/store register

Morello load/store unscaled immediate

These instructions are under morello_load_store_misc_4.

31							24	23 22	21	20 12	11	10	9 5	4 0
1	0	1	0	0	0	1	0	opc	0	imm9	0	0	Rn	Ct

opc	Instruction Details
00	STUR (capability, normal base)
01	LDUR (capability, normal base)

Morello load/store immediate postindex

These instructions are under morello_load_store_misc_4.

31							24	23 22	21	20 12	11	10	9 5	4 0
1	0	1	0	0	0	1	0	opc	0	imm9	0	1	Rn	Ct

opc	Instruction Details
00	STR (post-indexed)
01	LDR (post-indexed)

Morello load/store immediate translated

These instructions are under morello_load_store_misc_4.

31							24	23 22	21	20 12	11	10	9 5	4 0
1	0	1	0	0	0	1	0	opc	0	imm9	1	0	Rn	Ct

opc	Instruction Details
00	STTR
01	LDTR

Morello load/store immediate preindex

These instructions are under morello_load_store_misc_4.

31							24	23 22	21	20 12	11	10	9 5	4 0
1	0	1	0	0	0	1	0	opc	0	imm9	1	1	Rn	Ct

opc	Instruction Details
00	STR (pre-indexed)
01	LDR (pre-indexed)

Morello swap

These instructions are under morello_load_store_misc_4.

31							24	23 22	21	20 16	15					10	9 0
1	0	1	0	0	0	1	0	opc	1	Cs	1	0	0	0	0	0	opc2

opc	Instruction Details
00	SWP
01	SWPL
10	SWPA
11	SWPAL

Morello compare and swap

These instructions are under morello_load_store_misc_4.

31 24	23 22	21	20 15	14 10	9 0
1 0 1 0 0 0 1 0	opc	1	opc2	1 1 1 1 1	opc3

opc	opc2	Instruction Details
10	xxxxx0	CAS
10	xxxxx1	CASL
11	xxxxx0	CASA
11	xxxxx1	CASAL

Morello load/store register

These instructions are under morello_load_store_misc_4.

31 24	23 22	21	20 16	15	14	13	12	11	10	9 5	4 0
1 0 1 0 0 0 1 0	opc	1	Rm	A	1	B	S	1	0	Rn	Ct

opc	Instruction Details
00	STR (register offset, capability, normal base)
01	LDR (register offset, capability, normal base)

Morello load/store unsigned offset

These instructions are under Morello encodings.

31 23	22	21 10	9 5	4 0
1 1 0 0 0 0 1 0 0	L	imm12	Rn	Ct

L	Instruction Details
0	STR (unsigned offset, capability, normal base)
1	LDR (unsigned offset, capability, normal base)

Morello get/set system register

These instructions are under Morello encodings.

31 21	20	19	18 16	15 12	11 8	7 5	4 0
1 1 0 0 0 0 1 0 1 0 0	L	o0	op1	CRn	CRm	op2	Ct

L	Instruction Details
0	MSR
1	MRS

morello_misc

These instructions are under Morello encodings.

31									22	21 13	12 11	10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	op0	op1	op2		op3

op0	op1	op2	op3	Instruction details
000000xxx	10	0		Morello get field 1
0000010xx	10	0		Morello get field 2
0000011xx	10	0		Morello miscellaneous capability 0
0000100xx	10	0	00000	Morello branch
0000100xx	10	0	00001	Morello checks
0000100xx	10	0	00010	Morello branch sealed direct
0000100xx	10	0	00011	Morello branch restricted
0000110xx	10	0		SEAL (immediate)
0001000xx	10	0		Morello load pair and branch
0001001xx	10	0		Morello load/store tags
0001010xx	10	0		Morello convert to pointer
0001011xx	10	0		Morello convert to capability with implicit operand
000110xxx	10	0		CLRPERM (immediate)
0001110xx	10	0		Morello 1 src 1 dest
01xxxxxxx	10	0	0000x	Morello branch sealed indirect
0xxxxx0xx	00	0		Morello set field 1
0xxxxx0xx	00	1		Morello miscellaneous capability 1
0xxxxx10x	00	0		Morello set field 2
0xxxxx110	00	0		CVT (to pointer)
0xxxxx111	00	0		SCFLGS
0xxxxx1xx	00	1	00000	Morello branch to sealed
0xxxxx1xx	00	1	00001	Morello 2 src cap
0xxxxxxx0	01	0		Morello miscellaneous capability 2
0xxxxxxx0	11	0		Morello alignment
0xxxxxxx1	01	0		Morello bitwise
0xxxxxxx1	11	0		Morello immediate bounds
0xxxxxxxx	x1	1		CSEL
1xxxxx0x0	11	0		Morello convert to capability
1xxxxx100	11	0		SUBS
1xxxxxx1x		1		Morello load/store capability via alternate base

op0	op1	op2	op3	Instruction details
1xxxxxxxx	!= 11	0		Morello logical immediate

Morello get field 1

These instructions are under morello_misc.

31															16	15 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	0	opc	1	0	0	Cn	Rd

opc	Instruction Details
000	GCBASE
001	GCLEN
010	GCVALUE
011	GCOFF
100	GCTAG
101	GCSEAL
110	GCPERM
111	GCTYPE

Morello get field 2

These instructions are under morello_misc.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	1	0	opc	1	0	0	Cn	Rd

opc	Instruction Details
00	GCLIM
01	GCFLGS
10	CFHI

Morello miscellaneous capability 0

These instructions are under morello_misc.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	0	1	1	opc	1	0	0	Cn	Cd

opc	Instruction Details
00	CLRTAG
10	CPY

Morello branch

These instructions are under morello_misc.

31	30	29									20	19				15	14 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	opc	1	0	0	Cn	0	0	0	0	0

opc	Cn	Instruction Details
00		BR (indirect)
01		BLR (indirect)
10		RET
11	11111	BX

Morello checks

These instructions are under morello_misc.

31	30	29									20	19				15	14 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	opc	1	0	0	Cn	0	0	0	0	1

opc	Instruction Details
00	CHKSLD
01	CHKTGD

Morello branch sealed direct

These instructions are under morello_misc.

31	30	29									20	19				15	14 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	opc	1	0	0	Cn	0	0	0	1	0

opc	Instruction Details
00	BRS (capability)
01	BLRS (capability)
10	RETS (capability)

Morello branch restricted

These instructions are under morello_misc.

31	30	29									20	19				15	14 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	0	0	0	1	0	0	opc	1	0	0	Cn	0	0	0	1	1

opc	Instruction Details
00	BRR
01	BLRR
10	RETR

Morello load pair and branch

These instructions are under morello_misc.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	0	0	opc	1	0	0	Cn	Ct

opc	Instruction Details
00	LDPBR
01	LDPBLR

Morello load/store tags

These instructions are under morello_misc.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	0	1	opc	1	0	0	Cn	Ct

opc	Instruction Details
00	STCT
01	LDCT

Morello convert to pointer

These instructions are under morello_misc.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0	opc	1	0	0	Cn	Rd

opc	Instruction Details
00	CVTD (to pointer)
01	CVTP (to pointer)

Morello convert to capability with implicit operand

These instructions are under morello_misc.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	0	1	1	opc	1	0	0	Rn	Cd

opc	Instruction Details
00	CVTD (to capability)
01	CVTP (to capability)
10	CVTDZ
11	CVTPZ

Morello 1 src 1 dest

These instructions are under morello_misc.

31																15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	0	0	1	1	1	0	opc	1	0	0	Rn	Rd

opc	Instruction Details
00	RRLEN
01	RRMASK

Morello branch sealed indirect

These instructions are under morello_misc.

31	30	29									20	19 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	1	imm7	1	0	0	Cn	0	0	0	0	op

op	Instruction Details
0	BR (memory indirect)
1	BLR (memory indirect)

Morello set field 1

These instructions are under morello_misc.

31										21	20 16	15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Rm	0	opc	0	0	0	Cn	Cd

opc	Instruction Details
00	SCBNDS (register)
01	SCBNDSE
10	SCVALUE
11	SCOFF

Morello miscellaneous capability 1

These instructions are under morello_misc.

31										21	20 16	15	14 13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Cm	0	opc	0	0	1	Cn	Cd

opc	Instruction Details
00	BUILD
01	CPYTYPE
10	CSEAL
11	CPYVALUE

Morello set field 2

These instructions are under morello_misc.

31										21	20 16	15	14	13	12		10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Rm	1	0	op	0	0	0	Cn	Cd

op	Instruction Details
0	SCTAG
1	CLRPERM (register)

Morello branch to sealed

These instructions are under morello_misc.

31	30	29								21	20 16	15	14 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	Cm	1	opc	0	0	1	Cn	0	0	0	0	0

opc	Instruction Details
00	BRS (pair of capabilities)
01	BLRS (pair of capabilities)
10	RETS (pair of capabilities)

Morello 2 src cap

These instructions are under morello_misc.

31	30	29								21	20 16	15	14 13	12		10	9 5	4	3	2	1	0
1	1	0	0	0	0	1	0	1	1	0	Cm	1	opc	0	0	1	Cn	0	0	0	0	1

opc	Instruction Details
00	CHKSS
01	CHKEQ

Morello miscellaneous capability 2

These instructions are under morello_misc.

31										21	20 16	15 14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Cm	opc	0	0	1	0	Cn	Cd

opc	Instruction Details
00	SEAL (capability)
01	UNSEAL
10	CHKSSU

Morello alignment

These instructions are under morello_misc.

31										21	20 15	14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	imm6	U	0	1	1	0	Cn	Cd

U	Instruction Details
0	ALIGND
1	ALIGNU

Morello bitwise

These instructions are under morello_misc.

31										21	20 16	15 14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	Rm	opc	1	0	1	0	Cn	Cd

opc	Instruction Details
00	BICFLGS (register)
01	ORRFLGS (register)
10	EORFLGS (register)
11	CTHI

Morello immediate bounds

These instructions are under morello_misc.

31										21	20 15	14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	0	imm6	S	1	1	1	0	Cn	Cd

S	Instruction Details
0	SCBNDS (immediate) — Unscaled
1	SCBNDS (immediate) — Scaled

Morello convert to capability

These instructions are under morello_misc.

31										21	20 16	15	14	13			10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	Rm	0	op	0	1	1	0	Cn	Cd

op	Instruction Details
0	CVT (to capability)
1	CVTZ

Morello load/store capability via alternate base

These instructions are under morello_misc.

31										21	20 16	15	14	13	12	11	10	9 5	4 0
1	1	0	0	0	0	1	0	1	1	1	Rm	A	1	B	S	L	1	Rn	Ct

L	Instruction Details
0	STR (register offset, capability, alternate base)
1	LDR (register offset, capability, alternate base)

Morello logical immediate

These instructions are under morello_misc.

31	30	29	28	27	26	25	24	23	22	21	20 … 13	12 11	10	9 … 5	4 … 0
1	1	0	0	0	0	1	0	1	1	1	imm8	!= 11	0	Cn	Cd
												opc			

The following constraints also apply to this encoding: opc != 11 && opc != 11

opc	Instruction Details
00	BICFLGS (immediate)
01	ORRFLGS (immediate)
10	EORFLGS (immediate)

Morello load/store unscaled immediate via alternate base

These instructions are under Morello encodings.

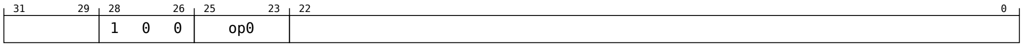
31	30	29	28	27	26	25	24	23 22	21	20 … 12	11 10	9 … 5	4 … 0
1	1	1	0	0	0	1	0	op1	V	imm9	op2	Rn	Rd

op1	V	op2	Instruction Details
00	0	00	STURB
00	0	01	LDURB
00	0	10	LDURSB — doubleword
00	0	11	LDURSB — word
00	1	00	STUR (SIMD&FP) — 8-bit
00	1	01	LDUR (SIMD&FP) — 8-bit
00	1	10	STUR (SIMD&FP) — 128-bit
00	1	11	LDUR (SIMD&FP) — 128-bit
01	0	00	STURH
01	0	01	LDURH
01	0	10	LDURSH — doubleword
01	0	11	LDURSH — word
01	1	00	STUR (SIMD&FP) — 16-bit
01	1	01	LDUR (SIMD&FP) — 16-bit
10	0	00	STUR (integer) — word
10	0	01	LDUR (integer) — word
10	0	10	LDURSW
10	0	11	STUR (capability, alternate base)
10	1	00	STUR (SIMD&FP) — 32-bit
10	1	01	LDUR (SIMD&FP) — 32-bit
11	0	00	STUR (integer) — doubleword
11	0	01	LDUR (integer) — doubleword
11	0	11	LDUR (capability, alternate base)

op1	V	op2	Instruction Details
11	1	00	STUR (SIMD&FP) — 64-bit
11	1	01	LDUR (SIMD&FP) — 64-bit

Data Processing – Immediate

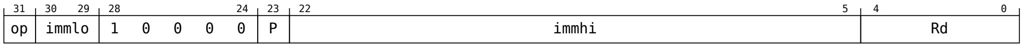
These instructions are under the top-level.



op0	Instruction details
00x	aarch64_adr
010	Add/subtract (immediate)
011	Add/subtract (immediate, with tags)
100	Logical (immediate)
101	Move wide (immediate)
110	Bitfield
111	Extract

aarch64_adr

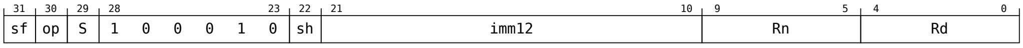
These instructions are under Data Processing – Immediate.



op	P	Instruction Details
0		ADR
1		ADRP
1	0	ADRDP
1	1	ADRP

Add/subtract (immediate)

These instructions are under Data Processing – Immediate.



sf	op	S	Instruction Details
0	0	0	ADD (immediate) — 32-bit
0	0	1	ADDS (immediate) — 32-bit
0	1	0	SUB (immediate) — 32-bit
0	1	1	SUBS (immediate) — 32-bit

sf	op	S	Instruction Details
1	0	0	ADD (immediate) — 64-bit
1	0	1	ADDS (immediate) — 64-bit
1	1	0	SUB (immediate) — 64-bit
1	1	1	SUBS (immediate) — 64-bit

Add/subtract (immediate, with tags)

These instructions are under Data Processing – Immediate.

31	30	29	28					23	22	21 16	15 14	13 10	9 5	4 0
sf	op	S	1	0	0	0	1	1	o2	uimm6	op3	uimm4	Rn	Rd

sf	S	o2	Instruction Details
		1	UNALLOCATED
0		0	UNALLOCATED
1	1	0	UNALLOCATED

Logical (immediate)

These instructions are under Data Processing – Immediate.

31	30 29	28					23	22	21 16	15 10	9 5	4 0
sf	opc	1	0	0	1	0	0	N	immr	imms	Rn	Rd

sf	opc	N	Instruction Details
0		1	UNALLOCATED
0	00	0	AND (immediate) — 32-bit
0	01	0	ORR (immediate) — 32-bit
0	10	0	EOR (immediate) — 32-bit
0	11	0	ANDS (immediate) — 32-bit
1	00		AND (immediate) — 64-bit
1	01		ORR (immediate) — 64-bit
1	10		EOR (immediate) — 64-bit
1	11		ANDS (immediate) — 64-bit

Move wide (immediate)

These instructions are under Data Processing – Immediate.

31	30 29	28					23	22 21	20 5	4 0
sf	opc	1	0	0	1	0	1	hw	imm16	Rd

sf	opc	hw	Instruction Details
	01		UNALLOCATED

sf	opc	hw	Instruction Details
0		1x	UNALLOCATED
0	00	0x	MOVN — 32-bit
0	10	0x	MOVZ — 32-bit
0	11	0x	MOVK — 32-bit
1	00		MOVN — 64-bit
1	10		MOVZ — 64-bit
1	11		MOVK — 64-bit

Bitfield

These instructions are under Data Processing – Immediate.

31	30 29	28					23	22	21 16	15 10	9 5	4 0
sf	opc	1	0	0	1	1	0	N	immr	imms	Rn	Rd

sf	opc	N	Instruction Details
	11		UNALLOCATED
0		1	UNALLOCATED
0	00	0	SBFM — 32-bit
0	01	0	BFM — 32-bit
0	10	0	UBFM — 32-bit
1		0	UNALLOCATED
1	00	1	SBFM — 64-bit
1	01	1	BFM — 64-bit
1	10	1	UBFM — 64-bit

Extract

These instructions are under Data Processing – Immediate.

31	30 29	28					23	22	21	20 16	15 10	9 5	4 0
sf	op21	1	0	0	1	1	1	N	o0	Rm	imms	Rn	Rd

sf	op21	N	o0	imms	Instruction Details
	x1				UNALLOCATED
	00		1		UNALLOCATED
	1x				UNALLOCATED
0				1xxxxx	UNALLOCATED
0		1			UNALLOCATED
0	00	0	0	0xxxxx	EXTR — 32-bit
1		0			UNALLOCATED

sf	op21	N	o0	imms	Instruction Details
1	00	1	0		EXTR — 64-bit

Branches, Exception Generating and System instructions

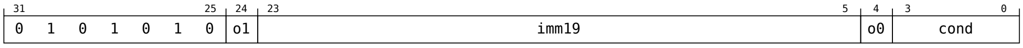
These instructions are under the top-level.

31 29	28 26	25 12	11 5	4 0
op0	1 0 1	op1		op2

op0	op1	op2	Instruction details
010	0xxxxxxxxxxxxx		Conditional branch (immediate)
010	1xxxxxxxxxxxxx		UNALLOCATED
110	00xxxxxxxxxxxx		Exception generation
110	010000000x000x		UNALLOCATED
110	010000000x001x		UNALLOCATED
110	0100000010000x		UNALLOCATED
110	0100000010001x		UNALLOCATED
110	01000000110000		UNALLOCATED
110	01000000110010	11111	Hints
110	01000000110010	!= 11111	UNALLOCATED
110	01000000110011		Barriers
110	01000001xx000x		UNALLOCATED
110	01000001xx001x		UNALLOCATED
110	0100000xxx0100		PSTATE
110	0100000xxx0101		UNALLOCATED
110	0100000xxx011x		UNALLOCATED
110	0100000xxx1xxx		UNALLOCATED
110	0100x01xxxxxxx		System instructions
110	0100x1xxxxxxxx		System register move
110	0101xxxxxxxxxx		UNALLOCATED
110	011xxxxxxxxxxx		UNALLOCATED
110	1xxxxxxxxxxxxx		Unconditional branch (register)
x00			Unconditional branch (immediate)
x01	0xxxxxxxxxxxxx		Compare and branch (immediate)
x01	1xxxxxxxxxxxxx		Test and branch (immediate)
x11			UNALLOCATED

Conditional branch (immediate)

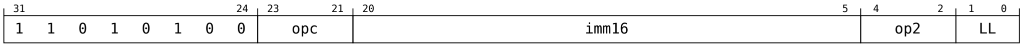
These instructions are under Branches, Exception Generating and System instructions.



o1	o0	Instruction Details
0	0	B.cond
0	1	UNALLOCATED
1		UNALLOCATED

Exception generation

These instructions are under Branches, Exception Generating and System instructions.



opc	op2	LL	Instruction Details
	001		UNALLOCATED
	01x		UNALLOCATED
	1xx		UNALLOCATED
000	000	00	UNALLOCATED
000	000	01	SVC
000	000	10	HVC
000	000	11	SMC
001	000	x1	UNALLOCATED
001	000	00	BRK
001	000	1x	UNALLOCATED
010	000	x1	UNALLOCATED
010	000	00	HLT
010	000	1x	UNALLOCATED
011	000	01	UNALLOCATED
011	000	1x	UNALLOCATED
100	000		UNALLOCATED
101	000	00	UNALLOCATED
101	000	01	DCPS1
101	000	10	DCPS2
101	000	11	DCPS3
110	000		UNALLOCATED
111	000		UNALLOCATED

Hints

These instructions are under Branches, Exception Generating and System instructions.

31																			12	11 8	7 5	4				0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	0	CRm	op2	1	1	1	1	1

CRm	op2	Instruction Details	Feature
		HINT	-
0000	000	NOP	-
0000	001	YIELD	-
0000	010	WFE	-
0000	011	WFI	-
0000	100	SEV	-
0000	101	SEVL	-
0010	000	ESB	FEAT_RAS
0010	001	PSB CSYNC	FEAT_SPE
0010	100	CSDB	-

Barriers

These instructions are under Branches, Exception Generating and System instructions.

31																			12	11 8	7 5	4 0
1	1	0	1	0	1	0	1	0	0	0	0	0	0	1	1	0	0	1	1	CRm	op2	Rt

CRm	op2	Rt	Instruction Details
	000		UNALLOCATED
	001	!= 11111	UNALLOCATED
	010	11111	CLREX
	101	11111	DMB
	110	11111	ISB
	111	!= 11111	UNALLOCATED
	111	11111	SB
!= 0x00	100	11111	DSB
0000	100	11111	SSBB
0001	011		UNALLOCATED
001x	011		UNALLOCATED
01xx	011		UNALLOCATED
0100	100	11111	PSSBB
1xxx	011		UNALLOCATED

PSTATE

These instructions are under Branches, Exception Generating and System instructions.

31 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1 0 1 0 1 0 0 0 0 0	op1	0 1 0 0	CRm	op2	Rt

Rt	Instruction Details
!= 11111	UNALLOCATED
11111	MSR (immediate)

System instructions

These instructions are under Branches, Exception Generating and System instructions.

31 22	21	20 19	18 16	15 12	11 8	7 5	4 0
1 1 0 1 0 1 0 1 0 0	L	0 1	op1	CRn	CRm	op2	Rt

L	Instruction Details
0	SYS
1	SYSL

System register move

These instructions are under Branches, Exception Generating and System instructions.

31 22	21	20	19	18 16	15 12	11 8	7 5	4 0
1 1 0 1 0 1 0 1 0 0	L	1	o0	op1	CRn	CRm	op2	Rt

L	Instruction Details
0	MSR (register)
1	MRS

Unconditional branch (register)

These instructions are under Branches, Exception Generating and System instructions.

31 25	24 21	20 16	15 10	9 5	4 0
1 1 0 1 0 1 1	opc	op2	op3	Rn	op4

opc	op2	op3	Rn	op4	Instruction Details
	!= 11111				UNALLOCATED
0000	11111	000000		!= 00000	UNALLOCATED
0000	11111	000000		00000	BR
0000	11111	000001			UNALLOCATED
0000	11111	000010		!= 11111	UNALLOCATED
0000	11111	000011		!= 11111	UNALLOCATED
0000	11111	0001xx			UNALLOCATED

opc	op2	op3	Rn	op4	Instruction Details
0000	11111	001xxx			UNALLOCATED
0000	11111	01xxxx			UNALLOCATED
0000	11111	1xxxxx			UNALLOCATED
0001	11111	000000		!= 00000	UNALLOCATED
0001	11111	000000		00000	BLR
0001	11111	000001			UNALLOCATED
0001	11111	000010		!= 11111	UNALLOCATED
0001	11111	000011		!= 11111	UNALLOCATED
0001	11111	0001xx			UNALLOCATED
0001	11111	001xxx			UNALLOCATED
0001	11111	01xxxx			UNALLOCATED
0001	11111	1xxxxx			UNALLOCATED
0010	11111	000000		!= 00000	UNALLOCATED
0010	11111	000000		00000	RET
0010	11111	000001			UNALLOCATED
0010	11111	000010	!= 11111	!= 11111	UNALLOCATED
0010	11111	000011	!= 11111	!= 11111	UNALLOCATED
0010	11111	0001xx			UNALLOCATED
0010	11111	001xxx			UNALLOCATED
0010	11111	01xxxx			UNALLOCATED
0010	11111	1xxxxx			UNALLOCATED
0011	11111				UNALLOCATED
0100	11111	000000	!= 11111	!= 00000	UNALLOCATED
0100	11111	000000	!= 11111	00000	UNALLOCATED
0100	11111	000000	11111	!= 00000	UNALLOCATED
0100	11111	000000	11111	00000	ERET
0100	11111	000001			UNALLOCATED
0100	11111	000010	!= 11111	!= 11111	UNALLOCATED
0100	11111	000010	!= 11111	11111	UNALLOCATED
0100	11111	000010	11111	!= 11111	UNALLOCATED
0100	11111	000011	!= 11111	!= 11111	UNALLOCATED
0100	11111	000011	!= 11111	11111	UNALLOCATED
0100	11111	000011	11111	!= 11111	UNALLOCATED
0100	11111	0001xx			UNALLOCATED

opc	op2	op3	Rn	op4	Instruction Details
0100	11111	001xxx			UNALLOCATED
0100	11111	01xxxx			UNALLOCATED
0100	11111	1xxxxx			UNALLOCATED
0101	11111	!= 000000			UNALLOCATED
0101	11111	000000	!= 11111	!= 00000	UNALLOCATED
0101	11111	000000	!= 11111	00000	UNALLOCATED
0101	11111	000000	11111	!= 00000	UNALLOCATED
0101	11111	000000	11111	00000	DRPS
011x	11111				UNALLOCATED
1000	11111	00000x			UNALLOCATED
1000	11111	0001xx			UNALLOCATED
1000	11111	001xxx			UNALLOCATED
1000	11111	01xxxx			UNALLOCATED
1000	11111	1xxxxx			UNALLOCATED
1001	11111	00000x			UNALLOCATED
1001	11111	0001xx			UNALLOCATED
1001	11111	001xxx			UNALLOCATED
1001	11111	01xxxx			UNALLOCATED
1001	11111	1xxxxx			UNALLOCATED
101x	11111				UNALLOCATED
11xx	11111				UNALLOCATED

Unconditional branch (immediate)

These instructions are under Branches, Exception Generating and System instructions.

31	30				26	25	0
op	0	0	1	0	1	imm26	

op	Instruction Details
0	B
1	BL

Compare and branch (immediate)

These instructions are under Branches, Exception Generating and System instructions.

31	30					25	24	23	5	4	0
sf	0	1	1	0	1	0	op	imm19		Rt	

sf	op	Instruction Details
0	0	CBZ — 32-bit
0	1	CBNZ — 32-bit
1	0	CBZ — 64-bit
1	1	CBNZ — 64-bit

Test and branch (immediate)

These instructions are under Branches, Exception Generating and System instructions.

31	30					25	24	23 19	18 5	4 0
b5	0	1	1	0	1	1	op	b40	imm14	Rt

op	Instruction Details
0	TBZ
1	TBNZ

Loads and Stores

These instructions are under the top-level.

31 28	27	26	25	24 23	22	21 16	15 12	11 10	9 0
op0	1	op1	0	op2		op3		op4	

op0	op1	op2	op3	op4	Instruction details
0x00	1	00	000000		Advanced SIMD load/store multiple structures
0x00	1	01	0xxxxx		Advanced SIMD load/store multiple structures (post-indexed)
0x00	1	0x	1xxxxx		UNALLOCATED
0x00	1	10	x00000		Advanced SIMD load/store single structure
0x00	1	11			Advanced SIMD load/store single structure (post-indexed)
0x00	1	x0	x1xxxx		UNALLOCATED
0x00	1	x0	xx1xxx		UNALLOCATED
0x00	1	x0	xxx1xx		UNALLOCATED
0x00	1	x0	xxxx1x		UNALLOCATED
0x00	1	x0	xxxxx1		UNALLOCATED
0x01	0	1x	1xxxxx		UNALLOCATED
1001	0	1x	1xxxxx		UNALLOCATED
1x00	1				UNALLOCATED
xx00	0	0x			Load/store exclusive
xx00	0	1x			UNALLOCATED

op0	op1	op2	op3	op4	Instruction details
xx01	0	1x	0xxxxx	00	UNALLOCATED
xx01	1	1x	0xxxxx	00	UNALLOCATED
xx01		0x			Load register (literal)
xx10		00			Load/store no-allocate pair (offset)
xx10		01			Load/store register pair (post-indexed)
xx10		10			Load/store register pair (offset)
xx10		11			Load/store register pair (pre-indexed)
xx11		0x	0xxxxx	00	Load/store register (unscaled immediate)
xx11		0x	0xxxxx	01	Load/store register (immediate post-indexed)
xx11		0x	0xxxxx	10	Load/store register (unprivileged)
xx11		0x	0xxxxx	11	Load/store register (immediate pre-indexed)
xx11		0x	1xxxxx	00	Atomic memory operations
xx11		0x	1xxxxx	10	Load/store register (register offset)
xx11		0x	1xxxxx	x1	Load/store register (pac)
xx11		1x			Load/store register (unsigned immediate)

Advanced SIMD load/store multiple structures

These instructions are under Loads and Stores.

31	30	29						23	22	21					16	15 12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	0	L	0	0	0	0	0	0	opcode	size	Rn	Rt

L	opcode	Instruction Details
0	0000	ST4 (multiple structures)
0	0001	UNALLOCATED
0	0010	ST1 (multiple structures) — four registers
0	0011	UNALLOCATED
0	0100	ST3 (multiple structures)
0	0101	UNALLOCATED
0	0110	ST1 (multiple structures) — three registers
0	0111	ST1 (multiple structures) — one register
0	1000	ST2 (multiple structures)

L	opcode	Instruction Details
0	1001	UNALLOCATED
0	1010	ST1 (multiple structures) — two registers
0	1011	UNALLOCATED
0	11xx	UNALLOCATED
1	0000	LD4 (multiple structures)
1	0001	UNALLOCATED
1	0010	LD1 (multiple structures) — four registers
1	0011	UNALLOCATED
1	0100	LD3 (multiple structures)
1	0101	UNALLOCATED
1	0110	LD1 (multiple structures) — three registers
1	0111	LD1 (multiple structures) — one register
1	1000	LD2 (multiple structures)
1	1001	UNALLOCATED
1	1010	LD1 (multiple structures) — two registers
1	1011	UNALLOCATED
1	11xx	UNALLOCATED

Advanced SIMD load/store multiple structures (post-indexed)

These instructions are under Loads and Stores.

31	30	29						23	22	21	20 16	15 12	11 10	9 5	4 0
0	Q	0	0	1	1	0	0	1	L	0	Rm	opcode	size	Rn	Rt

L	Rm	opcode	Instruction Details
0		0001	UNALLOCATED
0		0011	UNALLOCATED
0		0101	UNALLOCATED
0		1001	UNALLOCATED
0		1011	UNALLOCATED
0		11xx	UNALLOCATED
0	!= 11111	0000	ST4 (multiple structures) — register offset
0	!= 11111	0010	ST1 (multiple structures) — four registers, register offset

L	Rm	opcode	Instruction Details
0	!= 11111	0100	ST3 (multiple structures) — register offset
0	!= 11111	0110	ST1 (multiple structures) — three registers, register offset
0	!= 11111	0111	ST1 (multiple structures) — one register, register offset
0	!= 11111	1000	ST2 (multiple structures) — register offset
0	!= 11111	1010	ST1 (multiple structures) — two registers, register offset
0	11111	0000	ST4 (multiple structures) — immediate offset
0	11111	0010	ST1 (multiple structures) — four registers, immediate offset
0	11111	0100	ST3 (multiple structures) — immediate offset
0	11111	0110	ST1 (multiple structures) — three registers, immediate offset
0	11111	0111	ST1 (multiple structures) — one register, immediate offset
0	11111	1000	ST2 (multiple structures) — immediate offset
0	11111	1010	ST1 (multiple structures) — two registers, immediate offset
1		0001	UNALLOCATED
1		0011	UNALLOCATED
1		0101	UNALLOCATED
1		1001	UNALLOCATED
1		1011	UNALLOCATED
1		11xx	UNALLOCATED
1	!= 11111	0000	LD4 (multiple structures) — register offset
1	!= 11111	0010	LD1 (multiple structures) — four registers, register offset
1	!= 11111	0100	LD3 (multiple structures) — register offset
1	!= 11111	0110	LD1 (multiple structures) — three registers, register offset
1	!= 11111	0111	LD1 (multiple structures) — one register, register offset
1	!= 11111	1000	LD2 (multiple structures) — register offset

L	Rm	opcode	Instruction Details
1	!= 11111	1010	LD1 (multiple structures) — two registers, register offset
1	11111	0000	LD4 (multiple structures) — immediate offset
1	11111	0010	LD1 (multiple structures) — four registers, immediate offset
1	11111	0100	LD3 (multiple structures) — immediate offset
1	11111	0110	LD1 (multiple structures) — three registers, immediate offset
1	11111	0111	LD1 (multiple structures) — one register, immediate offset
1	11111	1000	LD2 (multiple structures) — immediate offset
1	11111	1010	LD1 (multiple structures) — two registers, immediate offset

Advanced SIMD load/store single structure

These instructions are under Loads and Stores.

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 0	L	R	0 0 0 0 0	opcode	S	size	Rn	Rt

L	R	opcode	S	size	Instruction Details
0		11x			UNALLOCATED
0	0	000			ST1 (single structure) — 8-bit
0	0	001			ST3 (single structure) — 8-bit
0	0	010		x0	ST1 (single structure) — 16-bit
0	0	010		x1	UNALLOCATED
0	0	011		x0	ST3 (single structure) — 16-bit
0	0	011		x1	UNALLOCATED
0	0	100		00	ST1 (single structure) — 32-bit
0	0	100		1x	UNALLOCATED
0	0	100	0	01	ST1 (single structure) — 64-bit
0	0	100	1	01	UNALLOCATED
0	0	101		00	ST3 (single structure) — 32-bit
0	0	101		10	UNALLOCATED
0	0	101	0	01	ST3 (single structure) — 64-bit
0	0	101	0	11	UNALLOCATED
0	0	101	1	x1	UNALLOCATED

L	R	opcode	S	size	Instruction Details
0	1	000			ST2 (single structure) — 8-bit
0	1	001			ST4 (single structure) — 8-bit
0	1	010		x0	ST2 (single structure) — 16-bit
0	1	010		x1	UNALLOCATED
0	1	011		x0	ST4 (single structure) — 16-bit
0	1	011		x1	UNALLOCATED
0	1	100		00	ST2 (single structure) — 32-bit
0	1	100		10	UNALLOCATED
0	1	100	0	01	ST2 (single structure) — 64-bit
0	1	100	0	11	UNALLOCATED
0	1	100	1	x1	UNALLOCATED
0	1	101		00	ST4 (single structure) — 32-bit
0	1	101		10	UNALLOCATED
0	1	101	0	01	ST4 (single structure) — 64-bit
0	1	101	0	11	UNALLOCATED
0	1	101	1	x1	UNALLOCATED
1	0	000			LD1 (single structure) — 8-bit
1	0	001			LD3 (single structure) — 8-bit
1	0	010		x0	LD1 (single structure) — 16-bit
1	0	010		x1	UNALLOCATED
1	0	011		x0	LD3 (single structure) — 16-bit
1	0	011		x1	UNALLOCATED
1	0	100		00	LD1 (single structure) — 32-bit
1	0	100		1x	UNALLOCATED
1	0	100	0	01	LD1 (single structure) — 64-bit
1	0	100	1	01	UNALLOCATED
1	0	101		00	LD3 (single structure) — 32-bit
1	0	101		10	UNALLOCATED
1	0	101	0	01	LD3 (single structure) — 64-bit
1	0	101	0	11	UNALLOCATED
1	0	101	1	x1	UNALLOCATED
1	0	110	0		LD1R
1	0	110	1		UNALLOCATED
1	0	111	0		LD3R

L	R	opcode	S	size	Instruction Details
1	0	111	1		UNALLOCATED
1	1	000			LD2 (single structure) — 8-bit
1	1	001			LD4 (single structure) — 8-bit
1	1	010		x0	LD2 (single structure) — 16-bit
1	1	010		x1	UNALLOCATED
1	1	011		x0	LD4 (single structure) — 16-bit
1	1	011		x1	UNALLOCATED
1	1	100		00	LD2 (single structure) — 32-bit
1	1	100		10	UNALLOCATED
1	1	100	0	01	LD2 (single structure) — 64-bit
1	1	100	0	11	UNALLOCATED
1	1	100	1	x1	UNALLOCATED
1	1	101		00	LD4 (single structure) — 32-bit
1	1	101		10	UNALLOCATED
1	1	101	0	01	LD4 (single structure) — 64-bit
1	1	101	0	11	UNALLOCATED
1	1	101	1	x1	UNALLOCATED
1	1	110	0		LD2R
1	1	110	1		UNALLOCATED
1	1	111	0		LD4R
1	1	111	1		UNALLOCATED

Advanced SIMD load/store single structure (post-indexed)

These instructions are under Loads and Stores.

31	30	29 23	22	21	20 16	15 13	12	11 10	9 5	4 0
0	Q	0 0 1 1 0 1 1	L	R	Rm	opcode	S	size	Rn	Rt

L	R	Rm	opcode	S	size	Instruction Details
0			11x			UNALLOCATED
0	0		010		x1	UNALLOCATED
0	0		011		x1	UNALLOCATED
0	0		100		1x	UNALLOCATED
0	0		100	1	01	UNALLOCATED
0	0		101		10	UNALLOCATED
0	0		101	0	11	UNALLOCATED
0	0		101	1	x1	UNALLOCATED

L	R	Rm	opcode	S	size	Instruction Details
0	0	!= 11111	000			ST1 (single structure) — 8-bit, register offset
0	0	!= 11111	001			ST3 (single structure) — 8-bit, register offset
0	0	!= 11111	010		x0	ST1 (single structure) — 16-bit, register offset
0	0	!= 11111	011		x0	ST3 (single structure) — 16-bit, register offset
0	0	!= 11111	100		00	ST1 (single structure) — 32-bit, register offset
0	0	!= 11111	100	0	01	ST1 (single structure) — 64-bit, register offset
0	0	!= 11111	101		00	ST3 (single structure) — 32-bit, register offset
0	0	!= 11111	101	0	01	ST3 (single structure) — 64-bit, register offset
0	0	11111	000			ST1 (single structure) — 8-bit, immediate offset
0	0	11111	001			ST3 (single structure) — 8-bit, immediate offset
0	0	11111	010		x0	ST1 (single structure) — 16-bit, immediate offset
0	0	11111	011		x0	ST3 (single structure) — 16-bit, immediate offset
0	0	11111	100		00	ST1 (single structure) — 32-bit, immediate offset
0	0	11111	100	0	01	ST1 (single structure) — 64-bit, immediate offset
0	0	11111	101		00	ST3 (single structure) — 32-bit, immediate offset
0	0	11111	101	0	01	ST3 (single structure) — 64-bit, immediate offset
0	1		010		x1	UNALLOCATED
0	1		011		x1	UNALLOCATED
0	1		100		10	UNALLOCATED
0	1		100	0	11	UNALLOCATED
0	1		100	1	x1	UNALLOCATED
0	1		101		10	UNALLOCATED
0	1		101	0	11	UNALLOCATED
0	1		101	1	x1	UNALLOCATED
0	1	!= 11111	000			ST2 (single structure) — 8-bit, register offset

L	R	Rm	opcode	S	size	Instruction Details
0	1	!= 11111	001			ST4 (single structure) — 8-bit, register offset
0	1	!= 11111	010		x0	ST2 (single structure) — 16-bit, register offset
0	1	!= 11111	011		x0	ST4 (single structure) — 16-bit, register offset
0	1	!= 11111	100		00	ST2 (single structure) — 32-bit, register offset
0	1	!= 11111	100	0	01	ST2 (single structure) — 64-bit, register offset
0	1	!= 11111	101		00	ST4 (single structure) — 32-bit, register offset
0	1	!= 11111	101	0	01	ST4 (single structure) — 64-bit, register offset
0	1	11111	000			ST2 (single structure) — 8-bit, immediate offset
0	1	11111	001			ST4 (single structure) — 8-bit, immediate offset
0	1	11111	010		x0	ST2 (single structure) — 16-bit, immediate offset
0	1	11111	011		x0	ST4 (single structure) — 16-bit, immediate offset
0	1	11111	100		00	ST2 (single structure) — 32-bit, immediate offset
0	1	11111	100	0	01	ST2 (single structure) — 64-bit, immediate offset
0	1	11111	101		00	ST4 (single structure) — 32-bit, immediate offset
0	1	11111	101	0	01	ST4 (single structure) — 64-bit, immediate offset
1	0		010		x1	UNALLOCATED
1	0		011		x1	UNALLOCATED
1	0		100		1x	UNALLOCATED
1	0		100	1	01	UNALLOCATED
1	0		101		10	UNALLOCATED
1	0		101	0	11	UNALLOCATED
1	0		101	1	x1	UNALLOCATED
1	0		110	1		UNALLOCATED
1	0		111	1		UNALLOCATED
1	0	!= 11111	000			LD1 (single structure) — 8-bit, register offset

L	R	Rm	opcode	S	size	Instruction Details
1	0	!= 11111	001			LD3 (single structure) — 8-bit, register offset
1	0	!= 11111	010		x0	LD1 (single structure) — 16-bit, register offset
1	0	!= 11111	011		x0	LD3 (single structure) — 16-bit, register offset
1	0	!= 11111	100		00	LD1 (single structure) — 32-bit, register offset
1	0	!= 11111	100	0	01	LD1 (single structure) — 64-bit, register offset
1	0	!= 11111	101		00	LD3 (single structure) — 32-bit, register offset
1	0	!= 11111	101	0	01	LD3 (single structure) — 64-bit, register offset
1	0	!= 11111	110	0		LD1R — register offset
1	0	!= 11111	111	0		LD3R — register offset
1	0	11111	000			LD1 (single structure) — 8-bit, immediate offset
1	0	11111	001			LD3 (single structure) — 8-bit, immediate offset
1	0	11111	010		x0	LD1 (single structure) — 16-bit, immediate offset
1	0	11111	011		x0	LD3 (single structure) — 16-bit, immediate offset
1	0	11111	100		00	LD1 (single structure) — 32-bit, immediate offset
1	0	11111	100	0	01	LD1 (single structure) — 64-bit, immediate offset
1	0	11111	101		00	LD3 (single structure) — 32-bit, immediate offset
1	0	11111	101	0	01	LD3 (single structure) — 64-bit, immediate offset
1	0	11111	110	0		LD1R — immediate offset
1	0	11111	111	0		LD3R — immediate offset
1	1		010		x1	UNALLOCATED
1	1		011		x1	UNALLOCATED
1	1		100		10	UNALLOCATED
1	1		100	0	11	UNALLOCATED
1	1		100	1	x1	UNALLOCATED
1	1		101		10	UNALLOCATED
1	1		101	0	11	UNALLOCATED

L	R	Rm	opcode	S	size	Instruction Details
1	1		101	1	x1	UNALLOCATED
1	1		110	1		UNALLOCATED
1	1		111	1		UNALLOCATED
1	1	!= 11111	000			LD2 (single structure) — 8-bit, register offset
1	1	!= 11111	001			LD4 (single structure) — 8-bit, register offset
1	1	!= 11111	010		x0	LD2 (single structure) — 16-bit, register offset
1	1	!= 11111	011		x0	LD4 (single structure) — 16-bit, register offset
1	1	!= 11111	100		00	LD2 (single structure) — 32-bit, register offset
1	1	!= 11111	100	0	01	LD2 (single structure) — 64-bit, register offset
1	1	!= 11111	101		00	LD4 (single structure) — 32-bit, register offset
1	1	!= 11111	101	0	01	LD4 (single structure) — 64-bit, register offset
1	1	!= 11111	110	0		LD2R — register offset
1	1	!= 11111	111	0		LD4R — register offset
1	1	11111	000			LD2 (single structure) — 8-bit, immediate offset
1	1	11111	001			LD4 (single structure) — 8-bit, immediate offset
1	1	11111	010		x0	LD2 (single structure) — 16-bit, immediate offset
1	1	11111	011		x0	LD4 (single structure) — 16-bit, immediate offset
1	1	11111	100		00	LD2 (single structure) — 32-bit, immediate offset
1	1	11111	100	0	01	LD2 (single structure) — 64-bit, immediate offset
1	1	11111	101		00	LD4 (single structure) — 32-bit, immediate offset
1	1	11111	101	0	01	LD4 (single structure) — 64-bit, immediate offset
1	1	11111	110	0		LD2R — immediate offset
1	1	11111	111	0		LD4R — immediate offset

Load/store exclusive

These instructions are under Loads and Stores.

31 30	29					24	23	22	21	20 16	15	14 10	9 5	4 0
size	0	0	1	0	0	0	o2	L	o1	Rs	o0	Rt2	Rn	Rt

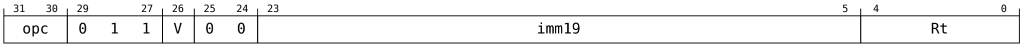
size	o2	L	o1	o0	Rt2	Instruction Details	Feature
	1		1		!= 11111	UNALLOCATED	-
0x	0		1		!= 11111	UNALLOCATED	-
00	0	0	0	0		STXRB	-
00	0	0	0	1		STLXRB	-
00	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASP	FEAT_LSE
00	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPL	FEAT_LSE
00	0	1	0	0		LDXRB	-
00	0	1	0	1		LDAXRB	-
00	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPA	FEAT_LSE
00	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 32-bit CASPAL	FEAT_LSE
00	1	0	0	0		STLLRB	FEAT_LOR
00	1	0	0	1		STLRB	-
00	1	0	1	0	11111	CASB, CASAB, CASALB, CASLB — CASB	FEAT_LSE
00	1	0	1	1	11111	CASB, CASAB, CASALB, CASLB — CASLB	FEAT_LSE
00	1	1	0	0		LDLARB	FEAT_LOR
00	1	1	0	1		LDARB	-
00	1	1	1	0	11111	CASB, CASAB, CASALB, CASLB — CASAB	FEAT_LSE
00	1	1	1	1	11111	CASB, CASAB, CASALB, CASLB — CASALB	FEAT_LSE
01	0	0	0	0		STXRH	-
01	0	0	0	1		STLXRH	-
01	0	0	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASP	FEAT_LSE
01	0	0	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPL	FEAT_LSE
01	0	1	0	0		LDXRH	-
01	0	1	0	1		LDAXRH	-
01	0	1	1	0	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPA	FEAT_LSE
01	0	1	1	1	11111	CASP, CASPA, CASPAL, CASPL — 64-bit CASPAL	FEAT_LSE
01	1	0	0	0		STLLRH	FEAT_LOR

size	o2	L	o1	o0	Rt2	Instruction Details	Feature
01	1	0	0	1		STLRH	-
01	1	0	1	0	11111	CASH, CASAH, CASALH, CASLH — CASH	FEAT_LSE
01	1	0	1	1	11111	CASH, CASAH, CASALH, CASLH — CASLH	FEAT_LSE
01	1	1	0	0		LDLARH	FEAT_LOR
01	1	1	0	1		LDARH	-
01	1	1	1	0	11111	CASH, CASAH, CASALH, CASLH — CASAH	FEAT_LSE
01	1	1	1	1	11111	CASH, CASAH, CASALH, CASLH — CASALH	FEAT_LSE
10	0	0	0	0		STXR — 32-bit	-
10	0	0	0	1		STLXR — 32-bit	-
10	0	0	1	0		STXP — 32-bit	-
10	0	0	1	1		STLXP — 32-bit	-
10	0	1	0	0		LDXR — 32-bit	-
10	0	1	0	1		LDAXR — 32-bit	-
10	0	1	1	0		LDXP — 32-bit	-
10	0	1	1	1		LDAXP — 32-bit	-
10	1	0	0	0		STLLR — 32-bit	FEAT_LOR
10	1	0	0	1		STLR — 32-bit	-
10	1	0	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CAS	FEAT_LSE
10	1	0	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASL	FEAT_LSE
10	1	1	0	0		LDLAR — 32-bit	FEAT_LOR
10	1	1	0	1		LDAR — 32-bit	-
10	1	1	1	0	11111	CAS, CASA, CASAL, CASL — 32-bit CASA	FEAT_LSE
10	1	1	1	1	11111	CAS, CASA, CASAL, CASL — 32-bit CASAL	FEAT_LSE
11	0	0	0	0		STXR — 64-bit	-
11	0	0	0	1		STLXR — 64-bit	-
11	0	0	1	0		STXP — 64-bit	-
11	0	0	1	1		STLXP — 64-bit	-
11	0	1	0	0		LDXR — 64-bit	-
11	0	1	0	1		LDAXR — 64-bit	-
11	0	1	1	0		LDXP — 64-bit	-

size	o2	L	o1	o0	Rt2	Instruction Details	Feature
11	0	1	1	1		LDAXP — 64-bit	-
11	1	0	0	0		STLLR — 64-bit	FEAT_LOR
11	1	0	0	1		STLR — 64-bit	-
11	1	0	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit CAS	FEAT_LSE
11	1	0	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASL	FEAT_LSE
11	1	1	0	0		LDLAR — 64-bit	FEAT_LOR
11	1	1	0	1		LDAR — 64-bit	-
11	1	1	1	0	11111	CAS, CASA, CASAL, CASL — 64-bit CASA	FEAT_LSE
11	1	1	1	1	11111	CAS, CASA, CASAL, CASL — 64-bit CASAL	FEAT_LSE

Load register (literal)

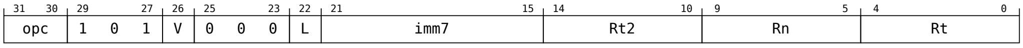
These instructions are under Loads and Stores.



opc	V	Instruction Details
00	0	LDR (literal) — 32-bit
00	1	LDR (literal, SIMD&FP) — 32-bit
01	0	LDR (literal) — 64-bit
01	1	LDR (literal, SIMD&FP) — 64-bit
10	0	LDRSW (literal)
10	1	LDR (literal, SIMD&FP) — 128-bit
11	0	PRFM (literal)
11	1	UNALLOCATED

Load/store no-allocate pair (offset)

These instructions are under Loads and Stores.



opc	V	L	Instruction Details
00	0	0	STNP — 32-bit
00	0	1	LDNP — 32-bit
00	1	0	STNP (SIMD&FP) — 32-bit
00	1	1	LDNP (SIMD&FP) — 32-bit

opc	V	L	Instruction Details
01	0		UNALLOCATED
01	1	0	STNP (SIMD&FP) — 64-bit
01	1	1	LDNP (SIMD&FP) — 64-bit
10	0	0	STNP — 64-bit
10	0	1	LDNP — 64-bit
10	1	0	STNP (SIMD&FP) — 128-bit
10	1	1	LDNP (SIMD&FP) — 128-bit
11			UNALLOCATED

Load/store register pair (post-indexed)

These instructions are under Loads and Stores.

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	V	0	0	1	L	imm7	Rt2	Rn	Rt

opc	V	L	Instruction Details
00	0	0	STP — 32-bit
00	0	1	LDP — 32-bit
00	1	0	STP (SIMD&FP) — 32-bit
00	1	1	LDP (SIMD&FP) — 32-bit
01	0	1	LDPSW
01	1	0	STP (SIMD&FP) — 64-bit
01	1	1	LDP (SIMD&FP) — 64-bit
10	0	0	STP — 64-bit
10	0	1	LDP — 64-bit
10	1	0	STP (SIMD&FP) — 128-bit
10	1	1	LDP (SIMD&FP) — 128-bit
11			UNALLOCATED

Load/store register pair (offset)

These instructions are under Loads and Stores.

31 30	29		27	26	25		23	22	21 15	14 10	9 5	4 0
opc	1	0	1	V	0	1	0	L	imm7	Rt2	Rn	Rt

opc	V	L	Instruction Details
00	0	0	STP — 32-bit
00	0	1	LDP — 32-bit
00	1	0	STP (SIMD&FP) — 32-bit

opc	V	L	Instruction Details
00	1	1	LDP (SIMD&FP) — 32-bit
01	0	1	LDPSW
01	1	0	STP (SIMD&FP) — 64-bit
01	1	1	LDP (SIMD&FP) — 64-bit
10	0	0	STP — 64-bit
10	0	1	LDP — 64-bit
10	1	0	STP (SIMD&FP) — 128-bit
10	1	1	LDP (SIMD&FP) — 128-bit
11			UNALLOCATED

Load/store register pair (pre-indexed)

These instructions are under Loads and Stores.

31 30	29 27	26	25 23	22	21 15	14 10	9 5	4 0
opc	1 0 1	V	0 1 1	L	imm7	Rt2	Rn	Rt

opc	V	L	Instruction Details
00	0	0	STP — 32-bit
00	0	1	LDP — 32-bit
00	1	0	STP (SIMD&FP) — 32-bit
00	1	1	LDP (SIMD&FP) — 32-bit
01	0	1	LDPSW
01	1	0	STP (SIMD&FP) — 64-bit
01	1	1	LDP (SIMD&FP) — 64-bit
10	0	0	STP — 64-bit
10	0	1	LDP — 64-bit
10	1	0	STP (SIMD&FP) — 128-bit
10	1	1	LDP (SIMD&FP) — 128-bit
11			UNALLOCATED

Load/store register (unscaled immediate)

These instructions are under Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	V	0 0	opc	0	imm9	0 0	Rn	Rt

size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STURB

size	V	opc	Instruction Details
00	0	01	LDURB
00	0	10	LDURSB — 64-bit
00	0	11	LDURSB — 32-bit
00	1	00	STUR (SIMD&FP) — 8-bit
00	1	01	LDUR (SIMD&FP) — 8-bit
00	1	10	STUR (SIMD&FP) — 128-bit
00	1	11	LDUR (SIMD&FP) — 128-bit
01	0	00	STURH
01	0	01	LDURH
01	0	10	LDURSH — 64-bit
01	0	11	LDURSH — 32-bit
01	1	00	STUR (SIMD&FP) — 16-bit
01	1	01	LDUR (SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STUR — 32-bit
10	0	01	LDUR — 32-bit
10	0	10	LDURSW
10	1	00	STUR (SIMD&FP) — 32-bit
10	1	01	LDUR (SIMD&FP) — 32-bit
11	0	00	STUR — 64-bit
11	0	01	LDUR — 64-bit
11	0	10	PRFUM
11	1	00	STUR (SIMD&FP) — 64-bit
11	1	01	LDUR (SIMD&FP) — 64-bit

Load/store register (immediate post-indexed)

These instructions are under Loads and Stores.

31 30	29 27	26	25 24	23 22	21	20 12	11 10	9 5	4 0
size	1 1 1	V	0 0	opc	0	imm9	0 1	Rn	Rt

size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit

size	V	opc	Instruction Details
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	UNALLOCATED
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Load/store register (unprivileged)

These instructions are under Loads and Stores.

31 30	29		27	26	25	24	23 22	21	20 12	11	10	9 5	4 0
size	1	1	1	V	0	0	opc	0	imm9	1	0	Rn	Rt

size	V	opc	Instruction Details
	1		UNALLOCATED

size	V	opc	Instruction Details
00	0	00	STTRB
00	0	01	LDTRB
00	0	10	LDTRSB — 64-bit
00	0	11	LDTRSB — 32-bit
01	0	00	STTRH
01	0	01	LDTRH
01	0	10	LDTRSH — 64-bit
01	0	11	LDTRSH — 32-bit
1x	0	11	UNALLOCATED
10	0	00	STTR — 32-bit
10	0	01	LDTR — 32-bit
10	0	10	LDTRSW
11	0	00	STTR — 64-bit
11	0	01	LDTR — 64-bit
11	0	10	UNALLOCATED

Load/store register (immediate pre-indexed)

These instructions are under Loads and Stores.

31 30	29		27	26	25	24	23 22	21	20 12	11	10	9 5	4 0
size	1	1	1	V	0	0	opc	0	imm9	1	1	Rn	Rt

size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit

size	V	opc	Instruction Details
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	UNALLOCATED
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Atomic memory operations

These instructions are under Loads and Stores.

31 30	29 27	26	25 24	23	22	21	20 16	15	14 12	11 10	9 5	4 0
size	1 1 1	V	0 0	A	R	1	Rs	o3	opc	0 0	Rn	Rt

size	V	A	R	o3	opc	Instruction Details	Feature
	0			1	11x	UNALLOCATED	-
	0	0		1	100	UNALLOCATED	-
	0	0	1	1	001	UNALLOCATED	-
	0	0	1	1	010	UNALLOCATED	-
	0	0	1	1	011	UNALLOCATED	-
	0	0	1	1	101	UNALLOCATED	-
	0	1	0	1	001	UNALLOCATED	-
	0	1	0	1	010	UNALLOCATED	-
	0	1	0	1	011	UNALLOCATED	-
	0	1	0	1	101	UNALLOCATED	-
	0	1	1	1	001	UNALLOCATED	-

size	V	A	R	o3	opc	Instruction Details	Feature
	0	1	1	1	010	UNALLOCATED	-
	0	1	1	1	011	UNALLOCATED	-
	0	1	1	1	100	UNALLOCATED	-
	0	1	1	1	101	UNALLOCATED	-
	1					UNALLOCATED	-
00	0	0	0	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDB	FEAT_LSE
00	0	0	0	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRB	FEAT_LSE
00	0	0	0	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORB	FEAT_LSE
00	0	0	0	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETB	FEAT_LSE
00	0	0	0	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXB	FEAT_LSE
00	0	0	0	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINB	FEAT_LSE
00	0	0	0	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXB	FEAT_LSE
00	0	0	0	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINB	FEAT_LSE
00	0	0	0	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPB	FEAT_LSE
00	0	0	0	1	001	UNALLOCATED	-
00	0	0	0	1	010	UNALLOCATED	-
00	0	0	0	1	011	UNALLOCATED	-
00	0	0	0	1	101	UNALLOCATED	-
00	0	0	1	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDLB	FEAT_LSE
00	0	0	1	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRLB	FEAT_LSE
00	0	0	1	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORLB	FEAT_LSE
00	0	0	1	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETLB	FEAT_LSE
00	0	0	1	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXLB	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
00	0	0	1	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINLB	FEAT_LSE
00	0	0	1	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXLB	FEAT_LSE
00	0	0	1	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINLB	FEAT_LSE
00	0	0	1	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPLB	FEAT_LSE
00	0	1	0	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDAB	FEAT_LSE
00	0	1	0	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRAB	FEAT_LSE
00	0	1	0	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORAB	FEAT_LSE
00	0	1	0	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETAB	FEAT_LSE
00	0	1	0	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXAB	FEAT_LSE
00	0	1	0	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINAB	FEAT_LSE
00	0	1	0	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXAB	FEAT_LSE
00	0	1	0	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINAB	FEAT_LSE
00	0	1	0	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPAB	FEAT_LSE
00	0	1	0	1	100	LDAPRB	FEAT_LRCPC
00	0	1	1	0	000	LDADDB, LDADDAB, LDADDALB, LDADDLB — LDADDALB	FEAT_LSE
00	0	1	1	0	001	LDCLRB, LDCLRAB, LDCLRALB, LDCLRLB — LDCLRALB	FEAT_LSE
00	0	1	1	0	010	LDEORB, LDEORAB, LDEORALB, LDEORLB — LDEORALB	FEAT_LSE
00	0	1	1	0	011	LDSETB, LDSETAB, LDSETALB, LDSETLB — LDSETALB	FEAT_LSE
00	0	1	1	0	100	LDSMAXB, LDSMAXAB, LDSMAXALB, LDSMAXLB — LDSMAXALB	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
00	0	1	1	0	101	LDSMINB, LDSMINAB, LDSMINALB, LDSMINLB — LDSMINALB	FEAT_LSE
00	0	1	1	0	110	LDUMAXB, LDUMAXAB, LDUMAXALB, LDUMAXLB — LDUMAXALB	FEAT_LSE
00	0	1	1	0	111	LDUMINB, LDUMINAB, LDUMINALB, LDUMINLB — LDUMINALB	FEAT_LSE
00	0	1	1	1	000	SWPB, SWPAB, SWPALB, SWPLB — SWPALB	FEAT_LSE
01	0	0	0	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDH	FEAT_LSE
01	0	0	0	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRH	FEAT_LSE
01	0	0	0	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORH	FEAT_LSE
01	0	0	0	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETH	FEAT_LSE
01	0	0	0	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXH	FEAT_LSE
01	0	0	0	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINH	FEAT_LSE
01	0	0	0	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXH	FEAT_LSE
01	0	0	0	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINH	FEAT_LSE
01	0	0	0	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPH	FEAT_LSE
01	0	0	0	1	001	UNALLOCATED	-
01	0	0	0	1	010	UNALLOCATED	-
01	0	0	0	1	011	UNALLOCATED	-
01	0	0	0	1	101	UNALLOCATED	-
01	0	0	1	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDLH	FEAT_LSE
01	0	0	1	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRLH	FEAT_LSE
01	0	0	1	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORLH	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
01	0	0	1	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETLH	FEAT_LSE
01	0	0	1	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXLH	FEAT_LSE
01	0	0	1	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINLH	FEAT_LSE
01	0	0	1	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXLH	FEAT_LSE
01	0	0	1	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINLH	FEAT_LSE
01	0	0	1	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPLH	FEAT_LSE
01	0	1	0	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDAH	FEAT_LSE
01	0	1	0	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRAH	FEAT_LSE
01	0	1	0	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORAH	FEAT_LSE
01	0	1	0	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETAH	FEAT_LSE
01	0	1	0	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXAH	FEAT_LSE
01	0	1	0	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINAH	FEAT_LSE
01	0	1	0	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXAH	FEAT_LSE
01	0	1	0	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINAH	FEAT_LSE
01	0	1	0	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPAH	FEAT_LSE
01	0	1	0	1	100	LDAPRH	FEAT_LRCPC
01	0	1	1	0	000	LDADDH, LDADDAH, LDADDALH, LDADDLH — LDADDALH	FEAT_LSE
01	0	1	1	0	001	LDCLRH, LDCLRAH, LDCLRALH, LDCLRLH — LDCLRALH	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
01	0	1	1	0	010	LDEORH, LDEORAH, LDEORALH, LDEORLH — LDEORALH	FEAT_LSE
01	0	1	1	0	011	LDSETH, LDSETAH, LDSETALH, LDSETLH — LDSETALH	FEAT_LSE
01	0	1	1	0	100	LDSMAXH, LDSMAXAH, LDSMAXALH, LDSMAXLH — LDSMAXALH	FEAT_LSE
01	0	1	1	0	101	LDSMINH, LDSMINAH, LDSMINALH, LDSMINLH — LDSMINALH	FEAT_LSE
01	0	1	1	0	110	LDUMAXH, LDUMAXAH, LDUMAXALH, LDUMAXLH — LDUMAXALH	FEAT_LSE
01	0	1	1	0	111	LDUMINH, LDUMINAH, LDUMINALH, LDUMINLH — LDUMINALH	FEAT_LSE
01	0	1	1	1	000	SWPH, SWPAH, SWPALH, SWPLH — SWPALH	FEAT_LSE
10	0	0	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADD	FEAT_LSE
10	0	0	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLR	FEAT_LSE
10	0	0	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEOR	FEAT_LSE
10	0	0	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSET	FEAT_LSE
10	0	0	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAX	FEAT_LSE
10	0	0	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMIN	FEAT_LSE
10	0	0	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAX	FEAT_LSE
10	0	0	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMIN	FEAT_LSE
10	0	0	0	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWP	FEAT_LSE
10	0	0	0	1	001	UNALLOCATED	-
10	0	0	0	1	010	UNALLOCATED	-
10	0	0	0	1	011	UNALLOCATED	-
10	0	0	0	1	101	UNALLOCATED	-
10	0	0	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDL	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
10	0	0	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRL	FEAT_LSE
10	0	0	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORL	FEAT_LSE
10	0	0	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETL	FEAT_LSE
10	0	0	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXL	FEAT_LSE
10	0	0	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINL	FEAT_LSE
10	0	0	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXL	FEAT_LSE
10	0	0	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINL	FEAT_LSE
10	0	0	1	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPL	FEAT_LSE
10	0	1	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDA	FEAT_LSE
10	0	1	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRA	FEAT_LSE
10	0	1	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORA	FEAT_LSE
10	0	1	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETA	FEAT_LSE
10	0	1	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXA	FEAT_LSE
10	0	1	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINA	FEAT_LSE
10	0	1	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXA	FEAT_LSE
10	0	1	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINA	FEAT_LSE
10	0	1	0	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPA	FEAT_LSE
10	0	1	0	1	100	LDAPR — 32-bit	FEAT_LRCPC
10	0	1	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 32-bit LDADDAL	FEAT_LSE
10	0	1	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 32-bit LDCLRAL	FEAT_LSE
10	0	1	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 32-bit LDEORAL	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
10	0	1	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 32-bit LDSETAL	FEAT_LSE
10	0	1	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 32-bit LDSMAXAL	FEAT_LSE
10	0	1	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 32-bit LDSMINAL	FEAT_LSE
10	0	1	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 32-bit LDUMAXAL	FEAT_LSE
10	0	1	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 32-bit LDUMINAL	FEAT_LSE
10	0	1	1	1	000	SWP, SWPA, SWPAL, SWPL — 32-bit SWPAL	FEAT_LSE
11	0	0	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADD	FEAT_LSE
11	0	0	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLR	FEAT_LSE
11	0	0	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEOR	FEAT_LSE
11	0	0	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSET	FEAT_LSE
11	0	0	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAX	FEAT_LSE
11	0	0	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMIN	FEAT_LSE
11	0	0	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAX	FEAT_LSE
11	0	0	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMIN	FEAT_LSE
11	0	0	0	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWP	FEAT_LSE
11	0	0	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDL	FEAT_LSE
11	0	0	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRL	FEAT_LSE
11	0	0	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORL	FEAT_LSE
11	0	0	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETL	FEAT_LSE
11	0	0	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXL	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
11	0	0	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINL	FEAT_LSE
11	0	0	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXL	FEAT_LSE
11	0	0	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINL	FEAT_LSE
11	0	0	1	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPL	FEAT_LSE
11	0	1	0	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDA	FEAT_LSE
11	0	1	0	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRA	FEAT_LSE
11	0	1	0	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORA	FEAT_LSE
11	0	1	0	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETA	FEAT_LSE
11	0	1	0	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXA	FEAT_LSE
11	0	1	0	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINA	FEAT_LSE
11	0	1	0	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXA	FEAT_LSE
11	0	1	0	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINA	FEAT_LSE
11	0	1	0	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPA	FEAT_LSE
11	0	1	0	1	100	LDAPR — 64-bit	FEAT_LRCPC
11	0	1	1	0	000	LDADD, LDADDA, LDADDAL, LDADDL — 64-bit LDADDAL	FEAT_LSE
11	0	1	1	0	001	LDCLR, LDCLRA, LDCLRAL, LDCLRL — 64-bit LDCLRAL	FEAT_LSE
11	0	1	1	0	010	LDEOR, LDEORA, LDEORAL, LDEORL — 64-bit LDEORAL	FEAT_LSE
11	0	1	1	0	011	LDSET, LDSETA, LDSETAL, LDSETL — 64-bit LDSETAL	FEAT_LSE
11	0	1	1	0	100	LDSMAX, LDSMAXA, LDSMAXAL, LDSMAXL — 64-bit LDSMAXAL	FEAT_LSE
11	0	1	1	0	101	LDSMIN, LDSMINA, LDSMINAL, LDSMINL — 64-bit LDSMINAL	FEAT_LSE
11	0	1	1	0	110	LDUMAX, LDUMAXA, LDUMAXAL, LDUMAXL — 64-bit LDUMAXAL	FEAT_LSE

size	V	A	R	o3	opc	Instruction Details	Feature
11	0	1	1	0	111	LDUMIN, LDUMINA, LDUMINAL, LDUMINL — 64-bit LDUMINAL	FEAT_LSE
11	0	1	1	1	000	SWP, SWPA, SWPAL, SWPL — 64-bit SWPAL	FEAT_LSE

Load/store register (register offset)

These instructions are under Loads and Stores.

31 30	29		27	26	25	24	23 22	21	20 16	15 13	12	11	10	9 5	4 0
size	1	1	1	V	0	0	opc	1	Rm	option	S	1	0	Rn	Rt

size	V	opc	option	Instruction Details
x1	1	1x		UNALLOCATED
00	0	00	!= 011	STRB (register) — extended register
00	0	00	011	STRB (register) — shifted register
00	0	01	!= 011	LDRB (register) — extended register
00	0	01	011	LDRB (register) — shifted register
00	0	10	!= 011	LDRSB (register) — 64-bit with extended register offset
00	0	10	011	LDRSB (register) — 64-bit with shifted register offset
00	0	11	!= 011	LDRSB (register) — 32-bit with extended register offset
00	0	11	011	LDRSB (register) — 32-bit with shifted register offset
00	1	00	!= 011	STR (register, SIMD&FP)
00	1	00	011	STR (register, SIMD&FP)
00	1	01	!= 011	LDR (register, SIMD&FP)
00	1	01	011	LDR (register, SIMD&FP)
00	1	10		STR (register, SIMD&FP)
00	1	11		LDR (register, SIMD&FP)
01	0	00		STRH (register)
01	0	01		LDRH (register)
01	0	10		LDRSH (register) — 64-bit
01	0	11		LDRSH (register) — 32-bit
01	1	00		STR (register, SIMD&FP)
01	1	01		LDR (register, SIMD&FP)
1x	0	11		UNALLOCATED
1x	1	1x		UNALLOCATED

size	V	opc	option	Instruction Details
10	0	00		STR (register) — 32-bit
10	0	01		LDR (register) — 32-bit
10	0	10		LDRSW (register)
10	1	00		STR (register, SIMD&FP)
10	1	01		LDR (register, SIMD&FP)
11	0	00		STR (register) — 64-bit
11	0	01		LDR (register) — 64-bit
11	0	10		PRFM (register)
11	1	00		STR (register, SIMD&FP)
11	1	01		LDR (register, SIMD&FP)

Load/store register (pac)

These instructions are under Loads and Stores.

31 30	29		27	26	25	24	23	22	21	20 12	11	10	9 5	4 0
size	1	1	1	V	0	0	M	S	1	imm9	W	1	Rn	Rt

size	V	Instruction Details
!= 11		UNALLOCATED
11	1	UNALLOCATED

Load/store register (unsigned immediate)

These instructions are under Loads and Stores.

31 30	29		27	26	25	24	23 22	21 10	9 5	4 0
size	1	1	1	V	0	1	opc	imm12	Rn	Rt

size	V	opc	Instruction Details
x1	1	1x	UNALLOCATED
00	0	00	STRB (immediate)
00	0	01	LDRB (immediate)
00	0	10	LDRSB (immediate) — 64-bit
00	0	11	LDRSB (immediate) — 32-bit
00	1	00	STR (immediate, SIMD&FP) — 8-bit
00	1	01	LDR (immediate, SIMD&FP) — 8-bit
00	1	10	STR (immediate, SIMD&FP) — 128-bit
00	1	11	LDR (immediate, SIMD&FP) — 128-bit
01	0	00	STRH (immediate)

size	V	opc	Instruction Details
01	0	01	LDRH (immediate)
01	0	10	LDRSH (immediate) — 64-bit
01	0	11	LDRSH (immediate) — 32-bit
01	1	00	STR (immediate, SIMD&FP) — 16-bit
01	1	01	LDR (immediate, SIMD&FP) — 16-bit
1x	0	11	UNALLOCATED
1x	1	1x	UNALLOCATED
10	0	00	STR (immediate) — 32-bit
10	0	01	LDR (immediate) — 32-bit
10	0	10	LDRSW (immediate)
10	1	00	STR (immediate, SIMD&FP) — 32-bit
10	1	01	LDR (immediate, SIMD&FP) — 32-bit
11	0	00	STR (immediate) — 64-bit
11	0	01	LDR (immediate) — 64-bit
11	0	10	PRFM (immediate)
11	1	00	STR (immediate, SIMD&FP) — 64-bit
11	1	01	LDR (immediate, SIMD&FP) — 64-bit

Data Processing – Register

These instructions are under the top-level.

31	30	29	28	27		25	24 21	20 16	15 10	9 0
	op0		op1	1	0	1	op2		op3	

op0	op1	op2	op3	Instruction details
0	1	0110		Data-processing (2 source)
1	1	0110		Data-processing (1 source)
	0	0xxx		Logical (shifted register)
	0	1xx0		Add/subtract (shifted register)
	0	1xx1		Add/subtract (extended register)
	1	0000	000000	Add/subtract (with carry)
	1	0000	000011	UNALLOCATED
	1	0000	0001xx	UNALLOCATED
	1	0000	001xxx	UNALLOCATED

op0	op1	op2	op3	Instruction details
	1	0000	x00001	Rotate right into flags
	1	0000	xx0010	Evaluate into flags
	1	0010	xxxx0x	Conditional compare (register)
	1	0010	xxxx1x	Conditional compare (immediate)
	1	0100		Conditional select
	1	0xx1		UNALLOCATED
	1	1xxx		Data-processing (3 source)

Data-processing (2 source)

These instructions are under Data Processing – Register.

31	30	29	28							21	20 … 16	15 … 10	9 … 5	4 … 0
sf	0	S	1	1	0	1	0	1	1	0	Rm	opcode	Rn	Rd

sf	S	opcode	Instruction Details
		000001	UNALLOCATED
		011xxx	UNALLOCATED
		1xxxxx	UNALLOCATED
	0	00011x	UNALLOCATED
	0	001101	UNALLOCATED
	0	00111x	UNALLOCATED
	1	00001x	UNALLOCATED
	1	0001xx	UNALLOCATED
	1	001xxx	UNALLOCATED
	1	01xxxx	UNALLOCATED
0		000000	UNALLOCATED
0	0	000010	UDIV — 32-bit
0	0	000011	SDIV — 32-bit
0	0	00010x	UNALLOCATED
0	0	001000	LSLV — 32-bit
0	0	001001	LSRV — 32-bit
0	0	001010	ASRV — 32-bit
0	0	001011	RORV — 32-bit
0	0	001100	UNALLOCATED
0	0	010x11	UNALLOCATED
0	0	010000	CRC32B, CRC32H, CRC32W, CRC32X — CRC32B

sf	S	opcode	Instruction Details
0	0	010001	CRC32B, CRC32H, CRC32W, CRC32X — CRC32H
0	0	010010	CRC32B, CRC32H, CRC32W, CRC32X — CRC32W
0	0	010100	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CB
0	0	010101	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CH
0	0	010110	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CW
1	0	000010	UDIV — 64-bit
1	0	000011	SDIV — 64-bit
1	0	001000	LSLV — 64-bit
1	0	001001	LSRV — 64-bit
1	0	001010	ASRV — 64-bit
1	0	001011	RORV — 64-bit
1	0	010xx0	UNALLOCATED
1	0	010x0x	UNALLOCATED
1	0	010011	CRC32B, CRC32H, CRC32W, CRC32X — CRC32X
1	0	010111	CRC32CB, CRC32CH, CRC32CW, CRC32CX — CRC32CX

Data-processing (1 source)

These instructions are under Data Processing – Register.

31	30	29	28							21	20 16	15 10	9 5	4 0
sf	1	S	1	1	0	1	0	1	1	0	opcode2	opcode	Rn	Rd

sf	S	opcode2	opcode	Instruction Details
			1xxxxx	UNALLOCATED
		xxx1x		UNALLOCATED
		xx1xx		UNALLOCATED
		x1xxx		UNALLOCATED
		1xxxx		UNALLOCATED
	0	00000	00011x	UNALLOCATED
	0	00000	001xxx	UNALLOCATED
	0	00000	01xxxx	UNALLOCATED
	1			UNALLOCATED
0		00001		UNALLOCATED

sf	S	opcode2	opcode	Instruction Details
0	0	00000	000000	RBIT — 32-bit
0	0	00000	000001	REV16 — 32-bit
0	0	00000	000010	REV — 32-bit
0	0	00000	000011	UNALLOCATED
0	0	00000	000100	CLZ — 32-bit
0	0	00000	000101	CLS — 32-bit
1	0	00000	000000	RBIT — 64-bit
1	0	00000	000001	REV16 — 64-bit
1	0	00000	000010	REV32
1	0	00000	000011	REV — 64-bit
1	0	00000	000100	CLZ — 64-bit
1	0	00000	000101	CLS — 64-bit
1	0	00001	01001x	UNALLOCATED
1	0	00001	0101xx	UNALLOCATED
1	0	00001	011xxx	UNALLOCATED

Logical (shifted register)

These instructions are under Data Processing – Register.

31	30 29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	opc	0 1 0 1 0	shift	N	Rm	imm6	Rn	Rd

sf	opc	N	imm6	Instruction Details
0			1xxxxx	UNALLOCATED
0	00	0		AND (shifted register) — 32-bit
0	00	1		BIC (shifted register) — 32-bit
0	01	0		ORR (shifted register) — 32-bit
0	01	1		ORN (shifted register) — 32-bit
0	10	0		EOR (shifted register) — 32-bit
0	10	1		EON (shifted register) — 32-bit
0	11	0		ANDS (shifted register) — 32-bit
0	11	1		BICS (shifted register) — 32-bit
1	00	0		AND (shifted register) — 64-bit
1	00	1		BIC (shifted register) — 64-bit
1	01	0		ORR (shifted register) — 64-bit
1	01	1		ORN (shifted register) — 64-bit
1	10	0		EOR (shifted register) — 64-bit

sf	opc	N	imm6	Instruction Details
1	10	1		EON (shifted register) — 64-bit
1	11	0		ANDS (shifted register) — 64-bit
1	11	1		BICS (shifted register) — 64-bit

Add/subtract (shifted register)

These instructions are under Data Processing – Register.

31	30	29	28 24	23 22	21	20 16	15 10	9 5	4 0
sf	op	S	0 1 0 1 1	shift	0	Rm	imm6	Rn	Rd

sf	op	S	shift	imm6	Instruction Details
			11		UNALLOCATED
0				1xxxxx	UNALLOCATED
0	0	0			ADD (shifted register) — 32-bit
0	0	1			ADDS (shifted register) — 32-bit
0	1	0			SUB (shifted register) — 32-bit
0	1	1			SUBS (shifted register) — 32-bit
1	0	0			ADD (shifted register) — 64-bit
1	0	1			ADDS (shifted register) — 64-bit
1	1	0			SUB (shifted register) — 64-bit
1	1	1			SUBS (shifted register) — 64-bit

Add/subtract (extended register)

These instructions are under Data Processing – Register.

31	30	29	28 24	23 22	21	20 16	15 13	12 10	9 5	4 0
sf	op	S	0 1 0 1 1	opt	1	Rm	option	imm3	Rn	Rd

sf	op	S	opt	imm3	Instruction Details
				1x1	UNALLOCATED
				11x	UNALLOCATED
			x1		UNALLOCATED
			1x		UNALLOCATED
0	0	0	00		ADD (extended register) — 32-bit
0	0	1	00		ADDS (extended register) — 32-bit
0	1	0	00		SUB (extended register) — 32-bit
0	1	1	00		SUBS (extended register) — 32-bit
1	0	0	00		ADD (extended register) — 64-bit
1	0	1	00		ADDS (extended register) — 64-bit

sf	op	S	opt	imm3	Instruction Details
1	1	0	00		SUB (extended register) — 64-bit
1	1	1	00		SUBS (extended register) — 64-bit

Add/subtract (with carry)

These instructions are under Data Processing – Register.

31	30	29	28 21	20 16	15 10	9 5	4 0
sf	op	S	1 1 0 1 0 0 0 0	Rm	0 0 0 0 0 0	Rn	Rd

sf	op	S	Instruction Details
0	0	0	ADC — 32-bit
0	0	1	ADCS — 32-bit
0	1	0	SBC — 32-bit
0	1	1	SBCS — 32-bit
1	0	0	ADC — 64-bit
1	0	1	ADCS — 64-bit
1	1	0	SBC — 64-bit
1	1	1	SBCS — 64-bit

Rotate right into flags

These instructions are under Data Processing – Register.

31	30	29	28 21	20 15	14 10	9 5	4	3 0
sf	op	S	1 1 0 1 0 0 0 0	imm6	0 0 0 0 1	Rn	o2	mask

sf	op	S	o2	Instruction Details
0				UNALLOCATED
1	0	0		UNALLOCATED
1	0	1	1	UNALLOCATED
1	1			UNALLOCATED

Evaluate into flags

These instructions are under Data Processing – Register.

31	30	29	28 21	20 15	14	13 10	9 5	4	3 0
sf	op	S	1 1 0 1 0 0 0 0	opcode2	sz	0 0 1 0	Rn	o3	mask

sf	op	S	opcode2	o3	mask	Instruction Details
0	0	0				UNALLOCATED
0	0	1	!= 000000			UNALLOCATED
0	0	1	000000	0	!= 1101	UNALLOCATED

sf	op	S	opcode2	o3	mask	Instruction Details
0	0	1	000000	1		UNALLOCATED
0	1					UNALLOCATED
1						UNALLOCATED

Conditional compare (register)

These instructions are under Data Processing – Register.

31	30	29	28							21	20 16	15 12	11	10	9 5	4	3 0
sf	op	S	1	1	0	1	0	0	1	0	Rm	cond	0	o2	Rn	o3	nzcv

sf	op	S	o2	o3	Instruction Details
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	CCMN (register) — 32-bit
0	1	1	0	0	CCMP (register) — 32-bit
1	0	1	0	0	CCMN (register) — 64-bit
1	1	1	0	0	CCMP (register) — 64-bit

Conditional compare (immediate)

These instructions are under Data Processing – Register.

31	30	29	28							21	20 16	15 12	11	10	9 5	4	3 0
sf	op	S	1	1	0	1	0	0	1	0	imm5	cond	1	o2	Rn	o3	nzcv

sf	op	S	o2	o3	Instruction Details
				1	UNALLOCATED
			1		UNALLOCATED
		0			UNALLOCATED
0	0	1	0	0	CCMN (immediate) — 32-bit
0	1	1	0	0	CCMP (immediate) — 32-bit
1	0	1	0	0	CCMN (immediate) — 64-bit
1	1	1	0	0	CCMP (immediate) — 64-bit

Conditional select

These instructions are under Data Processing – Register.

31	30	29	28							21	20 16	15 12	11 10	9 5	4 0
sf	op	S	1	1	0	1	0	1	0	0	Rm	cond	op2	Rn	Rd

sf	op	S	op2	Instruction Details
			1x	UNALLOCATED
		1		UNALLOCATED
0	0	0	00	CSEL — 32-bit
0	0	0	01	CSINC — 32-bit
0	1	0	00	CSINV — 32-bit
0	1	0	01	CSNEG — 32-bit
1	0	0	00	CSEL — 64-bit
1	0	0	01	CSINC — 64-bit
1	1	0	00	CSINV — 64-bit
1	1	0	01	CSNEG — 64-bit

Data-processing (3 source)

These instructions are under Data Processing – Register.

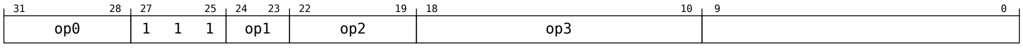
31	30 29	28				24	23 21	20 16	15	14 10	9 5	4 0
sf	op54	1	1	0	1	1	op31	Rm	o0	Ra	Rn	Rd

sf	op54	op31	o0	Instruction Details
	00	010	1	UNALLOCATED
	00	011		UNALLOCATED
	00	100		UNALLOCATED
	00	110	1	UNALLOCATED
	00	111		UNALLOCATED
	01			UNALLOCATED
	1x			UNALLOCATED
0	00	000	0	MADD — 32-bit
0	00	000	1	MSUB — 32-bit
0	00	001	0	UNALLOCATED
0	00	001	1	UNALLOCATED
0	00	010	0	UNALLOCATED
0	00	101	0	UNALLOCATED
0	00	101	1	UNALLOCATED
0	00	110	0	UNALLOCATED
1	00	000	0	MADD — 64-bit
1	00	000	1	MSUB — 64-bit
1	00	001	0	SMADDL
1	00	001	1	SMSUBL

sf	op54	op31	o0	Instruction Details
1	00	010	0	SMULH
1	00	101	0	UMADDL
1	00	101	1	UMSUBL
1	00	110	0	UMULH

Data Processing – Scalar Floating-Point and Advanced SIMD

These instructions are under the top-level.



op0	op1	op2	op3	Instruction details	Architecture version
0000	0x	x101	00xxxxx10	UNALLOCATED	-
0010	0x	x101	00xxxxx10	UNALLOCATED	-
0100	0x	x101	00xxxxx10	Cryptographic AES	-
0101	0x	x0xx	xxx0xxx00	Cryptographic three-register SHA	-
0101	0x	x0xx	xxx0xxx10	UNALLOCATED	-
0101	0x	x101	00xxxxx10	Cryptographic two-register SHA	-
0110	0x	x101	00xxxxx10	UNALLOCATED	-
0111	0x	x0xx	xxx0xxxx0	UNALLOCATED	-
0111	0x	x101	00xxxxx10	UNALLOCATED	-
01x1	00	00xx	xxx0xxxx1	Advanced SIMD scalar copy	-
01x1	01	00xx	xxx0xxxx1	UNALLOCATED	-
01x1	0x	0111	00xxxxx10	UNALLOCATED	-
01x1	0x	10xx	xxx00xxx1	Advanced SIMD scalar three same FP16	-
01x1	0x	10xx	xxx01xxx1	UNALLOCATED	-
01x1	0x	1111	00xxxxx10	Advanced SIMD scalar two-register miscellaneous FP16	-
01x1	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
01x1	0x	x0xx	xxx1xxxx1	Advanced SIMD scalar three same extra	-
01x1	0x	x100	00xxxxx10	Advanced SIMD scalar two-register miscellaneous	-
01x1	0x	x110	00xxxxx10	Advanced SIMD scalar pairwise	-
01x1	0x	x1xx	1xxxxxx10	UNALLOCATED	-
01x1	0x	x1xx	x1xxxxx10	UNALLOCATED	-
01x1	0x	x1xx	xxxxxxx00	Advanced SIMD scalar three different	-
01x1	0x	x1xx	xxxxxxxx1	Advanced SIMD scalar three same	-

op0	op1	op2	op3	Instruction details	Architecture version
01x1	10		xxxxxxxx1	Advanced SIMD scalar shift by immediate	-
01x1	11		xxxxxxxx1	UNALLOCATED	-
01x1	1x		xxxxxxxx0	Advanced SIMD scalar x indexed element	-
0x00	0x	x0xx	xxx0xxx00	Advanced SIMD table lookup	-
0x00	0x	x0xx	xxx0xxx10	Advanced SIMD permute	-
0x10	0x	x0xx	xxx0xxxx0	Advanced SIMD extract	-
0xx0	00	00xx	xxx0xxxx1	Advanced SIMD copy	-
0xx0	01	00xx	xxx0xxxx1	UNALLOCATED	-
0xx0	0x	0111	00xxxxx10	UNALLOCATED	-
0xx0	0x	10xx	xxx00xxx1	Advanced SIMD three same (FP16)	-
0xx0	0x	10xx	xxx01xxx1	UNALLOCATED	-
0xx0	0x	1111	00xxxxx10	Advanced SIMD two-register miscellaneous (FP16)	-
0xx0	0x	x0xx	xxx1xxxx0	UNALLOCATED	-
0xx0	0x	x0xx	xxx1xxxx1	Advanced SIMD three-register extension	-
0xx0	0x	x100	00xxxxx10	Advanced SIMD two-register miscellaneous	-
0xx0	0x	x110	00xxxxx10	Advanced SIMD across lanes	-
0xx0	0x	x1xx	1xxxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	x1xxxxx10	UNALLOCATED	-
0xx0	0x	x1xx	xxxxxxx00	Advanced SIMD three different	-
0xx0	0x	x1xx	xxxxxxxx1	Advanced SIMD three same	-
0xx0	10	0000	xxxxxxxx1	Advanced SIMD modified immediate	-
0xx0	10	!= 0000	xxxxxxxx1	Advanced SIMD shift by immediate	-
0xx0	11		xxxxxxxx1	UNALLOCATED	-
0xx0	1x		xxxxxxxx0	Advanced SIMD vector x indexed element	-
1100	00	10xx	xxx10xxxx	Cryptographic three-register, imm2	-
1100	00	11xx	xxx1x00xx	Cryptographic three-register SHA 512	-
1100	00		xxx0xxxxx	Cryptographic four-register	-
1100	01	00xx		XAR	FEAT_SHA3
1100	01	1000	0001000xx	Cryptographic two-register SHA 512	-
11x1				UNALLOCATED	-
1xx0	1x			UNALLOCATED	-

op0	op1	op2	op3	Instruction details	Architecture version
x0x1	0x	x0xx		Conversion between floating-point and fixed-point	-
x0x1	0x	x1xx	xxx000000	Conversion between floating-point and integer	-
x0x1	0x	x1xx	xxx100000	UNALLOCATED	-
x0x1	0x	x1xx	xxxx10000	Floating-point data-processing (1 source)	-
x0x1	0x	x1xx	xxxxx1000	Floating-point compare	-
x0x1	0x	x1xx	xxxxxx100	Floating-point immediate	-
x0x1	0x	x1xx	xxxxxxx01	Floating-point conditional compare	-
x0x1	0x	x1xx	xxxxxxx10	Floating-point data-processing (2 source)	-
x0x1	0x	x1xx	xxxxxxx11	Floating-point conditional select	-
x0x1	1x			Floating-point data-processing (3 source)	-

Cryptographic AES

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 24	23 22	21 17	16 12	11 10	9 5	4 0
0 1 0 0 1 1 1 0	size	1 0 1 0 0	opcode	1 0	Rn	Rd

size	opcode	Instruction Details
	x1xxx	UNALLOCATED
	000xx	UNALLOCATED
	1xxxx	UNALLOCATED
x1		UNALLOCATED
00	00100	AESE
00	00101	AESD
00	00110	AESMC
00	00111	AESIMC
1x		UNALLOCATED

Cryptographic three-register SHA

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 24	23 22	21	20 16	15	14 12	11 10	9 5	4 0
0 1 0 1 1 1 1 0	size	0	Rm	0	opcode	0 0	Rn	Rd

size	opcode	Instruction Details
	111	UNALLOCATED

size	opcode	Instruction Details
x1		UNALLOCATED
00	000	SHA1C
00	001	SHA1P
00	010	SHA1M
00	011	SHA1SU0
00	100	SHA256H
00	101	SHA256H2
00	110	SHA256SU1
1x		UNALLOCATED

Cryptographic two-register SHA

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 24	23 22	21 17	16 12	11 10	9 5	4 0
0 1 0 1 1 1 1 0	size	1 0 1 0 0	opcode	1 0	Rn	Rd

size	opcode	Instruction Details
	xx1xx	UNALLOCATED
	x1xxx	UNALLOCATED
	1xxxx	UNALLOCATED
x1		UNALLOCATED
00	00000	SHA1H
00	00001	SHA1SU1
00	00010	SHA256SU0
00	00011	UNALLOCATED
1x		UNALLOCATED

Advanced SIMD scalar copy

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 30	29	28 21	20 16	15	14 11	10	9 5	4 0
0 1	op	1 1 1 1 0 0 0 0	imm5	0	imm4	1	Rn	Rd

op	imm4	Instruction Details
0	xxx1	UNALLOCATED
0	xx1x	UNALLOCATED
0	x1xx	UNALLOCATED
0	0000	DUP (element)
0	1xxx	UNALLOCATED

op	imm4	Instruction Details
1		UNALLOCATED

Advanced SIMD scalar three same FP16

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20 16	15	14	13 11	10	9 5	4 0
0	1	U	1	1	1	1	0	a	1	0	Rm	0	0	opcode	1	Rn	Rd

U	a	opcode	Instruction Details	Feature
		110	UNALLOCATED	-
	1	011	UNALLOCATED	-
0	0	011	FMULX	FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	111	FRECPS	FEAT_FP16
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	111	FRSQRTS	FEAT_FP16
1	0	011	UNALLOCATED	-
1	0	100	FCMGE (register)	FEAT_FP16
1	0	101	FACGE	FEAT_FP16
1	0	111	UNALLOCATED	-
1	1	010	FABD	FEAT_FP16
1	1	100	FCMGT (register)	FEAT_FP16
1	1	101	FACGT	FEAT_FP16
1	1	111	UNALLOCATED	-

Advanced SIMD scalar two-register miscellaneous FP16

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22					17	16 12	11	10	9 5	4 0
0	1	U	1	1	1	1	0	a	1	1	1	1	0	0	opcode	1	0	Rn	Rd

U	a	opcode	Instruction Details	Feature
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11110	UNALLOCATED	-

U	a	opcode	Instruction Details	Feature
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	01111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11010	FCVTNS (vector)	FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_FP16
0	1	11101	FRECPE	FEAT_FP16
0	1	11111	FRECPX	FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_FP16
1	1	01110	UNALLOCATED	-
1	1	11010	FCVTPU (vector)	FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_FP16
1	1	11101	FRSQRTE	FEAT_FP16
1	1	11111	UNALLOCATED	-

Advanced SIMD scalar three same extra

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	1	U	1 1 1 1 0	size	0	Rm	1	opcode	1	Rn	Rd

U	opcode	Instruction Details	Feature
	001x	UNALLOCATED	-
	01xx	UNALLOCATED	-
	1xxx	UNALLOCATED	-

U	opcode	Instruction Details	Feature
0	0000	UNALLOCATED	-
0	0001	UNALLOCATED	-
1	0000	SQRDMLAH (vector)	FEAT_RDM
1	0001	SQRDMLSH (vector)	FEAT_RDM

Advanced SIMD scalar two-register miscellaneous

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 30	29	28 24	23 22	21 17	16 12	11 10	9 5	4 0
0 1	U	1 1 1 1 0	size	1 0 0 0 0	opcode	1 0	Rn	Rd

U	size	opcode	Instruction Details
		0000x	UNALLOCATED
		00010	UNALLOCATED
		0010x	UNALLOCATED
		00110	UNALLOCATED
		01111	UNALLOCATED
		1000x	UNALLOCATED
		10011	UNALLOCATED
		10101	UNALLOCATED
		10111	UNALLOCATED
		1100x	UNALLOCATED
		11110	UNALLOCATED
	0x	011xx	UNALLOCATED
	0x	11111	UNALLOCATED
	1x	10110	UNALLOCATED
	1x	11100	UNALLOCATED
0		00011	SUQADD
0		00111	SQABS
0		01000	CMGT (zero)
0		01001	CMEQ (zero)
0		01010	CMLT (zero)
0		01011	ABS
0		10010	UNALLOCATED
0		10100	SQXTN, SQXTN2
0	0x	10110	UNALLOCATED
0	0x	11010	FCVTNS (vector)

U	size	opcode	Instruction Details
0	0x	11011	FCVTMS (vector)
0	0x	11100	FCVTAS (vector)
0	0x	11101	SCVTF (vector, integer)
0	1x	01100	FCMGT (zero)
0	1x	01101	FCMEQ (zero)
0	1x	01110	FCMLT (zero)
0	1x	11010	FCVTPS (vector)
0	1x	11011	FCVTZS (vector, integer)
0	1x	11101	FRECPE
0	1x	11111	FRECPX
1		00011	USQADD
1		00111	SQNEG
1		01000	CMGE (zero)
1		01001	CMLE (zero)
1		01010	UNALLOCATED
1		01011	NEG (vector)
1		10010	SQXTUN, SQXTUN2
1		10100	UQXTN, UQXTN2
1	0x	10110	FCVTXN, FCVTXN2
1	0x	11010	FCVTNU (vector)
1	0x	11011	FCVTMU (vector)
1	0x	11100	FCVTAU (vector)
1	0x	11101	UCVTF (vector, integer)
1	1x	01100	FCMGE (zero)
1	1x	01101	FCMLE (zero)
1	1x	01110	UNALLOCATED
1	1x	11010	FCVTPU (vector)
1	1x	11011	FCVTZU (vector, integer)
1	1x	11101	FRSQRTE
1	1x	11111	UNALLOCATED

Advanced SIMD scalar pairwise

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21 17	16 12	11	10	9 5	4 0
0	1	U	1 1 1 1 0	size	1 1 0 0 0	opcode	1	0	Rn	Rd

U	size	opcode	Instruction Details	Feature
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		11010	UNALLOCATED	-
		111xx	UNALLOCATED	-
	1x	01101	UNALLOCATED	-
0		11011	ADDP (scalar)	-
0	00	01100	FMAXNMP (scalar) — half-precision	FEAT_FP16
0	00	01101	FADDP (scalar) — half-precision	FEAT_FP16
0	00	01111	FMAXP (scalar) — half-precision	FEAT_FP16
0	01	01100	UNALLOCATED	-
0	01	01101	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	FMINNMP (scalar) — half-precision	FEAT_FP16
0	10	01111	FMINP (scalar) — half-precision	FEAT_FP16
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		11011	UNALLOCATED	-
1	0x	01100	FMAXNMP (scalar) — single-precision and double-precision	-
1	0x	01101	FADDP (scalar) — single-precision and double-precision	-
1	0x	01111	FMAXP (scalar) — single-precision and double-precision	-
1	1x	01100	FMINNMP (scalar) — single-precision and double-precision	-
1	1x	01111	FMINP (scalar) — single-precision and double-precision	-

Advanced SIMD scalar three different

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 30	29	28 24	23 22	21	20 16	15 12	11 10	9 5	4 0
0 1	U	1 1 1 1 0	size	1	Rm	opcode	0 0	Rn	Rd

U	opcode	Instruction Details
	00xx	UNALLOCATED
	01xx	UNALLOCATED
	1000	UNALLOCATED
	1010	UNALLOCATED
	1100	UNALLOCATED
	111x	UNALLOCATED
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED

Advanced SIMD scalar three same

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 16	15 11	10	9 5	4 0
0	1	U	1 1 1 1 0	size	1	Rm	opcode	1	Rn	Rd

U	size	opcode	Instruction Details
		00000	UNALLOCATED
		0001x	UNALLOCATED
		00100	UNALLOCATED
		011xx	UNALLOCATED
		1001x	UNALLOCATED
	1x	11011	UNALLOCATED
0		00001	SQADD
0		00101	SQSUB
0		00110	CMGT (register)
0		00111	CMGE (register)
0		01000	SSHL
0		01001	SQSHL (register)
0		01010	SRSHL
0		01011	SQRSHL
0		10000	ADD (vector)
0		10001	CMTST
0		10100	UNALLOCATED

U	size	opcode	Instruction Details
0		10101	UNALLOCATED
0		10110	SQDMULH (vector)
0		10111	UNALLOCATED
0	0x	11000	UNALLOCATED
0	0x	11001	UNALLOCATED
0	0x	11010	UNALLOCATED
0	0x	11011	FMULX
0	0x	11100	FCMEQ (register)
0	0x	11101	UNALLOCATED
0	0x	11110	UNALLOCATED
0	0x	11111	FRECPS
0	1x	11000	UNALLOCATED
0	1x	11001	UNALLOCATED
0	1x	11010	UNALLOCATED
0	1x	11100	UNALLOCATED
0	1x	11101	UNALLOCATED
0	1x	11110	UNALLOCATED
0	1x	11111	FRSQRTS
1		00001	UQADD
1		00101	UQSUB
1		00110	CMHI (register)
1		00111	CMHS (register)
1		01000	USHL
1		01001	UQSHL (register)
1		01010	URSHL
1		01011	UQRSHL
1		10000	SUB (vector)
1		10001	CMEQ (register)
1		10100	UNALLOCATED
1		10101	UNALLOCATED
1		10110	SQRDMULH (vector)
1		10111	UNALLOCATED
1	0x	11000	UNALLOCATED
1	0x	11001	UNALLOCATED

U	size	opcode	Instruction Details
1	0x	11010	UNALLOCATED
1	0x	11011	UNALLOCATED
1	0x	11100	FCMGE (register)
1	0x	11101	FACGE
1	0x	11110	UNALLOCATED
1	0x	11111	UNALLOCATED
1	1x	11000	UNALLOCATED
1	1x	11001	UNALLOCATED
1	1x	11010	FABD
1	1x	11100	FCMGT (register)
1	1x	11101	FACGT
1	1x	11110	UNALLOCATED
1	1x	11111	UNALLOCATED

Advanced SIMD scalar shift by immediate

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28					23	22 19	18 16	15 11	10	9 5	4 0
0	1	U	1	1	1	1	1	0	immh	immb	opcode	1	Rn	Rd

U	immh	opcode	Instruction Details
	!= 0000	00001	UNALLOCATED
	!= 0000	00011	UNALLOCATED
	!= 0000	00101	UNALLOCATED
	!= 0000	00111	UNALLOCATED
	!= 0000	01001	UNALLOCATED
	!= 0000	01011	UNALLOCATED
	!= 0000	01101	UNALLOCATED
	!= 0000	01111	UNALLOCATED
	!= 0000	101xx	UNALLOCATED
	!= 0000	110xx	UNALLOCATED
	!= 0000	11101	UNALLOCATED
	!= 0000	11110	UNALLOCATED
	0000		UNALLOCATED
0	!= 0000	00000	SSHR
0	!= 0000	00010	SSRA
0	!= 0000	00100	SRSHR

U	immh	opcode	Instruction Details
0	!= 0000	00110	SRSRA
0	!= 0000	01000	UNALLOCATED
0	!= 0000	01010	SHL
0	!= 0000	01100	UNALLOCATED
0	!= 0000	01110	SQSHL (immediate)
0	!= 0000	10000	UNALLOCATED
0	!= 0000	10001	UNALLOCATED
0	!= 0000	10010	SQSHRN, SQSHRN2
0	!= 0000	10011	SQRSHRN, SQRSHRN2
0	!= 0000	11100	SCVTF (vector, fixed-point)
0	!= 0000	11111	FCVTZS (vector, fixed-point)
1	!= 0000	00000	USHR
1	!= 0000	00010	USRA
1	!= 0000	00100	URSHR
1	!= 0000	00110	URSRA
1	!= 0000	01000	SRI
1	!= 0000	01010	SLI
1	!= 0000	01100	SQSHLU
1	!= 0000	01110	UQSHL (immediate)
1	!= 0000	10000	SQSHRUN, SQSHRUN2
1	!= 0000	10001	SQRSHRUN, SQRSHRUN2
1	!= 0000	10010	UQSHRN, UQSHRN2
1	!= 0000	10011	UQRSHRN, UQRSHRN2
1	!= 0000	11100	UCVTF (vector, fixed-point)
1	!= 0000	11111	FCVTZU (vector, fixed-point)

Advanced SIMD scalar x indexed element

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	1	U	1	1	1	1	1	size	L	M	Rm	opcode	H	0	Rn	Rd

U	size	opcode	Instruction Details	Feature
		0000	UNALLOCATED	-
		0010	UNALLOCATED	-
		0100	UNALLOCATED	-
		0110	UNALLOCATED	-

U	size	opcode	Instruction Details	Feature
		1000	UNALLOCATED	-
		1010	UNALLOCATED	-
		1110	UNALLOCATED	-
	01	0001	UNALLOCATED	-
	01	0101	UNALLOCATED	-
	01	1001	UNALLOCATED	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	-
0		0111	SQDMLSL, SQDMLSL2 (by element)	-
0		1011	SQDMULL, SQDMULL2 (by element)	-
0		1100	SQDMULH (by element)	-
0		1101	SQRDMULH (by element)	-
0		1111	UNALLOCATED	-
0	00	0001	FMLA (by element) — half-precision	FEAT_FP16
0	00	0101	FMLS (by element) — half-precision	FEAT_FP16
0	00	1001	FMUL (by element) — half-precision	FEAT_FP16
0	1x	0001	FMLA (by element) — single-precision and double-precision	-
0	1x	0101	FMLS (by element) — single-precision and double-precision	-
0	1x	1001	FMUL (by element) — single-precision and double-precision	-
1		0011	UNALLOCATED	-
1		0111	UNALLOCATED	-
1		1011	UNALLOCATED	-
1		1100	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	00	0001	UNALLOCATED	-
1	00	0101	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	FEAT_FP16
1	1x	0001	UNALLOCATED	-
1	1x	0101	UNALLOCATED	-

U	size	opcode	Instruction Details	Feature
1	1x	1001	FMULX (by element) — single-precision and double-precision	-

Advanced SIMD table lookup

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23 22	21	20 16	15	14 13	12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	op2	0	Rm	0	len	op	0	0	Rn	Rd

op2	len	op	Instruction Details
x1			UNALLOCATED
00	00	0	TBL — single register table
00	00	1	TBX — single register table
00	01	0	TBL — two register table
00	01	1	TBX — two register table
00	10	0	TBL — three register table
00	10	1	TBX — three register table
00	11	0	TBL — four register table
00	11	1	TBX — four register table
1x			UNALLOCATED

Advanced SIMD permute

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23 22	21	20 16	15	14 12	11	10	9 5	4 0
0	Q	0	0	1	1	1	0	size	0	Rm	0	opcode	1	0	Rn	Rd

opcode	Instruction Details
000	UNALLOCATED
001	UZP1
010	TRN1
011	ZIP1
100	UNALLOCATED
101	UZP2
110	TRN2
111	ZIP2

Advanced SIMD extract

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29					24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	1	0	1	1	1	0	op2	0	Rm	0	imm4	0	Rn	Rd

op2	Instruction Details
x1	UNALLOCATED
00	EXT
1x	UNALLOCATED

Advanced SIMD copy

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28							21	20 16	15	14 11	10	9 5	4 0
0	Q	op	0	1	1	1	0	0	0	0	imm5	0	imm4	1	Rn	Rd

Q	op	imm5	imm4	Instruction Details
		x0000		UNALLOCATED
	0		0000	DUP (element)
	0		0001	DUP (general)
	0		0010	UNALLOCATED
	0		0100	UNALLOCATED
	0		0110	UNALLOCATED
	0		1xxx	UNALLOCATED
0	0		0011	UNALLOCATED
0	0		0101	SMOV
0	0		0111	UMOV
0	1			UNALLOCATED
1	0		0011	INS (general)
1	0		0101	SMOV
1	0	x1000	0111	UMOV
1	1			INS (element)

Advanced SIMD three same (FP16)

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23	22	21	20 16	15	14	13 11	10	9 5	4 0
0	Q	U	0	1	1	1	0	a	1	0	Rm	0	0	opcode	1	Rn	Rd

U	a	opcode	Instruction Details	Feature
0	0	000	FMAXNM (vector)	FEAT_FP16
0	0	001	FMLA (vector)	FEAT_FP16
0	0	010	FADD (vector)	FEAT_FP16

U	a	opcode	Instruction Details	Feature
0	0	011	FMULX	FEAT_FP16
0	0	100	FCMEQ (register)	FEAT_FP16
0	0	101	UNALLOCATED	-
0	0	110	FMAX (vector)	FEAT_FP16
0	0	111	FRECPS	FEAT_FP16
0	1	000	FMINNM (vector)	FEAT_FP16
0	1	001	FMLS (vector)	FEAT_FP16
0	1	010	FSUB (vector)	FEAT_FP16
0	1	011	UNALLOCATED	-
0	1	100	UNALLOCATED	-
0	1	101	UNALLOCATED	-
0	1	110	FMIN (vector)	FEAT_FP16
0	1	111	FRSQRTS	FEAT_FP16
1	0	000	FMAXNMP (vector)	FEAT_FP16
1	0	001	UNALLOCATED	-
1	0	010	FADDP (vector)	FEAT_FP16
1	0	011	FMUL (vector)	FEAT_FP16
1	0	100	FCMGE (register)	FEAT_FP16
1	0	101	FACGE	FEAT_FP16
1	0	110	FMAXP (vector)	FEAT_FP16
1	0	111	FDIV (vector)	FEAT_FP16
1	1	000	FMINNMP (vector)	FEAT_FP16
1	1	001	UNALLOCATED	-
1	1	010	FABD	FEAT_FP16
1	1	011	UNALLOCATED	-
1	1	100	FCMGT (register)	FEAT_FP16
1	1	101	FACGT	FEAT_FP16
1	1	110	FMINP (vector)	FEAT_FP16
1	1	111	UNALLOCATED	-

Advanced SIMD two-register miscellaneous (FP16)

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23	22 17	16 12	11	10	9 5	4 0
0	Q	U	0 1 1 1 0	a	1 1 1 1 0 0	opcode	1	0	Rn	Rd

U	a	opcode	Instruction Details	Feature
		00xxx	UNALLOCATED	-
		010xx	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		11110	UNALLOCATED	-
	0	011xx	UNALLOCATED	-
	0	11111	UNALLOCATED	-
	1	11100	UNALLOCATED	-
0	0	11000	FRINTN (vector)	FEAT_FP16
0	0	11001	FRINTM (vector)	FEAT_FP16
0	0	11010	FCVTNS (vector)	FEAT_FP16
0	0	11011	FCVTMS (vector)	FEAT_FP16
0	0	11100	FCVTAS (vector)	FEAT_FP16
0	0	11101	SCVTF (vector, integer)	FEAT_FP16
0	1	01100	FCMGT (zero)	FEAT_FP16
0	1	01101	FCMEQ (zero)	FEAT_FP16
0	1	01110	FCMLT (zero)	FEAT_FP16
0	1	01111	FABS (vector)	FEAT_FP16
0	1	11000	FRINTP (vector)	FEAT_FP16
0	1	11001	FRINTZ (vector)	FEAT_FP16
0	1	11010	FCVTPS (vector)	FEAT_FP16
0	1	11011	FCVTZS (vector, integer)	FEAT_FP16
0	1	11101	FRECPE	FEAT_FP16
0	1	11111	UNALLOCATED	-
1	0	11000	FRINTA (vector)	FEAT_FP16
1	0	11001	FRINTX (vector)	FEAT_FP16
1	0	11010	FCVTNU (vector)	FEAT_FP16
1	0	11011	FCVTMU (vector)	FEAT_FP16
1	0	11100	FCVTAU (vector)	FEAT_FP16
1	0	11101	UCVTF (vector, integer)	FEAT_FP16
1	1	01100	FCMGE (zero)	FEAT_FP16
1	1	01101	FCMLE (zero)	FEAT_FP16
1	1	01110	UNALLOCATED	-
1	1	01111	FNEG (vector)	FEAT_FP16
1	1	11000	UNALLOCATED	-

U	a	opcode	Instruction Details	Feature
1	1	11001	FRINTI (vector)	FEAT_FP16
1	1	11010	FCVTPU (vector)	FEAT_FP16
1	1	11011	FCVTZU (vector, integer)	FEAT_FP16
1	1	11101	FRSQRTE	FEAT_FP16
1	1	11111	FSQRT (vector)	FEAT_FP16

Advanced SIMD three-register extension

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 16	15	14 11	10	9 5	4 0
0	Q	U	0	1	1	1	0	size	0	Rm	1	opcode	1	Rn	Rd

Q	U	size	opcode	Instruction Details	Feature
		0x	0011	UNALLOCATED	-
		11	0011	UNALLOCATED	-
	0		0000	UNALLOCATED	-
	0		0001	UNALLOCATED	-
	0		0010	SDOT (vector)	FEAT_DotProd
	0		1xxx	UNALLOCATED	-
	1		0000	SQRDMLAH (vector)	FEAT_RDM
	1		0001	SQRDMLSH (vector)	FEAT_RDM
	1		0010	UDOT (vector)	FEAT_DotProd
	1	00	1101	UNALLOCATED	-
	1	00	1111	UNALLOCATED	-
	1	1x	1101	UNALLOCATED	-
	1	10	0011	UNALLOCATED	-
	1	10	1111	UNALLOCATED	-
0			01xx	UNALLOCATED	-
0	1	01	1101	UNALLOCATED	-
1		0x	01xx	UNALLOCATED	-
1		1x	011x	UNALLOCATED	-
1	1	10	0101	UNALLOCATED	-

Advanced SIMD two-register miscellaneous

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21				17	16 12	11	10	9 5	4 0
0	Q	U	0	1	1	1	0	size	1	0	0	0	0	opcode	1	0	Rn	Rd

U	size	opcode	Instruction Details
		1000x	UNALLOCATED
		10101	UNALLOCATED
	0x	011xx	UNALLOCATED
	1x	10111	UNALLOCATED
	1x	11110	UNALLOCATED
	11	10110	UNALLOCATED
0		00000	REV64
0		00001	REV16 (vector)
0		00010	SADDLP
0		00011	SUQADD
0		00100	CLS (vector)
0		00101	CNT
0		00110	SADALP
0		00111	SQABS
0		01000	CMGT (zero)
0		01001	CMEQ (zero)
0		01010	CMLT (zero)
0		01011	ABS
0		10010	XTN, XTN2
0		10011	UNALLOCATED
0		10100	SQXTN, SQXTN2
0	0x	10110	FCVTN, FCVTN2
0	0x	10111	FCVTL, FCVTL2
0	0x	11000	FRINTN (vector)
0	0x	11001	FRINTM (vector)
0	0x	11010	FCVTNS (vector)
0	0x	11011	FCVTMS (vector)
0	0x	11100	FCVTAS (vector)
0	0x	11101	SCVTF (vector, integer)
0	1x	01100	FCMGT (zero)
0	1x	01101	FCMEQ (zero)
0	1x	01110	FCMLT (zero)
0	1x	01111	FABS (vector)
0	1x	11000	FRINTP (vector)

U	size	opcode	Instruction Details
0	1x	11001	FRINTZ (vector)
0	1x	11010	FCVTPS (vector)
0	1x	11011	FCVTZS (vector, integer)
0	1x	11100	URECPE
0	1x	11101	FRECPE
0	1x	11111	UNALLOCATED
1		00000	REV32 (vector)
1		00001	UNALLOCATED
1		00010	UADDLP
1		00011	USQADD
1		00100	CLZ (vector)
1		00110	UADALP
1		00111	SQNEG
1		01000	CMGE (zero)
1		01001	CMLE (zero)
1		01010	UNALLOCATED
1		01011	NEG (vector)
1		10010	SQXTUN, SQXTUN2
1		10011	SHLL, SHLL2
1		10100	UQXTN, UQXTN2
1	0x	10110	FCVTXN, FCVTXN2
1	0x	10111	UNALLOCATED
1	0x	11000	FRINTA (vector)
1	0x	11001	FRINTX (vector)
1	0x	11010	FCVTNU (vector)
1	0x	11011	FCVTMU (vector)
1	0x	11100	FCVTAU (vector)
1	0x	11101	UCVTF (vector, integer)
1	00	00101	NOT
1	01	00101	RBIT (vector)
1	1x	00101	UNALLOCATED
1	1x	01100	FCMGE (zero)
1	1x	01101	FCMLE (zero)
1	1x	01110	UNALLOCATED

U	size	opcode	Instruction Details
1	1x	01111	FNEG (vector)
1	1x	11000	UNALLOCATED
1	1x	11001	FRINTI (vector)
1	1x	11010	FCVTPU (vector)
1	1x	11011	FCVTZU (vector, integer)
1	1x	11100	URSQRTE
1	1x	11101	FRSQRTE
1	1x	11111	FSQRT (vector)
1	10	10110	UNALLOCATED

Advanced SIMD across lanes

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21 17	16 12	11 10	9 5	4 0
0	Q	U	0 1 1 1 0	size	1 1 0 0 0	opcode	1 0	Rn	Rd

U	size	opcode	Instruction Details	Feature
		0000x	UNALLOCATED	-
		00010	UNALLOCATED	-
		001xx	UNALLOCATED	-
		0100x	UNALLOCATED	-
		01011	UNALLOCATED	-
		01101	UNALLOCATED	-
		01110	UNALLOCATED	-
		10xxx	UNALLOCATED	-
		1100x	UNALLOCATED	-
		111xx	UNALLOCATED	-
0		00011	SADDLV	-
0		01010	SMAXV	-
0		11010	SMINV	-
0		11011	ADDV	-
0	00	01100	FMAXNMV — half-precision	FEAT_FP16
0	00	01111	FMAXV — half-precision	FEAT_FP16
0	01	01100	UNALLOCATED	-
0	01	01111	UNALLOCATED	-
0	10	01100	FMINNMV — half-precision	FEAT_FP16
0	10	01111	FMINV — half-precision	FEAT_FP16

U	size	opcode	Instruction Details	Feature
0	11	01100	UNALLOCATED	-
0	11	01111	UNALLOCATED	-
1		00011	UADDLV	-
1		01010	UMAXV	-
1		11010	UMINV	-
1		11011	UNALLOCATED	-
1	0x	01100	FMAXNMV — single-precision and double-precision	-
1	0x	01111	FMAXV — single-precision and double-precision	-
1	1x	01100	FMINNMV — single-precision and double-precision	-
1	1x	01111	FMINV — single-precision and double-precision	-

Advanced SIMD three different

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 16	15 12	11	10	9 5	4 0
0	Q	U	0	1	1	1	0	size	1	Rm	opcode	0	0	Rn	Rd

U	opcode	Instruction Details
	1111	UNALLOCATED
0	0000	SADDL, SADDL2
0	0001	SADDW, SADDW2
0	0010	SSUBL, SSUBL2
0	0011	SSUBW, SSUBW2
0	0100	ADDHN, ADDHN2
0	0101	SABAL, SABAL2
0	0110	SUBHN, SUBHN2
0	0111	SABDL, SABDL2
0	1000	SMLAL, SMLAL2 (vector)
0	1001	SQDMLAL, SQDMLAL2 (vector)
0	1010	SMLSL, SMLSL2 (vector)
0	1011	SQDMLSL, SQDMLSL2 (vector)
0	1100	SMULL, SMULL2 (vector)
0	1101	SQDMULL, SQDMULL2 (vector)
0	1110	PMULL, PMULL2

U	opcode	Instruction Details
1	0000	UADDL, UADDL2
1	0001	UADDW, UADDW2
1	0010	USUBL, USUBL2
1	0011	USUBW, USUBW2
1	0100	RADDHN, RADDHN2
1	0101	UABAL, UABAL2
1	0110	RSUBHN, RSUBHN2
1	0111	UABDL, UABDL2
1	1000	UMLAL, UMLAL2 (vector)
1	1001	UNALLOCATED
1	1010	UMLSL, UMLSL2 (vector)
1	1011	UNALLOCATED
1	1100	UMULL, UMULL2 (vector)
1	1101	UNALLOCATED
1	1110	UNALLOCATED

Advanced SIMD three same

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 16	15 11	10	9 5	4 0
0	Q	U	0 1 1 1 0	size	1	Rm	opcode	1	Rn	Rd

U	size	opcode	Instruction Details	Feature
0		00000	SHADD	-
0		00001	SQADD	-
0		00010	SRHADD	-
0		00100	SHSUB	-
0		00101	SQSUB	-
0		00110	CMGT (register)	-
0		00111	CMGE (register)	-
0		01000	SSHL	-
0		01001	SQSHL (register)	-
0		01010	SRSHL	-
0		01011	SQRSHL	-
0		01100	SMAX	-
0		01101	SMIN	-
0		01110	SABD	-

U	size	opcode	Instruction Details	Feature
0		01111	SABA	-
0		10000	ADD (vector)	-
0		10001	CMTST	-
0		10010	MLA (vector)	-
0		10011	MUL (vector)	-
0		10100	SMAXP	-
0		10101	SMINP	-
0		10110	SQDMULH (vector)	-
0		10111	ADDP (vector)	-
0	0x	11000	FMAXNM (vector)	-
0	0x	11001	FMLA (vector)	-
0	0x	11010	FADD (vector)	-
0	0x	11011	FMULX	-
0	0x	11100	FCMEQ (register)	-
0	0x	11110	FMAX (vector)	-
0	0x	11111	FRECPS	-
0	00	00011	AND (vector)	-
0	00	11101	FMLAL, FMLAL2 (vector) — FMLAL	FEAT_FHM
0	01	00011	BIC (vector, register)	-
0	01	11101	UNALLOCATED	-
0	1x	11000	FMINNM (vector)	-
0	1x	11001	FMLS (vector)	-
0	1x	11010	FSUB (vector)	-
0	1x	11011	UNALLOCATED	-
0	1x	11100	UNALLOCATED	-
0	1x	11110	FMIN (vector)	-
0	1x	11111	FRSQRTS	-
0	10	00011	ORR (vector, register)	-
0	10	11101	FMLSL, FMLSL2 (vector) — FMLSL	FEAT_FHM
0	11	00011	ORN (vector)	-
0	11	11101	UNALLOCATED	-
1		00000	UHADD	-
1		00001	UQADD	-

U	size	opcode	Instruction Details	Feature
1		00010	URHADD	-
1		00100	UHSUB	-
1		00101	UQSUB	-
1		00110	CMHI (register)	-
1		00111	CMHS (register)	-
1		01000	USHL	-
1		01001	UQSHL (register)	-
1		01010	URSHL	-
1		01011	UQRSHL	-
1		01100	UMAX	-
1		01101	UMIN	-
1		01110	UABD	-
1		01111	UABA	-
1		10000	SUB (vector)	-
1		10001	CMEQ (register)	-
1		10010	MLS (vector)	-
1		10011	PMUL	-
1		10100	UMAXP	-
1		10101	UMINP	-
1		10110	SQRDMULH (vector)	-
1		10111	UNALLOCATED	-
1	0x	11000	FMAXNMP (vector)	-
1	0x	11010	FADDP (vector)	-
1	0x	11011	FMUL (vector)	-
1	0x	11100	FCMGE (register)	-
1	0x	11101	FACGE	-
1	0x	11110	FMAXP (vector)	-
1	0x	11111	FDIV (vector)	-
1	00	00011	EOR (vector)	-
1	00	11001	FMLAL, FMLAL2 (vector) — FMLAL2	FEAT_FHM
1	01	00011	BSL	-
1	01	11001	UNALLOCATED	-
1	1x	11000	FMINNMP (vector)	-
1	1x	11010	FABD	-

U	size	opcode	Instruction Details	Feature
1	1x	11011	UNALLOCATED	-
1	1x	11100	FCMGT (register)	-
1	1x	11101	FACGT	-
1	1x	11110	FMINP (vector)	-
1	1x	11111	UNALLOCATED	-
1	10	00011	BIT	-
1	10	11001	FMLSL, FMLSL2 (vector) — FMLSL2	FEAT_FHM
1	11	00011	BIF	-
1	11	11001	UNALLOCATED	-

Advanced SIMD modified immediate

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28									19	18	17	16	15 12	11	10	9	8	7	6	5	4 0
0	Q	op	0	1	1	1	1	0	0	0	0	0	a	b	c	cmode	o2	1	d	e	f	g	h	Rd

Q	op	cmode	o2	Instruction Details	Feature
	0	0xxx	1	UNALLOCATED	-
	0	0xx0	0	MOVI — 32-bit shifted immediate	-
	0	0xx1	0	ORR (vector, immediate) — 32-bit	-
	0	10xx	1	UNALLOCATED	-
	0	10x0	0	MOVI — 16-bit shifted immediate	-
	0	10x1	0	ORR (vector, immediate) — 16-bit	-
	0	110x	0	MOVI — 32-bit shifting ones	-
	0	110x	1	UNALLOCATED	-
	0	1110	0	MOVI — 8-bit	-
	0	1110	1	UNALLOCATED	-
	0	1111	0	FMOV (vector, immediate) — single-precision	-
	0	1111	1	FMOV (vector, immediate) — half-precision	FEAT_FP16
	1		1	UNALLOCATED	-
	1	0xx0	0	MVNI — 32-bit shifted immediate	-
	1	0xx1	0	BIC (vector, immediate) — 32-bit	-
	1	10x0	0	MVNI — 16-bit shifted immediate	-
	1	10x1	0	BIC (vector, immediate) — 16-bit	-
	1	110x	0	MVNI — 32-bit shifting ones	-

Q	op	cmode	o2	Instruction Details	Feature
0	1	1110	0	MOVI — 64-bit scalar	-
0	1	1111	0	UNALLOCATED	-
1	1	1110	0	MOVI — 64-bit vector	-
1	1	1111	0	FMOV (vector, immediate) — double-precision	-

Advanced SIMD shift by immediate

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 23	22 19	18 16	15 11	10	9 5	4 0
0	Q	U	0 1 1 1 1 0	!= 0000	immb	opcode	1	Rn	Rd
				immh					

The following constraints also apply to this encoding: immh != 0000 && immh != 0000

U	opcode	Instruction Details
	00001	UNALLOCATED
	00011	UNALLOCATED
	00101	UNALLOCATED
	00111	UNALLOCATED
	01001	UNALLOCATED
	01011	UNALLOCATED
	01101	UNALLOCATED
	01111	UNALLOCATED
	10101	UNALLOCATED
	1011x	UNALLOCATED
	110xx	UNALLOCATED
	11101	UNALLOCATED
	11110	UNALLOCATED
0	00000	SSHR
0	00010	SSRA
0	00100	SRSHR
0	00110	SRSRA
0	01000	UNALLOCATED
0	01010	SHL
0	01100	UNALLOCATED
0	01110	SQSHL (immediate)
0	10000	SHRN, SHRN2
0	10001	RSHRN, RSHRN2

U	opcode	Instruction Details
0	10010	SQSHRN, SQSHRN2
0	10011	SQRSHRN, SQRSHRN2
0	10100	SSHLL, SSHLL2
0	11100	SCVTF (vector, fixed-point)
0	11111	FCVTZS (vector, fixed-point)
1	00000	USHR
1	00010	USRA
1	00100	URSHR
1	00110	URSRA
1	01000	SRI
1	01010	SLI
1	01100	SQSHLU
1	01110	UQSHL (immediate)
1	10000	SQSHRUN, SQSHRUN2
1	10001	SQRSHRUN, SQRSHRUN2
1	10010	UQSHRN, UQSHRN2
1	10011	UQRSHRN, UQRSHRN2
1	10100	USHLL, USHLL2
1	11100	UCVTF (vector, fixed-point)
1	11111	FCVTZU (vector, fixed-point)

Advanced SIMD vector x indexed element

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20	19 16	15 12	11	10	9 5	4 0
0	Q	U	0 1 1 1 1	size	L	M	Rm	opcode	H	0	Rn	Rd

U	size	opcode	Instruction Details	Feature
	01	1001	UNALLOCATED	-
0		0010	SMLAL, SMLAL2 (by element)	-
0		0011	SQDMLAL, SQDMLAL2 (by element)	-
0		0110	SMLSL, SMLSL2 (by element)	-
0		0111	SQDMLSL, SQDMLSL2 (by element)	-
0		1000	MUL (by element)	-
0		1010	SMULL, SMULL2 (by element)	-

U	size	opcode	Instruction Details	Feature
0		1011	SQDMULL, SQDMULL2 (by element)	-
0		1100	SQDMULH (by element)	-
0		1101	SQRDMULH (by element)	-
0		1110	SDOT (by element)	FEAT_DotProd
0	0x	0000	UNALLOCATED	-
0	0x	0100	UNALLOCATED	-
0	00	0001	FMLA (by element) — half-precision	FEAT_FP16
0	00	0101	FMLS (by element) — half-precision	FEAT_FP16
0	00	1001	FMUL (by element) — half-precision	FEAT_FP16
0	01	0001	UNALLOCATED	-
0	01	0101	UNALLOCATED	-
0	1x	0001	FMLA (by element) — single-precision and double-precision	-
0	1x	0101	FMLS (by element) — single-precision and double-precision	-
0	1x	1001	FMUL (by element) — single-precision and double-precision	-
0	10	0000	FMLAL, FMLAL2 (by element) — FMLAL	FEAT_FHM
0	10	0100	FMLSL, FMLSL2 (by element) — FMLSL	FEAT_FHM
0	11	0000	UNALLOCATED	-
0	11	0100	UNALLOCATED	-
1		0000	MLA (by element)	-
1		0010	UMLAL, UMLAL2 (by element)	-
1		0100	MLS (by element)	-
1		0110	UMLSL, UMLSL2 (by element)	-
1		1010	UMULL, UMULL2 (by element)	-
1		1011	UNALLOCATED	-
1		1101	SQRDMLAH (by element)	FEAT_RDM
1		1110	UDOT (by element)	FEAT_DotProd
1		1111	SQRDMLSH (by element)	FEAT_RDM
1	0x	1000	UNALLOCATED	-
1	0x	1100	UNALLOCATED	-
1	00	0001	UNALLOCATED	-
1	00	0011	UNALLOCATED	-

U	size	opcode	Instruction Details	Feature
1	00	0101	UNALLOCATED	-
1	00	0111	UNALLOCATED	-
1	00	1001	FMULX (by element) — half-precision	FEAT_FP16
1	1x	1001	FMULX (by element) — single-precision and double-precision	-
1	10	1000	FMLAL, FMLAL2 (by element) — FMLAL2	FEAT_FHM
1	10	1100	FMLSL, FMLSL2 (by element) — FMLSL2	FEAT_FHM
1	11	0001	UNALLOCATED	-
1	11	0011	UNALLOCATED	-
1	11	0101	UNALLOCATED	-
1	11	0111	UNALLOCATED	-
1	11	1000	UNALLOCATED	-
1	11	1100	UNALLOCATED	-

Cryptographic three-register, imm2

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 21	20 16	15	14	13 12	11 10	9 5	4 0
1 1 0 0 1 1 1 0 0 1 0	Rm	1	0	imm2	opcode	Rn	Rd

opcode	Instruction Details	Feature
00	SM3TT1A	FEAT_SM3
01	SM3TT1B	FEAT_SM3
10	SM3TT2A	FEAT_SM3
11	SM3TT2B	FEAT_SM3

Cryptographic three-register SHA 512

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31 21	20 16	15	14	13	12	11 10	9 5	4 0
1 1 0 0 1 1 1 0 0 1 1	Rm	1	O	0	0	opcode	Rn	Rd

O	opcode	Instruction Details	Feature
0	00	SHA512H	FEAT_SHA512
0	01	SHA512H2	FEAT_SHA512
0	10	SHA512SU1	FEAT_SHA512
0	11	RAX1	FEAT_SHA3
1	00	SM3PARTW1	FEAT_SM3

O	opcode	Instruction Details	Feature
1	01	SM3PARTW2	FEAT_SM3
1	10	SM4EKEY	FEAT_SM4
1	11	UNALLOCATED	-

Cryptographic four-register

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31								23	22 21	20 16	15	14 10	9 5	4 0
1	1	0	0	1	1	1	0	0	Op0	Rm	0	Ra	Rn	Rd

Op0	Instruction Details	Feature
00	EOR3	FEAT_SHA3
01	BCAX	FEAT_SHA3
10	SM3SS1	FEAT_SM3
11	UNALLOCATED	-

Cryptographic two-register SHA 512

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31																			12	11 10	9 5	4 0
1	1	0	0	1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	0	opcode	Rn	Rd

opcode	Instruction Details	Feature
00	SHA512SU0	FEAT_SHA512
01	SM4E	FEAT_SM4
1x	UNALLOCATED	-

Conversion between floating-point and fixed-point

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 19	18 16	15 10	9 5	4 0
sf	0	S	1	1	1	1	0	ptype	0	rmode	opcode	scale	Rn	Rd

sf	S	ptype	rmode	opcode	scale	Instruction Details	Feature
				1xx		UNALLOCATED	-
			x0	00x		UNALLOCATED	-
			x1	01x		UNALLOCATED	-
			0x	00x		UNALLOCATED	-
			1x	01x		UNALLOCATED	-
		10				UNALLOCATED	-
	1					UNALLOCATED	-

sf	S	ptype	rmode	opcode	scale	Instruction Details	Feature
0					0xxxxx	UNALLOCATED	-
0	0	00	00	010		SCVTF (scalar, fixed-point) — 32-bit to single-precision	-
0	0	00	00	011		UCVTF (scalar, fixed-point) — 32-bit to single-precision	-
0	0	00	11	000		FCVTZS (scalar, fixed-point) — single-precision to 32-bit	-
0	0	00	11	001		FCVTZU (scalar, fixed-point) — single-precision to 32-bit	-
0	0	01	00	010		SCVTF (scalar, fixed-point) — 32-bit to double-precision	-
0	0	01	00	011		UCVTF (scalar, fixed-point) — 32-bit to double-precision	-
0	0	01	11	000		FCVTZS (scalar, fixed-point) — double-precision to 32-bit	-
0	0	01	11	001		FCVTZU (scalar, fixed-point) — double-precision to 32-bit	-
0	0	11	00	010		SCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011		UCVTF (scalar, fixed-point) — 32-bit to half-precision	FEAT_FP16
0	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 32-bit	FEAT_FP16
1	0	00	00	010		SCVTF (scalar, fixed-point) — 64-bit to single-precision	-
1	0	00	00	011		UCVTF (scalar, fixed-point) — 64-bit to single-precision	-
1	0	00	11	000		FCVTZS (scalar, fixed-point) — single-precision to 64-bit	-
1	0	00	11	001		FCVTZU (scalar, fixed-point) — single-precision to 64-bit	-
1	0	01	00	010		SCVTF (scalar, fixed-point) — 64-bit to double-precision	-
1	0	01	00	011		UCVTF (scalar, fixed-point) — 64-bit to double-precision	-
1	0	01	11	000		FCVTZS (scalar, fixed-point) — double-precision to 64-bit	-
1	0	01	11	001		FCVTZU (scalar, fixed-point) — double-precision to 64-bit	-

sf	S	ptype	rmode	opcode	scale	Instruction Details	Feature
1	0	11	00	010		SCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011		UCVTF (scalar, fixed-point) — 64-bit to half-precision	FEAT_FP16
1	0	11	11	000		FCVTZS (scalar, fixed-point) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	001		FCVTZU (scalar, fixed-point) — half-precision to 64-bit	FEAT_FP16

Conversion between floating-point and integer

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 19	18 16	15					10	9 5	4 0
sf	0	S	1	1	1	1	0	ptype	1	rmode	opcode	0	0	0	0	0	0	Rn	Rd

sf	S	ptype	rmode	opcode	Instruction Details	Feature
			x1	01x	UNALLOCATED	-
			x1	10x	UNALLOCATED	-
			1x	01x	UNALLOCATED	-
			1x	10x	UNALLOCATED	-
	0	10		0xx	UNALLOCATED	-
	0	10		10x	UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	x1	11x	UNALLOCATED	-
0	0	00	00	000	FCVTNS (scalar) — single-precision to 32-bit	-
0	0	00	00	001	FCVTNU (scalar) — single-precision to 32-bit	-
0	0	00	00	010	SCVTF (scalar, integer) — 32-bit to single-precision	-
0	0	00	00	011	UCVTF (scalar, integer) — 32-bit to single-precision	-
0	0	00	00	100	FCVTAS (scalar) — single-precision to 32-bit	-
0	0	00	00	101	FCVTAU (scalar) — single-precision to 32-bit	-
0	0	00	00	110	FMOV (general) — single-precision to 32-bit	-
0	0	00	00	111	FMOV (general) — 32-bit to single-precision	-
0	0	00	01	000	FCVTPS (scalar) — single-precision to 32-bit	-

sf	S	ptype	rmode	opcode	Instruction Details	Feature
0	0	00	01	001	FCVTPU (scalar) — single-precision to 32-bit	-
0	0	00	1x	11x	UNALLOCATED	-
0	0	00	10	000	FCVTMS (scalar) — single-precision to 32-bit	-
0	0	00	10	001	FCVTMU (scalar) — single-precision to 32-bit	-
0	0	00	11	000	FCVTZS (scalar, integer) — single-precision to 32-bit	-
0	0	00	11	001	FCVTZU (scalar, integer) — single-precision to 32-bit	-
0	0	01	0x	11x	UNALLOCATED	-
0	0	01	00	000	FCVTNS (scalar) — double-precision to 32-bit	-
0	0	01	00	001	FCVTNU (scalar) — double-precision to 32-bit	-
0	0	01	00	010	SCVTF (scalar, integer) — 32-bit to double-precision	-
0	0	01	00	011	UCVTF (scalar, integer) — 32-bit to double-precision	-
0	0	01	00	100	FCVTAS (scalar) — double-precision to 32-bit	-
0	0	01	00	101	FCVTAU (scalar) — double-precision to 32-bit	-
0	0	01	01	000	FCVTPS (scalar) — double-precision to 32-bit	-
0	0	01	01	001	FCVTPU (scalar) — double-precision to 32-bit	-
0	0	01	10	000	FCVTMS (scalar) — double-precision to 32-bit	-
0	0	01	10	001	FCVTMU (scalar) — double-precision to 32-bit	-
0	0	01	10	11x	UNALLOCATED	-
0	0	01	11	000	FCVTZS (scalar, integer) — double-precision to 32-bit	-
0	0	01	11	001	FCVTZU (scalar, integer) — double-precision to 32-bit	-
0	0	01	11	111	UNALLOCATED	-
0	0	10		11x	UNALLOCATED	-
0	0	11	00	000	FCVTNS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	001	FCVTNU (scalar) — half-precision to 32-bit	FEAT_FP16

sf	S	ptype	rmode	opcode	Instruction Details	Feature
0	0	11	00	010	SCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	011	UCVTF (scalar, integer) — 32-bit to half-precision	FEAT_FP16
0	0	11	00	100	FCVTAS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	101	FCVTAU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	110	FMOV (general) — half-precision to 32-bit	FEAT_FP16
0	0	11	00	111	FMOV (general) — 32-bit to half-precision	FEAT_FP16
0	0	11	01	000	FCVTPS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	01	001	FCVTPU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	10	000	FCVTMS (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	10	001	FCVTMU (scalar) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 32-bit	FEAT_FP16
0	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 32-bit	FEAT_FP16
1	0	00		11x	UNALLOCATED	-
1	0	00	00	000	FCVTNS (scalar) — single-precision to 64-bit	-
1	0	00	00	001	FCVTNU (scalar) — single-precision to 64-bit	-
1	0	00	00	010	SCVTF (scalar, integer) — 64-bit to single-precision	-
1	0	00	00	011	UCVTF (scalar, integer) — 64-bit to single-precision	-
1	0	00	00	100	FCVTAS (scalar) — single-precision to 64-bit	-
1	0	00	00	101	FCVTAU (scalar) — single-precision to 64-bit	-
1	0	00	01	000	FCVTPS (scalar) — single-precision to 64-bit	-
1	0	00	01	001	FCVTPU (scalar) — single-precision to 64-bit	-
1	0	00	10	000	FCVTMS (scalar) — single-precision to 64-bit	-

sf	S	ptype	rmode	opcode	Instruction Details	Feature
1	0	00	10	001	FCVTMU (scalar) — single-precision to 64-bit	-
1	0	00	11	000	FCVTZS (scalar, integer) — single-precision to 64-bit	-
1	0	00	11	001	FCVTZU (scalar, integer) — single-precision to 64-bit	-
1	0	01	x1	11x	UNALLOCATED	-
1	0	01	00	000	FCVTNS (scalar) — double-precision to 64-bit	-
1	0	01	00	001	FCVTNU (scalar) — double-precision to 64-bit	-
1	0	01	00	010	SCVTF (scalar, integer) — 64-bit to double-precision	-
1	0	01	00	011	UCVTF (scalar, integer) — 64-bit to double-precision	-
1	0	01	00	100	FCVTAS (scalar) — double-precision to 64-bit	-
1	0	01	00	101	FCVTAU (scalar) — double-precision to 64-bit	-
1	0	01	00	110	FMOV (general) — double-precision to 64-bit	-
1	0	01	00	111	FMOV (general) — 64-bit to double-precision	-
1	0	01	01	000	FCVTPS (scalar) — double-precision to 64-bit	-
1	0	01	01	001	FCVTPU (scalar) — double-precision to 64-bit	-
1	0	01	1x	11x	UNALLOCATED	-
1	0	01	10	000	FCVTMS (scalar) — double-precision to 64-bit	-
1	0	01	10	001	FCVTMU (scalar) — double-precision to 64-bit	-
1	0	01	11	000	FCVTZS (scalar, integer) — double-precision to 64-bit	-
1	0	01	11	001	FCVTZU (scalar, integer) — double-precision to 64-bit	-
1	0	10	x0	11x	UNALLOCATED	-
1	0	10	01	110	FMOV (general) — top half of 128-bit to 64-bit	-
1	0	10	01	111	FMOV (general) — 64-bit to top half of 128-bit	-
1	0	10	1x	11x	UNALLOCATED	-

sf	S	ptype	rmode	opcode	Instruction Details	Feature
1	0	11	00	000	FCVTNS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	001	FCVTNU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	010	SCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	011	UCVTF (scalar, integer) — 64-bit to half-precision	FEAT_FP16
1	0	11	00	100	FCVTAS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	101	FCVTAU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	110	FMOV (general) — half-precision to 64-bit	FEAT_FP16
1	0	11	00	111	FMOV (general) — 64-bit to half-precision	FEAT_FP16
1	0	11	01	000	FCVTPS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	01	001	FCVTPU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	10	000	FCVTMS (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	10	001	FCVTMU (scalar) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	000	FCVTZS (scalar, integer) — half-precision to 64-bit	FEAT_FP16
1	0	11	11	001	FCVTZU (scalar, integer) — half-precision to 64-bit	FEAT_FP16

Floating-point data-processing (1 source)

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 15	14 10	9 5	4 0
M	0	S	1 1 1 1 0	ptype	1	opcode	1 0 0 0 0	Rn	Rd

M	S	ptype	opcode	Instruction Details	Feature
			1xxxxx	UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	000000	FMOV (register) — single-precision	-
0	0	00	000001	FABS (scalar) — single-precision	-
0	0	00	000010	FNEG (scalar) — single-precision	-
0	0	00	000011	FSQRT (scalar) — single-precision	-
0	0	00	000100	UNALLOCATED	-

M	S	ptype	opcode	Instruction Details	Feature
0	0	00	000101	FCVT — single-precision to double-precision	-
0	0	00	000110	UNALLOCATED	-
0	0	00	000111	FCVT — single-precision to half-precision	-
0	0	00	001000	FRINTN (scalar) — single-precision	-
0	0	00	001001	FRINTP (scalar) — single-precision	-
0	0	00	001010	FRINTM (scalar) — single-precision	-
0	0	00	001011	FRINTZ (scalar) — single-precision	-
0	0	00	001100	FRINTA (scalar) — single-precision	-
0	0	00	001101	UNALLOCATED	-
0	0	00	001110	FRINTX (scalar) — single-precision	-
0	0	00	001111	FRINTI (scalar) — single-precision	-
0	0	00	0101xx	UNALLOCATED	-
0	0	00	011xxx	UNALLOCATED	-
0	0	01	000000	FMOV (register) — double-precision	-
0	0	01	000001	FABS (scalar) — double-precision	-
0	0	01	000010	FNEG (scalar) — double-precision	-
0	0	01	000011	FSQRT (scalar) — double-precision	-
0	0	01	000100	FCVT — double-precision to single-precision	-
0	0	01	000101	UNALLOCATED	-
0	0	01	000111	FCVT — double-precision to half-precision	-
0	0	01	001000	FRINTN (scalar) — double-precision	-
0	0	01	001001	FRINTP (scalar) — double-precision	-
0	0	01	001010	FRINTM (scalar) — double-precision	-
0	0	01	001011	FRINTZ (scalar) — double-precision	-
0	0	01	001100	FRINTA (scalar) — double-precision	-
0	0	01	001101	UNALLOCATED	-
0	0	01	001110	FRINTX (scalar) — double-precision	-
0	0	01	001111	FRINTI (scalar) — double-precision	-
0	0	01	0101xx	UNALLOCATED	-
0	0	01	011xxx	UNALLOCATED	-
0	0	10	0xxxxx	UNALLOCATED	-
0	0	11	000000	FMOV (register) — half-precision	FEAT_FP16

M	S	ptype	opcode	Instruction Details	Feature
0	0	11	000001	FABS (scalar) — half-precision	FEAT_FP16
0	0	11	000010	FNEG (scalar) — half-precision	FEAT_FP16
0	0	11	000011	FSQRT (scalar) — half-precision	FEAT_FP16
0	0	11	000100	FCVT — half-precision to single-precision	-
0	0	11	000101	FCVT — half-precision to double-precision	-
0	0	11	00011x	UNALLOCATED	-
0	0	11	001000	FRINTN (scalar) — half-precision	FEAT_FP16
0	0	11	001001	FRINTP (scalar) — half-precision	FEAT_FP16
0	0	11	001010	FRINTM (scalar) — half-precision	FEAT_FP16
0	0	11	001011	FRINTZ (scalar) — half-precision	FEAT_FP16
0	0	11	001100	FRINTA (scalar) — half-precision	FEAT_FP16
0	0	11	001101	UNALLOCATED	-
0	0	11	001110	FRINTX (scalar) — half-precision	FEAT_FP16
0	0	11	001111	FRINTI (scalar) — half-precision	FEAT_FP16
0	0	11	01xxxx	UNALLOCATED	-
1				UNALLOCATED	-

Floating-point compare

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 16	15 14	13 10	9 5	4 0
M	0	S	1 1 1 1 0	ptype	1	Rm	op	1 0 0 0	Rn	opcode2

M	S	ptype	op	opcode2	Instruction Details	Feature
				xxxx1	UNALLOCATED	-
				xxx1x	UNALLOCATED	-
				xx1xx	UNALLOCATED	-
			x1		UNALLOCATED	-
			1x		UNALLOCATED	-
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	00	00000	FCMP	-
0	0	00	00	01000	FCMP	-
0	0	00	00	10000	FCMPE	-
0	0	00	00	11000	FCMPE	-
0	0	01	00	00000	FCMP	-

M	S	ptype	op	opcode2	Instruction Details	Feature
0	0	01	00	01000	FCMP	-
0	0	01	00	10000	FCMPE	-
0	0	01	00	11000	FCMPE	-
0	0	11	00	00000	FCMP	FEAT_FP16
0	0	11	00	01000	FCMP	FEAT_FP16
0	0	11	00	10000	FCMPE	FEAT_FP16
0	0	11	00	11000	FCMPE	FEAT_FP16
1					UNALLOCATED	-

Floating-point immediate

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 13	12 10	9 5	4 0
M	0	S	1 1 1 1 0	ptype	1	imm8	1 0 0	imm5	Rd

M	S	ptype	imm5	Instruction Details	Feature
			xxxx1	UNALLOCATED	-
			xxx1x	UNALLOCATED	-
			xx1xx	UNALLOCATED	-
			x1xxx	UNALLOCATED	-
			1xxxx	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	00000	FMOV (scalar, immediate) — single-precision	-
0	0	01	00000	FMOV (scalar, immediate) — double-precision	-
0	0	11	00000	FMOV (scalar, immediate) — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point conditional compare

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 16	15 12	11 10	9 5	4	3 0
M	0	S	1 1 1 1 0	ptype	1	Rm	cond	0 1	Rn	op	nzcv

M	S	ptype	op	Instruction Details	Feature
		10		UNALLOCATED	-
	1			UNALLOCATED	-

M	S	ptype	op	Instruction Details	Feature
0	0	00	0	FCCMP — single-precision	-
0	0	00	1	FCCMPE — single-precision	-
0	0	01	0	FCCMP — double-precision	-
0	0	01	1	FCCMPE — double-precision	-
0	0	11	0	FCCMP — half-precision	FEAT_FP16
0	0	11	1	FCCMPE — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point data-processing (2 source)

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28 24	23 22	21	20 16	15 12	11	10	9 5	4 0
M	0	S	1 1 1 1 0	ptype	1	Rm	opcode	1	0	Rn	Rd

M	S	ptype	opcode	Instruction Details	Feature
			1xx1	UNALLOCATED	-
			1x1x	UNALLOCATED	-
			11xx	UNALLOCATED	-
		10		UNALLOCATED	-
	1			UNALLOCATED	-
0	0	00	0000	FMUL (scalar) — single-precision	-
0	0	00	0001	FDIV (scalar) — single-precision	-
0	0	00	0010	FADD (scalar) — single-precision	-
0	0	00	0011	FSUB (scalar) — single-precision	-
0	0	00	0100	FMAX (scalar) — single-precision	-
0	0	00	0101	FMIN (scalar) — single-precision	-
0	0	00	0110	FMAXNM (scalar) — single-precision	-
0	0	00	0111	FMINNM (scalar) — single-precision	-
0	0	00	1000	FNMUL (scalar) — single-precision	-
0	0	01	0000	FMUL (scalar) — double-precision	-
0	0	01	0001	FDIV (scalar) — double-precision	-
0	0	01	0010	FADD (scalar) — double-precision	-
0	0	01	0011	FSUB (scalar) — double-precision	-
0	0	01	0100	FMAX (scalar) — double-precision	-
0	0	01	0101	FMIN (scalar) — double-precision	-
0	0	01	0110	FMAXNM (scalar) — double-precision	-

M	S	ptype	opcode	Instruction Details	Feature
0	0	01	0111	FMINNM (scalar) — double-precision	-
0	0	01	1000	FNMUL (scalar) — double-precision	-
0	0	11	0000	FMUL (scalar) — half-precision	FEAT_FP16
0	0	11	0001	FDIV (scalar) — half-precision	FEAT_FP16
0	0	11	0010	FADD (scalar) — half-precision	FEAT_FP16
0	0	11	0011	FSUB (scalar) — half-precision	FEAT_FP16
0	0	11	0100	FMAX (scalar) — half-precision	FEAT_FP16
0	0	11	0101	FMIN (scalar) — half-precision	FEAT_FP16
0	0	11	0110	FMAXNM (scalar) — half-precision	FEAT_FP16
0	0	11	0111	FMINNM (scalar) — half-precision	FEAT_FP16
0	0	11	1000	FNMUL (scalar) — half-precision	FEAT_FP16
1				UNALLOCATED	-

Floating-point conditional select

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 16	15 12	11	10	9 5	4 0
M	0	S	1	1	1	1	0	ptype	1	Rm	cond	1	1	Rn	Rd

M	S	ptype	Instruction Details	Feature
		10	UNALLOCATED	-
	1		UNALLOCATED	-
0	0	00	FCSEL — single-precision	-
0	0	01	FCSEL — double-precision	-
0	0	11	FCSEL — half-precision	FEAT_FP16
1			UNALLOCATED	-

Floating-point data-processing (3 source)

These instructions are under Data Processing – Scalar Floating-Point and Advanced SIMD.

31	30	29	28				24	23 22	21	20 16	15	14 10	9 5	4 0
M	0	S	1	1	1	1	1	ptype	o1	Rm	o0	Ra	Rn	Rd

M	S	ptype	o1	o0	Instruction Details	Feature
		10			UNALLOCATED	-
	1				UNALLOCATED	-
0	0	00	0	0	FMADD — single-precision	-
0	0	00	0	1	FMSUB — single-precision	-
0	0	00	1	0	FNMADD — single-precision	-

M	S	ptype	o1	o0	Instruction Details	Feature
0	0	00	1	1	FNMSUB — single-precision	-
0	0	01	0	0	FMADD — double-precision	-
0	0	01	0	1	FMSUB — double-precision	-
0	0	01	1	0	FNMADD — double-precision	-
0	0	01	1	1	FNMSUB — double-precision	-
0	0	11	0	0	FMADD — half-precision	FEAT_FP16
0	0	11	0	1	FMSUB — half-precision	FEAT_FP16
0	0	11	1	0	FNMADD — half-precision	FEAT_FP16
0	0	11	1	1	FNMSUB — half-precision	FEAT_FP16
1					UNALLOCATED	-

Chapter 5

Pseudocode definitions

This chapter contains pseudocode that describes many features of the Morello architecture.

See also:

- Appendix K13, *Arm Pseudocode Definition*, *Arm® Architecture Reference Manual, Armv8-A*: additional information for understanding the Arm pseudocode.

5.1 aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =========================
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress,  integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.BRPs);

    enabled = DBGBCR_EL1[n].E == '1';
    ispriv = PSTATE.EL != EL0;
    linked = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt = TRUE;
    linked_to = FALSE;

    state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                     linked, DBGBCR_EL1[n].LBN, isbreakpnt, ispriv);
    value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

    if HaveAnyAArch32() && size == 4 then                  // Check second halfword
        // If the breakpoint address and BAS of an Address breakpoint match the address of the
        // second halfword of an instruction, but not the address of the first halfword, it is
        // CONSTRAINED UNPREDICTABLE whether or not this breakpoint generates a Breakpoint debug
        // event.
        match_i = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to);
        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    if vaddress<1> == '1' && DBGBCR_EL1[n].BAS == '1111' then
        // The above notwithstanding, if DBGBCR_EL1[n].BAS == '1111', then it is CONSTRAINED
        // UNPREDICTABLE whether or not a Breakpoint debug event is generated for an instruction
        // at the address DBGBVR_EL1[n]+2.
        if value_match then value_match = ConstrainUnpredictableBool(Unpredictable_BPMATCHHALF);

    match = value_match && state_match && enabled;

    return match;
```

5.2 aarch64/debug/breakpoint/AArch64.BreakpointValueMatch

```
// AArch64.BreakpointValueMatch()
// ==============================

boolean AArch64.BreakpointValueMatch(integer n, bits(64) vaddress, boolean linked_to)

    // "n" is the identity of the breakpoint unit to match against.
    // "vaddress" is the current instruction address, ignored if linked_to is TRUE and for Context
    //   matching breakpoints.
    // "linked_to" is TRUE if this is a call from StateMatch for linking.

    // If a non-existent breakpoint then it is CONSTRAINED UNPREDICTABLE whether this gives
    // no match or the breakpoint is mapped to another UNKNOWN implemented breakpoint.
    if n > UInt(ID_AA64DFR0_EL1.BRPs) then
        (c, n) = ConstrainUnpredictableInteger(0, UInt(ID_AA64DFR0_EL1.BRPs), Unpredictable_BPNOTIMPL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;

    // If this breakpoint is not enabled, it cannot generate a match. (This could also happen on a
    // call from StateMatch for linking).
    if DBGBCR_EL1[n].E == '0' then return FALSE;

    context_aware = (n >= UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));

    // If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
    dbgtype = DBGBCR_EL1[n].BT;

    if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt()) ||               // Context matching
          dbgtype == '010x' ||                                              // Reserved
          (dbgtype != '0x0x' && !context_aware) ||                          // Context matching
          (dbgtype == '1xxx' && !HaveEL(EL2))) then                         // EL2 extension
        (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then return FALSE;
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    // Determine what to compare against.
    match_addr = (dbgtype == '0x0x');
    match_vmid = (dbgtype == '10xx');
    match_cid  = (dbgtype == '001x');
    match_cid1 = (dbgtype IN { '101x', 'x11x'});
    match_cid2 = (dbgtype == '11xx');
    linked     = (dbgtype == 'xxx1');

    // If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
    // VMID and/or context ID match, of if not context-aware. The above assertions mean that the
    // code can just test for match_addr == TRUE to confirm all these things.
    if linked_to && (!linked || match_addr) then return FALSE;

    // If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
    if !linked_to && linked && !match_addr then return FALSE;

    // Do the comparison.
    if match_addr then
        byte = UInt(vaddress<1:0>);
        if HaveAnyAArch32() then
            // T32 instructions can be executed at EL0 in an AArch64 translation regime.
            assert byte IN {0,2};                 // "vaddress" is halfword aligned
            byte_select_match = (DBGBCR_EL1[n].BAS<byte> == '1');
        else
            assert byte == 0;                     // "vaddress" is word aligned
            byte_select_match = TRUE;             // DBGBCR_EL1[n].BAS<byte> is RES1
        top = AddrTop(vaddress, PSTATE.EL);
        BVR_match = vaddress<top:2> == DBGBVR_EL1[n]<top:2> && byte_select_match;
    elsif match_cid then
        if IsInHost() then
            BVR_match = (CONTEXTIDR_EL2 == DBGBVR_EL1[n]<31:0>);
        else
            BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    elsif match_cid1 then
        BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1 == DBGBVR_EL1[n]<31:0>);
    if match_vmid then
        if !Have16bitVMID() || VTCR_EL2.VS == '0' then
            vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
            bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
        else
            vmid = VTTBR_EL2.VMID;
            bvr_vmid = DBGBVR_EL1[n]<47:32>;
        BXVR_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                      !IsInHost() &&
                      vmid == bvr_vmid);
    elsif match_cid2 then
        BXVR_match = (!IsSecure() && HaveVirtHostExt() &&
                      DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2);

    bvr_match_valid = (match_addr || match_cid || match_cid1);
    bxvr_match_valid = (match_vmid || match_cid2);

    match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

    return match;
```

5.3 aarch64/debug/breakpoint/AArch64.StateMatch

```
// AArch64.StateMatch()
// ====================
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean linked, bits(4) LBN,
                           boolean isbreakpnt,  boolean ispriv)
    // "SSC", "HMC", "PxC" are the control fields from the DBGBCR[n] or DBGWCR[n] register.
    // "linked" is TRUE if this is a linked breakpoint/watchpoint type.
    // "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
    // "isbreakpnt" is TRUE for breakpoints, FALSE for watchpoints.
    // "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.

    // If parameters are set to a reserved type, behaves as either disabled or a defined type
    (c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpnt);
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the HMC,SSC,PxC values are either valid or the values returned by
    // CheckValidStateMatch are valid.

    EL3_match = HaveEL(EL3) && HMC == '1' && SSC<0> == '0';
    EL2_match = HaveEL(EL2) && ((HMC == '1' && (SSC:PxC != '1000')) || SSC == '11');
    EL1_match = PxC<0> == '1';
    EL0_match = PxC<1> == '1';

    if !ispriv && !isbreakpnt then
        priv_match = EL0_match;
    else
        case PSTATE.EL of
            when EL3  priv_match = EL3_match;
            when EL2  priv_match = EL2_match;
            when EL1  priv_match = EL1_match;
            when EL0  priv_match = EL0_match;

    case SSC of
        when '00'  security_state_match = TRUE;                          // Both
        when '01'  security_state_match = !IsSecure();                   // Non-secure only
        when '10'  security_state_match = IsSecure();                    // Secure only
        when '11'  security_state_match = (HMC == '1' || IsSecure());  // HMC=1 -> Both, 0 -> Secure only

    if linked then
        // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
        // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
        // UNKNOWN breakpoint that is context-aware.
        lbn = UInt(LBN);
        first_ctx_cmp = (UInt(ID_AA64DFR0_EL1.BRPs) - UInt(ID_AA64DFR0_EL1.CTX_CMPs));
        last_ctx_cmp = UInt(ID_AA64DFR0_EL1.BRPs);
        if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
            (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTXCMP);
            assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
            case c of
                when Constraint_DISABLED  return FALSE;      // Disabled
                when Constraint_NONE      linked = FALSE;    // No linking
                // Otherwise ConstrainUnpredictableInteger returned a context-aware breakpoint

    if linked then
        vaddress = bits(64) UNKNOWN;
        linked_to = TRUE;
        linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

    return priv_match && security_state_match && (!linked || linked_match);
```

5.4 aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =================================

boolean AArch64.GenerateDebugExceptions()
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, IsSecure(), PSTATE.D);
```

5.5 aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====================================

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from, boolean secure, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = HaveEL(EL2) && !secure && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');
    target = (if route_to_el2 then EL2 else EL1);

    enabled = !HaveEL(EL3) || !secure || MDCR_EL3.SDD == '0';

    if from == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from);

    return enabled;
```

5.6 aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =============================
// Signal Performance Monitors overflow IRQ and CTI overflow events

boolean AArch64.CheckForPMUOverflow()

    pmuirq = PMCR_EL0.E == '1' && PMINTENSET_EL1<31> == '1' && PMOVSSET_EL0<31> == '1';
    for n = 0 to UInt(PMCR_EL0.N) - 1
        if HaveEL(EL2) then
            E = (if n < UInt(MDCR_EL2.HPMN) then PMCR_EL0.E else MDCR_EL2.HPME);
        else
            E = PMCR_EL0.E;
        if E == '1' && PMINTENSET_EL1<n> == '1' && PMOVSSET_EL0<n> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

    CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else LOW);

    // The request remains set until the condition is cleared. (For example, an interrupt handler
    // or cross-triggered event handler clears the overflow status flag by writing to PMOVSCLR_EL0.)

    return pmuirq;
```

5.7 aarch64/debug/pmu/AArch64.CountEvents

```
// AArch64.CountEvents()
// =====================
// Return TRUE if counter "n" should count its event. For the cycle counter, n == 31.

boolean AArch64.CountEvents(integer n)
    assert n == 31 || n < UInt(PMCR_EL0.N);

    // Event counting is disabled in Debug state
    debug = Halted();

    // In Non-secure state, some counters are reserved for EL2
    if HaveEL(EL2) then
        E = if n < UInt(MDCR_EL2.HPMN) || n == 31 then PMCR_EL0.E else MDCR_EL2.HPME;
    else
        E = PMCR_EL0.E;
    enabled = E == '1' && PMCNTENSET_EL0<n> == '1';

    // Event counting in Secure state is prohibited unless any one of:
    // * EL3 is not implemented
    // * EL3 is using AArch64 and MDCR_EL3.SPME == 1
    prohibited = HaveEL(EL3) && IsSecure() && MDCR_EL3.SPME == '0';

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * Executing at EL2
    // * PMNx is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
    if !prohibited && HaveEL(EL2) && HaveHPMDExt() && PSTATE.EL == EL2 && (n < UInt(MDCR_EL2.HPMN) || n ==
        ↪31) then
        prohibited = (MDCR_EL2.HPMD == '1');

    // The IMPLEMENTATION DEFINED authentication interface might override software controls
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();
    // For the cycle counter, PMCR_EL0.DP enables counting when otherwise prohibited
    if prohibited && n == 31 then prohibited = (PMCR_EL0.DP == '1');

    // Event counting can be filtered by the {P, U, NSK, NSU, NSH, M} bits
    filter = if n == 31 then PMCCFILTR_EL0[31:0] else PMEVTYPER_EL0[n]<31:0>;

    P = filter<31>;
    U = filter<30>;
    NSK = if HaveEL(EL3) then filter<29> else '0';
    NSU = if HaveEL(EL3) then filter<28> else '0';
    NSH = if HaveEL(EL2) then filter<27> else '0';
    M = if HaveEL(EL3) then filter<26> else '0';

    case PSTATE.EL of
        when EL0 filtered = if IsSecure() then U == '1' else U != NSU;
        when EL1 filtered = if IsSecure() then P == '1' else P != NSK;
        when EL2 filtered = (NSH == '0');
        when EL3 filtered = (M != P);

    return !debug && enabled && !prohibited && !filtered;
```

5.8 aarch64/debug/statisticalprofiling/CheckProfilingBufferAccess

```
// CheckProfilingBufferAccess()
// ============================

SysRegAccess CheckProfilingBufferAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.E2PB<0> != '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

5.9 aarch64/debug/statisticalprofiling/CheckStatisticalProfilingAccess

```
// CheckStatisticalProfilingAccess()
// =================================

SysRegAccess CheckStatisticalProfilingAccess()
    if !HaveStatisticalProfiling() || PSTATE.EL == EL0 || UsingAArch32() then
        return SysRegAccess_UNDEFINED;

    if PSTATE.EL == EL1 && EL2Enabled() && MDCR_EL2.TPMS == '1' then
        return SysRegAccess_TrapToEL2;

    if HaveEL(EL3) && PSTATE.EL != EL3 && MDCR_EL3.NSPB != SCR_EL3.NS:'1' then
        return SysRegAccess_TrapToEL3;

    return SysRegAccess_OK;
```

5.10 aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// ====================

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled() then return FALSE;
    if  PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled() && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

5.11 aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// ====================

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if  EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

5.12 aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// ========================

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    if !secure && HaveEL(EL2) then
        return PMSCR_EL2.PA == '1' && (el == EL2 || PMSCR_EL1.PA == '1');
    else
        return PMSCR_EL1.PA == '1';
```

5.13 aarch64/debug/statisticalprofiling/CollectRecord

```
// CollectRecord()
// ===============

boolean CollectRecord(bits(64) events, integer total_latency, OpType optype)
    assert StatisticalProfilingEnabled();

    // Filtering by event
    if PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1) then
        bits(64) mask = 0xFFFF0000FF00F0AA<63:0>;    // Bits [63:48,31:24,15:12,7,5,3,1]
        if HaveStatisticalProfiling() then
            mask<11> = '1';                          // Alignment flag
        e = events AND mask;
        m = PMSEVFR_EL1 AND mask;
        if !IsZero(NOT(e) AND m) then return FALSE;

    // Filtering by type
    if PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>) then
        case optype of
            when OpType_Branch
                if PMSFCR_EL1.B == '0' then return FALSE;
            when OpType_Load
                if PMSFCR_EL1.LD == '0' then return FALSE;
            when OpType_Store
                if PMSFCR_EL1.ST == '0' then return FALSE;
            when OpType_LoadAtomic
                if PMSFCR_EL1.<LD,ST> == '00' then return FALSE;
            otherwise
                return FALSE;

    // Filtering by latency
    if PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT) then
        if total_latency < UInt(PMSLATFR_EL1.MINLAT) then
            return FALSE;

    // Check for UNPREDICTABLE cases
    if ((PMSFCR_EL1.FE == '1' && !IsZero(PMSEVFR_EL1)) ||
        (PMSFCR_EL1.FT == '1' && !IsZero(PMSFCR_EL1.<B,LD,ST>)) ||
        (PMSFCR_EL1.FL == '1' && !IsZero(PMSLATFR_EL1.MINLAT))) then
        return ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    return TRUE;
```

5.14 aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// ==================

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled() then return TimeStamp_None;
    (secure, el) = ProfilingBufferOwner();
    if el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp_None;
    if EL2Enabled() then
        pct = PMSCR_EL2.PCT == '01' && (el == EL2 || PMSCR_EL1.PCT == '01');
    else
        pct = PMSCR_EL1.PCT == '01';
    return (if pct then TimeStamp_Physical else TimeStamp_Virtual);
```

5.15 aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
 OpType_Load,                 // Any memory-read operation other than atomics, compare-and-swap, and swap
 OpType_Store,                // Any memory-write operation, including atomics without return
 OpType_LoadAtomic,           // Atomics with return, compare-and-swap and swap
 OpType_Branch,               // Software write to the PC
 OpType_Other                 // Any other class of operation
 };
```

5.16 aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// ========================

boolean ProfilingBufferEnabled()
    if !HaveStatisticalProfiling() then return FALSE;
    (secure, el) = ProfilingBufferOwner();
    non_secure_bit = if secure then '0' else '1';
    return (!ELUsingAArch32(el) && non_secure_bit == SCR_EL3.NS &&
            PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

5.17 aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// ======================

(boolean, bits(2)) ProfilingBufferOwner()
    secure = if HaveEL(EL3) then (MDCR_EL3.NSPB<1> == '0') else IsSecure();
    el = if !secure && HaveEL(EL2) && MDCR_EL2.E2PB == '00' then EL2 else EL1;
    return (secure, el);
```

5.18 aarch64/debug/statisticalprofiling/ProfilingSynchronizationBarrier

```
// Barrier to ensure that all existing profiling data has been formatted, and profiling buffer
// addresses have been translated such that writes to the profiling buffer have been initiated.
// A following DSB completes when writes to the profiling buffer have completed.
ProfilingSynchronizationBarrier();
```

5.19 aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =============================

boolean StatisticalProfilingEnabled()
    if !HaveStatisticalProfiling() || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    in_host = EL2Enabled() && HCR_EL2.TGE == '1';
    (secure, el) = ProfilingBufferOwner();
    if UInt(el) <  UInt(PSTATE.EL) || secure != IsSecure() || (in_host && el == EL1)  then
        return FALSE;

    case PSTATE.EL of
        when EL3  Unreachable();
        when EL2  spe_bit = PMSCR_EL2.E2SPE;
        when EL1  spe_bit = PMSCR_EL1.E1SPE;
        when EL0  spe_bit = (if in_host then PMSCR_EL2.E0HSPE else PMSCR_EL1.E0SPE);

    return spe_bit == '1';
```

5.20 aarch64/debug/statisticalprofiling/SysRegAccess

```
enumeration SysRegAccess { SysRegAccess_OK,
                           SysRegAccess_UNDEFINED,
                           SysRegAccess_TrapToEL1,
                           SysRegAccess_TrapToEL2,
                           SysRegAccess_TrapToEL3 };
```

5.21 aarch64/debug/statisticalprofiling/TimeStamp

```
enumeration TimeStamp {
    TimeStamp_None,        // No timestamp
    TimeStamp_CoreSight,   // CoreSight time (IMPLEMENTATION DEFINED)
    TimeStamp_Virtual,     // Physical counter value minus CNTVOFF_EL2
    TimeStamp_Physical };  // Physical counter value with no offset
```

5.22 aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// AArch64.TakeExceptionInDebugState()
// ===================================
// Take an exception in Debug state to an Exception Level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(32) UNKNOWN;

    if IsAccessToCapabilitiesEnabledAtEL(PSTATE.EL) then
        CELR[] = CapSetValue(PCC, bits(64) UNKNOWN);
    else
        ELR[] = bits(64) UNKNOWN;

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if UNKNOWN.
    PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;

    DSPSR_EL0 = bits(32) UNKNOWN;
    CDLR_EL0 = Capability UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields();                        // Update EDSCR processor state flags.

    if sync_errors then
        SynchronizeErrors();

    EndOfInstruction();
```

5.23 aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =============================

boolean AArch64.WatchpointByteMatch(integer n,  bits(64) vaddress)

    el = PSTATE.EL;
    top = AddrTop(vaddress, el);
    bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;            // Word or doubleword
    byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
    mask = UInt(DBGWCR_EL1[n].MASK);

    // If DBGWCR_EL1[n].MASK is non-zero value and DBGWCR_EL1[n].BAS is not set to '11111111', or
    // DBGWCR_EL1[n].BAS specifies a non-contiguous set of bytes behavior is CONSTRAINED
    // UNPREDICTABLE.
    if mask > 0 && !IsOnes(DBGWCR_EL1[n].BAS) then
        byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPMASKANDBAS);
    else
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1));  MSB = (DBGWCR_EL1[n].BAS + LSB);
        if !IsZero(MSB AND (MSB - 1)) then                    // Not contiguous
            byte_select_match = ConstrainUnpredictableBool(Unpredictable_WPBASCONTIGUOUS);
            bottom = 3;                                       // For the whole doubleword

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        (c, mask) = ConstrainUnpredictableInteger(3, 31, Unpredictable_RESWPMASK);
        assert c IN {Constraint_DISABLED, Constraint_NONE, Constraint_UNKNOWN};
        case c of
            when Constraint_DISABLED  return FALSE;           // Disabled
            when Constraint_NONE      mask = 0;               // No masking
            // Otherwise the value returned by ConstrainUnpredictableInteger is a not-reserved value

    if mask > bottom then
        WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
        // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
    else
        WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

    return WVR_match && byte_select_match;
```

5.24 aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =========================
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n <= UInt(ID_AA64DFR0_EL1.WRPs);

    // "ispriv" is FALSE for LDTR/STTR instructions executed at EL1 and all
    // load/stores at EL0, TRUE for all other load/stores. "iswrite" is TRUE for stores, FALSE for
    // loads.
    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, ispriv);

    ls_match = (DBGWCR_EL1[n].LSC<(if iswrite then 1 else 0)> == '1');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

5.25 aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// ===============
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, fault);
```

5.26 aarch64/exceptions/aborts/AArch64.AbortSyndrome

```
// AArch64.AbortSyndrome()
// =======================
// Creates an exception syndrome record for Abort and Watchpoint exceptions
// from an AArch64 translation regime.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(64) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype IN {Exception_DataAbort, Exception_Watchpoint};

    exception.syndrome = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.ipaddress = fault.ipaddress;
    else
        exception.ipavalid = FALSE;

    return exception;
```

5.27 aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// ==========================

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

5.28 aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// ===================

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)

    bits(2) cap_target_el;
    if fault.statuscode IN {Fault_CapTag, Fault_CapSeal, Fault_CapPerm, Fault_CapBounds} then
        cap_target_el = TargetELForCapabilityExceptions();
    else
        cap_target_el = EL0;
    route_to_el3 = (HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault)) || (cap_target_el == EL3);
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && (HCR_EL2.TGE == '1' ||
                    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                    (cap_target_el == EL2) ||
                    IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;
    exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.29 aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// ==========================

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    bits(2) cap_target_el;
    if fault.statuscode IN {Fault_CapTag, Fault_CapSeal, Fault_CapPerm, Fault_CapBounds} then
        cap_target_el = TargetELForCapabilityExceptions();
    else
        cap_target_el = EL0;
    route_to_el3 = (HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault)) || (cap_target_el == EL3);
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || IsSecondStage(fault) ||
                    (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault))));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.30 aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// ==========================
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PCAlignment);
    exception.vaddress = ThisInstrAddr();

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.31 aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// ==========================
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SPAlignment);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.32 aarch64/exceptions/aborts/CapabilityFault

```
// CapabilityFault()
// =================
// Generate a FaultRecord for a capability fault

FaultRecord CapabilityFault(Fault faulttype, AccType acctype, boolean iswrite)
    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;
    extflag = bit UNKNOWN;
    boolean ns = FALSE;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(faulttype, ipaddress,  level, acctype, iswrite,
                                     extflag,  errortype, secondstage, s2fs1walk);
```

5.33 aarch64/exceptions/aborts/CheckCapability

```
// CheckCapability()
// =================
// Check whether a capability is valid for accessing a given range of memory
// with a required set of permissions. If not generate an appropriate fault

bits(64) CheckCapability(Capability c, bits(64) address, integer size, bits(64) requested_perms, AccType acctype)

    // The below replicates and condenses the logic used in address translation
    // to recover the address as used for translation for input to bounds checks.
    el = AArch64.AccessUsesEL(acctype);
    msbit = AddrTop(address, el);
    s1_enabled = AArch64.IsStageOneEnabled(acctype);
    bits(64) addressforbounds = address;

    if msbit != 63 then
        if s1_enabled then
            if (PSTATE.EL IN {EL0, EL1} || ELIsInHost(el)) && address<msbit> == '1' then
                addressforbounds = SignExtend(address<msbit:0>);
            else
                addressforbounds = ZeroExtend(address<msbit:0>);
        else
            addressforbounds = ZeroExtend(address<msbit:0>);

    Fault fault_type = Fault_None;
    if CapIsTagClear(c) then
        fault_type = Fault_CapTag;
    elsif CapIsSealed(c) then
        fault_type = Fault_CapSeal;
    elsif !CapCheckPermissions(c, requested_perms) then
        fault_type = Fault_CapPerm;
    elsif ((requested_perms AND CAP_PERM_EXECUTE) != CAP_PERM_NONE) && !CapIsExecutePermitted(c) then
        fault_type = Fault_CapPerm;
    elsif !CapIsRangeInBounds(c, addressforbounds, size<64:0>) then
        fault_type = Fault_CapBounds;

    if fault_type != Fault_None then
        boolean is_store = CapPermsInclude(requested_perms, CAP_PERM_STORE);
        FaultRecord fault = CapabilityFault(fault_type, acctype, is_store);
        AArch64.Abort(address, fault);

    return address;
```

5.34 aarch64/exceptions/aborts/CheckPCCCapability

```
// CheckPCCCapability()
// ====================
// Check whether the current PCC is valid for instruction fetch and if not
// generate an appropriate fault

bits(64) CheckPCCCapability()
    return CheckCapability(PCC, CapGetValue(PCC), 4, CAP_PERM_EXECUTE, AccType_IFETCH);
```

5.35 aarch64/exceptions/asynch/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// ==================================

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.36 aarch64/exceptions/asynch/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// ==================================
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.37 aarch64/exceptions/asynch/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====================================

AArch64.TakePhysicalSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome<24> = if impdef_syndrome then '1' else '0';
    exception.syndrome<23:0> = syndrome;

    ClearPendingPhysicalSError();

    if PSTATE.EL == EL3 || route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.38 aarch64/exceptions/asynch/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =================================

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1';  // Virtual IRQ enabled if TGE==0 and FMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.39 aarch64/exceptions/asynch/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =================================

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1';  // Virtual IRQ enabled if TGE==0 and IMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.40 aarch64/exceptions/asynch/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// ====================================

AArch64.TakeVirtualSErrorException(boolean impdef_syndrome, bits(24) syndrome)

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';  // Virtual SError enabled if TGE==0 and AMO==1

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    exception = ExceptionSyndrome(Exception_SError);
    if HaveRASExt() then
        exception.syndrome<24>    = VSESR_EL2.IDS;
        exception.syndrome<23:0> = VSESR_EL2.ISS;
    else
        exception.syndrome<24> = if impdef_syndrome then '1' else '0';
        if impdef_syndrome then exception.syndrome<23:0> = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.41 aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =============================

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.42 aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// ============================

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.43 aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// ===============================

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.44 aarch64/exceptions/debug/AArch64.VectorCatchException

```
// AArch64.VectorCatchException()
// ==============================
// Vector Catch taken from EL0 or EL1 to EL2. This can only be called when debug exceptions are
// being routed to EL2, as Vector Catch is a legacy debug event.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

5.45 aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =============================

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception_Watchpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.46 aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// ========================
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il = if ThisInstrLength() == 32 then '1' else '0';
    from_32 = UsingAArch32();
    assert from_32 || il == '1';            // AArch64 instructions always 32-bit

    case exceptype of
        when Exception_Uncategorized        ec = 0x00; il = '1';
        when Exception_WFxTrap              ec = 0x01;
        when Exception_CP15RTTrap           ec = 0x03;           assert from_32;
        when Exception_CP15RRTTrap          ec = 0x04;           assert from_32;
        when Exception_CP14RTTrap           ec = 0x05;           assert from_32;
        when Exception_CP14DTTrap           ec = 0x06;           assert from_32;
        when Exception_AdvSIMDFPAccessTrap  ec = 0x07;
        when Exception_FPIDTrap             ec = 0x08;
        when Exception_CP14RRTTrap          ec = 0x0C;           assert from_32;
        when Exception_IllegalState         ec = 0x0E; il = '1';
        when Exception_SupervisorCall       ec = 0x11;
        when Exception_HypervisorCall       ec = 0x12;
        when Exception_MonitorCall          ec = 0x13;
        when Exception_SystemRegisterTrap   ec = 0x18;           assert !from_32;
        when Exception_InstructionAbort     ec = 0x20; il = '1';
        when Exception_PCAlignment          ec = 0x22; il = '1';
        when Exception_DataAbort            ec = 0x24;
        when Exception_SPAlignment          ec = 0x26; il = '1'; assert !from_32;
        when Exception_FPTrappedException   ec = 0x28;
        when Exception_CapabilityAccess     ec = 0x29;
        when Exception_CapabilitySysRegTrap ec = 0x2A;
        when Exception_SError               ec = 0x2F; il = '1';
        when Exception_Breakpoint           ec = 0x30; il = '1';
        when Exception_SoftwareStep         ec = 0x32; il = '1';
        when Exception_Watchpoint           ec = 0x34; il = '1';
        when Exception_SoftwareBreakpoint   ec = 0x38;
        when Exception_VectorCatch          ec = 0x3A; il = '1'; assert from_32;
        otherwise                           Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    return (ec,il);
```

5.47 aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =========================
// Report syndrome information for exception taken to AArch64 state.

```

```
AArch64.ReportException(ExceptionRecord exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;

    // IL is not valid for Data Abort exceptions without valid instruction syndrome information
    if ec IN {0x24,0x25} && iss<24> == '0' then
        il = '1';

    ESR[target_el] = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAlignment, Exception_DataAbort,
                Exception_Watchpoint} then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if target_el == EL2 then
        if exception.ipavalid then
            HPFAR_EL2<39:4> = exception.ipaddress<47:12>;
        else
            HPFAR_EL2<39:4> = bits(36) UNKNOWN;

    return;
```

5.48 aarch64/exceptions/exceptions/AArch64.ResetControlRegisters

```
// Resets System registers and memory-mapped control registers that have architecturally-defined
// reset values to those values.
AArch64.ResetControlRegisters(boolean cold_reset);
```

5.49 aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// ===================
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    assert !HighestELUsingAArch32();

    // Enter the highest implemented Exception level in AArch64 state
    PSTATE.nRW = '0';
    if HaveEL(EL3) then
        PSTATE.EL = EL3;
    elsif HaveEL(EL2) then
        PSTATE.EL = EL2;
    else
        PSTATE.EL = EL1;

    // Reset the system registers and other system components
    AArch64.ResetControlRegisters(cold_reset);

    // Reset all other PSTATE fields
    PSTATE.SP = '1';              // Select stack pointer
    PSTATE.<D,A,I,F>  = '1111';   // All asynchronous exceptions masked
    PSTATE.SS = '0';              // Clear software step bit
    PSTATE.C64 = '0';             // Set default instruction set state
    PSTATE.IL = '0';              // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    AArch64.ResetSIMDFPRegisters();
    AArch64.ResetSpecialRegisters();
    ResetExternalDebugRegisters(cold_reset);

    bits(64) rv;                      // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        rv = RVBAR_EL3;
    elsif HaveEL(EL2) then
```

```
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    assert IsZero(rv<63:PAMax()>) && IsZero(rv<1:0>);

    BranchTo(rv, BranchType_RESET);
```

5.50 aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// ============================

AArch64.FPTrappedException(boolean is_ase, integer element, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome(Exception_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1';                          // TFV
        else
            exception.syndrome<23> = '0';                          // TFV
    else
        exception.syndrome<23> = '1';                              // TFV
    exception.syndrome<10:8> = bits(3) UNKNOWN;                    // VECITR
    if exception.syndrome<23> == '1' then
        exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        exception.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.51 aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// ========================
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_HypervisorCall);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

5.52 aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// ===========================

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL(EL3) && !ELUsingAArch32(EL3);
    SSAdvance();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_MonitorCall);
    exception.syndrome<15:0> = immediate;

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

5.53 aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// ========================
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)

    SSAdvance();
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SupervisorCall);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.54 aarch64/exceptions/takeexception/AArch64.TakeException

```
// AArch64.TakeException()
// =======================
// Take an exception to an Exception Level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception,
                      bits(64) preferred_exception_return, integer vect_offset)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();

    if UInt(target_el) > UInt(PSTATE.EL) then
        boolean lower_32;
        if target_el == EL3 then
            if EL2Enabled() then
                lower_32 = ELUsingAArch32(EL2);
            else
                lower_32 = ELUsingAArch32(EL1);
        elsif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
            lower_32 = ELUsingAArch32(EL0);
        else
            lower_32 = ELUsingAArch32(target_el - 1);
        vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

    elsif PSTATE.SP == '1' && !IsInRestricted() then
        vect_offset = vect_offset + 0x200;

    spsr = GetPSRFromPSTATE();

    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = spsr;

    if IsAccessToCapabilitiesEnabledAtEL(PSTATE.EL) then
        CELR[] = CapSetValue(PCC, preferred_exception_return);
    else
        ELR[] = preferred_exception_return;

    PSTATE.SS = '0';
    PSTATE.<D,A,I,F> = '1111';
    PSTATE.IL = '0';
    if from_32 then                             // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0';                         // PSTATE.J is RES0
    if (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;

    if IsAccessToCapabilitiesEnabledAtEL(PSTATE.EL) then
        PSTATE.C64 = CCTLR[].C64E;
        Capability c = CVBAR[];
        bits(64)   v = CapGetValue(c);
        c = CapSetValue(c, v<63:11>:vect_offset<10:0>);
        BranchToCapability(c, BranchType_EXCEPTION);
    else
        PSTATE.C64 = '0';
        BranchTo(VBAR[]<63:11>:vect_offset<10:0>, BranchType_EXCEPTION);

    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

    EndOfInstruction();
```

5.55 aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```
// AArch64.AArch32SystemAccessTrap()
// =================================
// Trapped AARCH32 system register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

5.56 aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrapSyndrome

```
// AArch64.AArch32SystemAccessTrapSyndrome()
// =========================================
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions,
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTrap, Exception_CP15RTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.exceptype != Exception_FPIDTrap then    // When trap is not for VMRS
            iss<19:17> = instr<7:5>;           // opc2
            iss<16:14> = instr<23:21>;         // opc1
            iss<13:10> = instr<19:16>;         // CRn
            iss<4:1>   = instr<3:0>;           // CRm
        else
            iss<19:17> = '000';
            iss<16:14> = '111';
            iss<13:10> = instr<19:16>;         // reg
            iss<4:1>   = '0000';

        if instr<20> == '1' && instr<15:12> == '1111' then    // MRC, Rt==15
            iss<9:5> = '11111';
        elsif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap,
        ↪Exception_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        iss<19:16> = instr<7:4>;          // opc1
        if instr<19:16> == '1111' then    // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then    // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        iss<4:1>   = instr<3:0>;         // CRm
    elsif exception.exceptype == Exception_CP14DTTrap then
        // Trapped LDC/STC
        iss<19:12> = instr<7:0>;         // imm8
        iss<4>     = instr<23>;          // U
        iss<2:1>   = instr<24,21>;       // P,W
        if instr<19:16> == '1111' then   // Rn==15, LDC(Literal addressing)/STC
            iss<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
    elsif exception.exceptype == Exception_Uncategorized then
        // Trapped for unknown reason
        iss<9:5> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>; // Rn
        iss<3>   = '0';

    iss<0> = instr<20>;                  // Direction

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0>  = iss;

    return exception;
```

5.57 aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =============================
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

5.58 aarch64/exceptions/traps/AArch64.CheckCP15InstrCoarseTraps

```
// AArch64.CheckCP15InstrCoarseTraps()
// ===================================
// Check for coarse-grained AArch32 CP15 traps in HSTR_EL2 and HCR_EL2.

boolean AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MCR, MRC, MCRR and MRRC disabled by HSTR_EL2<CRn/CRm>
        major = if nreg == 1 then CRn else CRm;
        if !IsInHost() && !(major IN {4,14}) && HSTR_EL2<major> == '1' then
            return TRUE;

        // Check for MRC and MCR disabled by HCR_EL2.TIDCP
        if (HCR_EL2.TIDCP == '1' && nreg == 1 &&
            ((CRn == 9  && CRm IN {0,1,2,    5,6,7,8   }) ||
             (CRn == 10 && CRm IN {0,1,    4,      8   }) ||
             (CRn == 11 && CRm IN {0,1,2,3,4,5,6,7,8,15}))) then
            return TRUE;

    return FALSE;
```

5.59 aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// ===============================
// Check against CPACR[]

AArch64.CheckFPAdvSIMDEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost() then
        // Check if access disabled in CPACR_EL1
        case CPACR[].FPEN of
            when 'x0'  disabled = TRUE;
            when '01'  disabled = PSTATE.EL == EL0;
            when '11'  disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    AArch64.CheckFPAdvSIMDTrap();               // Also check against CPTR_EL2 and CPTR_EL3
```

5.60 aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// ============================
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
        // Check if access disabled in CPTR_EL2
        if HaveVirtHostExt() && HCR_EL2.E2H == '1' then
            case CPTR_EL2.FPEN of
                when 'x0'  disabled = !(PSTATE.EL == EL1 && HCR_EL2.TGE == '1');
                when '01'  disabled = (PSTATE.EL == EL0 && HCR_EL2.TGE == '1');
                when '11'  disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    if HaveEL(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL3);

    return;
```

5.61 aarch64/exceptions/traps/AArch64.CheckForSMCUndefOrTrap

```
// AArch64.CheckForSMCUndefOrTrap()
// ================================
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUndefOrTrap(bits(16) imm)
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if !HaveEL(EL3) || PSTATE.EL == EL0 then
        UNDEFINED;
    route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

5.62 aarch64/exceptions/traps/AArch64.CheckForWFxTrap

```
// AArch64.CheckForWFxTrap()
// =========================
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFxTrap(bits(2) target_el, boolean is_wfe)
    assert HaveEL(target_el);

    case target_el of
        when EL1 trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2 trap = (if is_wfe then HCR_EL2.TWE else HCR_EL2.TWI) == '1';
        when EL3 trap = (if is_wfe then SCR_EL3.TWE else SCR_EL3.TWI) == '1';
    if trap then
        AArch64.WFxTrap(target_el, is_wfe);
```

5.63 aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// ===========================
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;

        exception = ExceptionSyndrome(Exception_IllegalState);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.64 aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =========================
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

5.65 aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// ==========================
// Trapped access to AArch64 system register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

5.66 aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// ==================================
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;
    case ec of
        when 0x0                                                     // Trapped access due to unknown
            ↪reason.
            exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7                                                     // Trapped access to SVE, Advance
            ↪SIMD&FP system register.
            exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
        when 0x18                                                    // Trapped access to system register
            ↪or system instruction.
            exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            exception.syndrome<21:20> = instr<20:19>;          // Op0
            exception.syndrome<19:17> = instr<7:5>;            // Op2
            exception.syndrome<16:14> = instr<18:16>;          // Op1
            exception.syndrome<13:10> = instr<15:12>;          // CRn
            exception.syndrome<9:5>   = instr<4:0>;            // Rt
            exception.syndrome<4:1>   = instr<11:8>;           // CRm
            exception.syndrome<0>     = instr<21>;             // Direction
        when 0x29                         // Trapped access to 64-bit System register which is part of
            ↪Capability functionality
            exception = ExceptionSyndrome(Exception_CapabilityAccess);
        when 0x2a                                                    // Trapped access to Capability
            ↪system register
            exception = ExceptionSyndrome(Exception_CapabilitySysRegTrap);
            instr = ThisInstr();
            exception.syndrome<21:20> = '1':instr<19>;         // Op0
            exception.syndrome<19:17> = instr<7:5>;            // Op2
            exception.syndrome<16:14> = instr<18:16>;          // Op1
            exception.syndrome<13:10> = instr<15:12>;          // CRn
            exception.syndrome<9:5>   = instr<4:0>;            // Rt
            exception.syndrome<4:1>   = instr<11:8>;           // CRm
            exception.syndrome<0>     = instr<20>;             // Direction
        otherwise
            Unreachable();

    return exception;
```

5.67 aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// ========================

AArch64.UndefinedFault()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
```

```
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

5.68 aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =================

AArch64.WFxTrap(bits(2) target_el, boolean is_wfe)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<0> = if is_wfe then '1' else '0';

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

5.69 aarch64/exceptions/traps/CapabilityAccessTrap

```
// CapabilityAccessTrap()
// ======================
// Trapped access to Capabilities to CPACR_EL1 or CPTR_EL2 or CPTR_EL3.

CapabilityAccessTrap(bits(2) target_el)

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_CapabilityAccess);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

    return;
```

5.70 aarch64/exceptions/traps/CheckCapabilitiesEnabled

```
// CheckCapabilitiesEnabled()
// ==========================
// Check against CPACR_EL1, CPTR_EL2 and CPTR_EL3 and trap if not enabled.

CheckCapabilitiesEnabled()
    if PSTATE.EL IN {EL0, EL1} then
        case CPACR_EL1.CEN of
            when 'x0'  disabled = TRUE;
            when '01'  disabled = PSTATE.EL == EL0;
            when '11'  disabled = FALSE;

        // Special case when CPACR_EL1.CEN does not cause traps
        if HaveEL(EL2) && !IsSecure() && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' then
            disabled = FALSE;

        if disabled then
            if HaveEL(EL2) && HCR_EL2.TGE == '1' then
                CapabilityAccessTrap(EL2);
            else
                CapabilityAccessTrap(EL1);

    // Also check against CPTR_EL2 and CPTR_EL3
    if HaveEL(EL2) && !IsSecure() then
        if HCR_EL2.E2H == '1' then
            case CPTR_EL2.CEN of
                when 'x0'  disabled = (PSTATE.EL IN {EL0, EL1, EL2});
                when '01'  disabled = (PSTATE.EL == EL0 && HCR_EL2.TGE == '1');
                when '11'  disabled = FALSE;
            if disabled then CapabilityAccessTrap(EL2);
        else
```

```
            if CPTR_EL2.TC == '1' then CapabilityAccessTrap(EL2);

    if HaveEL(EL3) then
        if CPTR_EL3.EC == '0' then CapabilityAccessTrap(EL3);

    return;
```

5.71 aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =========================
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

5.72 aarch64/exceptions/traps/IsAccessToCapabilitiesDisabledAtEL0

```
// IsAccessToCapabilitiesDisabledAtEL0()
// =====================================
// Check if access to capabilities is disabled at EL0

boolean IsAccessToCapabilitiesDisabledAtEL0()
    if IsAccessToCapabilitiesDisabledAtEL1() then
        return TRUE;
    elsif !(HaveEL(EL2) && !IsSecure() && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1') && CPACR_EL1.CEN ==
        ↪'01' then
        return TRUE;
    else
        return HaveEL(EL2) && !IsSecure() && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && CPTR_EL2.CEN ==
            ↪'01';
```

5.73 aarch64/exceptions/traps/IsAccessToCapabilitiesDisabledAtEL1

```
// IsAccessToCapabilitiesDisabledAtEL1()
// =====================================
// Check if access to capabilities is disabled at EL1

boolean IsAccessToCapabilitiesDisabledAtEL1()
    if IsAccessToCapabilitiesDisabledAtEL2() then
        return TRUE;
    else
        return !(HaveEL(EL2) && !IsSecure() && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1') && CPACR_EL1.CEN
            ↪== 'x0';
```

5.74 aarch64/exceptions/traps/IsAccessToCapabilitiesDisabledAtEL2

```
// IsAccessToCapabilitiesDisabledAtEL2()
// =====================================
// Check if access to capabilities is disabled at EL2

boolean IsAccessToCapabilitiesDisabledAtEL2()
    if IsAccessToCapabilitiesDisabledAtEL3() then
        return TRUE;
    elsif HaveEL(EL2) && !IsSecure() then
        return (HCR_EL2.E2H == '1' && CPTR_EL2.CEN == 'x0') || (HCR_EL2.E2H == '0' && CPTR_EL2.TC == '1');
    else
        return FALSE;
```

5.75 aarch64/exceptions/traps/IsAccessToCapabilitiesDisabledAtEL3

```
// IsAccessToCapabilitiesDisabledAtEL3()
// =====================================
// Check if access to capabilities is disabled at EL3

```

```
boolean IsAccessToCapabilitiesDisabledAtEL3()
    return HaveEL(EL3) && CPTR_EL3.EC == '0';
```

5.76 aarch64/exceptions/traps/IsAccessToCapabilitiesEnabledAtEL

```
// IsAccessToCapabilitiesEnabledAtEL()
// ===================================
// Check if access to capabilities is enabled at a particular EL

boolean IsAccessToCapabilitiesEnabledAtEL(bits(2) el)
    case el of
        when EL3 return !IsAccessToCapabilitiesDisabledAtEL3();
        when EL2 return !IsAccessToCapabilitiesDisabledAtEL2();
        when EL1 return !IsAccessToCapabilitiesDisabledAtEL1();
        when EL0 return !IsAccessToCapabilitiesDisabledAtEL0();
```

5.77 aarch64/exceptions/traps/IsInC64

```
// IsInC64()
// =========
// Return whether the current instruction set is C64

boolean IsInC64()
    return PSTATE.C64 == '1';
```

5.78 aarch64/exceptions/traps/IsTagSettingDisabled

```
// IsTagSettingDisabled()
// ======================
// Check if instructions that explicitly set capability tags are disabled

boolean IsTagSettingDisabled()

    if PSTATE.EL == EL0 || PSTATE.EL == EL1 then
        if (EL2Enabled() && !ELUsingAArch32(EL2) && CHCR_EL2.SETTAG == '1') then
            return TRUE;
        elsif (HaveEL(EL3) && !ELUsingAArch32(EL3) && CSCR_EL3.SETTAG == '1') then
            return TRUE;
    elsif PSTATE.EL == EL2 then
        if HaveEL(EL3) && !ELUsingAArch32(EL3) && CSCR_EL3.SETTAG == '1' then
            return TRUE;
    return FALSE;
```

5.79 aarch64/exceptions/traps/TargetELForCapabilityExceptions

```
// TargetELForCapabilityExceptions()
// =================================
// Return the target exception level to which capability-related exceptions are routed

bits(2) TargetELForCapabilityExceptions()
    bits(2) lowest_el;
    if HighestEL() == EL1 || !IsAccessToCapabilitiesDisabledAtEL1() then
        if EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            lowest_el = EL2;
        else
            lowest_el = EL1;
    elsif HighestEL() == EL2 || (!IsAccessToCapabilitiesDisabledAtEL2() && EL2Enabled()) then
        lowest_el = EL2;
    else
        lowest_el = EL3;

    if UInt(lowest_el) < UInt(PSTATE.EL) then
        return PSTATE.EL;
    else
        return lowest_el;
```

5.80 aarch64/functions/aborts/AArch64.CreateFaultRecord

```
// AArch64.CreateFaultRecord()
// ===========================

FaultRecord AArch64.CreateFaultRecord(Fault statuscode, bits(48) ipaddress,
                                      integer level, AccType acctype, boolean write, bit extflag,
                                      bits(2) errortype, boolean secondstage, boolean s2fs1walk)

    FaultRecord fault;
    fault.statuscode = statuscode;
    fault.domain = bits(4) UNKNOWN;         // Not used from AArch64
    fault.debugmoe = bits(4) UNKNOWN;       // Not used from AArch64
    fault.errortype = errortype;
    fault.ipaddress = ipaddress;
    fault.level = level;
    fault.acctype = acctype;
    fault.write = write;
    fault.extflag = extflag;
    fault.secondstage = secondstage;
    fault.s2fs1walk = s2fs1walk;

    return fault;
```

5.81 aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =======================
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception Level using AArch64.

bits(25) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    bits(25) iss = Zeros();
    if HaveRASExt() && IsExternalSyncAbort(fault) then iss<12:11> = fault.errortype; // SET
    if d_side then
        if IsSecondStage(fault) && !fault.s2fs1walk then iss<24:14> = LSInstructionSyndrome();
        if fault.acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_IC, AccType_AT} then
            iss<8> = '1';  iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fs1walk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return iss;
```

5.82 aarch64/functions/exclusive/AArch64.ExclusiveMonitorsPass

```
// AArch64.ExclusiveMonitorsPass()
// ===============================

// Return TRUE if the Exclusives monitors for the current PE include all of the addresses
// associated with the virtual address region of size bytes starting at address.
// The immediately following memory write must be to the same addresses.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // It is IMPLEMENTATION DEFINED whether the detection of memory aborts happens
    // before or after the check on the local Exclusives monitor. As a result a failure
    // of the local monitor can occur on some implementations even if the memory
    // access would give an memory abort.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = (address == Align(address, size));
    if !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareable then
            passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

5.83 aarch64/functions/exclusive/AArch64.IsExclusiveVA

```
// An optional IMPLEMENTATION DEFINED test for an exclusive access to a virtual
// address region of size bytes starting at address.
//
// It is permitted (but not required) for this function to return FALSE and
// cause a store exclusive to fail if the virtual address region is not
// totally included within the region recorded by MarkExclusiveVA().
//
// It is always safe to return TRUE which will check the physical address only.
boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

5.84 aarch64/functions/exclusive/AArch64.MarkExclusiveVA

```
// Optionally record an exclusive access to the virtual address region of size bytes
// starting at address for processorid.
AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```

5.85 aarch64/functions/exclusive/AArch64.SetExclusiveMonitors

```
// AArch64.SetExclusiveMonitors()
// ==============================

// Sets the Exclusives monitors for the current PE to record the addresses associated
// with the virtual address region of size bytes starting at address.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)

    acctype = AccType_ATOMIC;
    iswrite = FALSE;
    aligned = (address == Align(address, size));
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareable then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

5.86 aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// ==================

bits(N) FPRSqrtStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

5.87 aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// ==================

bits(N) FPRecipStepFused(bits(N) op1, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    op1 = FPNeg(op1);
    (type1,sign1,value1) = FPUnpack(op1, FPCR);
    (type2,sign2,value2) = FPUnpack(op2, FPCR);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, FPCR);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0');
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if FPRoundingMode(FPCR) == FPRounding_NEGINF then '1' else '0';
                result = FPZero(sign);
            else
                result = FPRound(result_value, FPCR);
    return result;
```

5.88 aarch64/functions/memory/AArch64.CheckAlignment

```
// AArch64.CheckAlignment()
// ========================

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                               boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType_ATOMIC, AccType_ATOMICRW, AccType_ORDEREDATOMIC,
        ↪AccType_ORDEREDATOMICRW };
    ordered = acctype IN { AccType_ORDERED, AccType_ORDEREDRW, AccType_LIMITEDORDERED,
        ↪AccType_ORDEREDATOMIC, AccType_ORDEREDATOMICRW };
    vector  = acctype == AccType_VEC;
    check = (atomic || ordered || SCTLR[].A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

5.89 aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// ================================================
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    value = _Mem[memaddrdesc, size, accdesc];
    return value;

// AArch64.MemSingle[] - assignment (write) form
// =============================================
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean wasaligned] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);
    _Mem[memaddrdesc, size, accdesc] = value;
    return;
```

5.90 aarch64/functions/memory/AArch64.TaggedMemSingle

```
// AArch64.TaggedMemSingle[] - non-assignment (read) form
// ======================================================
// Perform an atomic, little-endian read of 'size' bytes with capability tags.

(bits(size DIV 16), bits(size*8)) AArch64.TaggedMemSingle(bits(64) address, integer size, AccType acctype, boolean wasaligned)
    assert size IN {16, 32};
    assert address == Align(address, 16);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    bits(size DIV 16) tags;
    iswrite = FALSE;

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Device memory locations marked as faulting loads of valid capabilities
    // will fault and will not read the memory location.
    if memaddrdesc.memattrs.memtype == MemType_Device then
        CheckLoadTagsPermission(memaddrdesc, acctype);

    accdesc = CreateAccessDescriptor(acctype);

    // Memory array access
    if memaddrdesc.memattrs.readtagzero then
        value = _ReadMem(memaddrdesc, size, accdesc);
        tags = Zeros(size DIV 16);
    else
        (tags, value) = _ReadTaggedMem(memaddrdesc, size, accdesc);

        if tags != Zeros(size DIV 16) then
            CheckLoadTagsPermission(memaddrdesc, acctype);

    return (tags, value);

// AArch64.TaggedMemSingle[] - assignment (write) form
// ====================================================
// Perform an atomic, little-endian write of 'size' bytes with capability tags.

AArch64.TaggedMemSingle(bits(64) address, integer size, AccType acctype, boolean wasaligned, bits(size DIV 16) tags, bits(size*8) value)
    assert size IN {16, 32};
    assert address == Align(address, 16);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    // MMU or MPU
    boolean valid_cap = (tags != Zeros(size DIV 16));
    memaddrdesc = AArch64.TranslateAddressWithTag(address, acctype, iswrite, wasaligned, size, valid_cap);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    accdesc = CreateAccessDescriptor(acctype);

    if tags != Zeros(size DIV 16) then
        CheckStoreTagsPermission(memaddrdesc, acctype);

    // Memory array access
    _WriteTaggedMem(memaddrdesc, size, accdesc, tags, value);
    return;
```

5.91 aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

```
// AArch64.TranslateAddressForAtomicAccess()
// =========================================
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned = AArch64.CheckAlignment(address, size, AccType_ATOMICRW, iswrite);

    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType_ATOMICRW, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    return memaddrdesc;
```

5.92 aarch64/functions/memory/CapabilityTag

```
// CapabilityTag() - non-assignment (read) form
// ============================================
// Reads a single capability tag from memory

bits(1) AArch64.CapabilityTag(bits(64) address, AccType acctype)

    boolean iswrite = FALSE;
    CheckCapabilityAlignment(address, acctype, iswrite);

    AddressDescriptor memaddrdesc;

    // MMU or MPU
    boolean wasaligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, wasaligned, CAPABILITY_DBYTES DIV 8);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Device memory locations marked as faulting loads of valid capabilities
    // will fault and will not read the memory location.
    if memaddrdesc.memattrs.memtype == MemType_Device then
        CheckLoadTagsPermission(memaddrdesc, acctype);

    accdesc = CreateAccessDescriptor(acctype);

    bits(1) tag;
    if memaddrdesc.memattrs.readtagzero then
        tag = '0';
    else
        bits(48) paddress = memaddrdesc.paddress.address;

        assert paddress == Align(paddress, CAPABILITY_DBYTES);
        tag = _ReadTags(memaddrdesc, 1, accdesc);

        if tag == '1' then
            CheckLoadTagsPermission(memaddrdesc, acctype);

    return tag;

// CapabilityTag() - assignment (write) form
// =========================================
// Writes a single capability tag from memory

AArch64.CapabilityTag[bits(64) address, AccType acctype] = bits(1) tag

    boolean iswrite = TRUE;
    CheckCapabilityAlignment(address, acctype, iswrite);

    AddressDescriptor memaddrdesc;
    boolean wasaligned = TRUE;

    // MMU or MPU
    boolean valid_cap = (tag == '1');
    memaddrdesc = AArch64.TranslateAddressWithTag(address, acctype, iswrite, wasaligned,
        ↪CAPABILITY_DBYTES, valid_cap);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareable then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), CAPABILITY_DBYTES);

    accdesc = CreateAccessDescriptor(acctype);

    bits(48) paddress = memaddrdesc.paddress.address;

    assert paddress == Align(paddress, CAPABILITY_DBYTES);

    if tag == '1' then
```

```
        CheckStoreTagsPermission(memaddrdesc, acctype);

    _WriteTags(memaddrdesc, 1, tag, accdesc);

    return;
```

5.93 aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// ==================
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    bits(64) sp = SP[];
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR[].SA0 != '0');
    else
        stack_align_check = (SCTLR[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

5.94 aarch64/functions/memory/Mem

```
constant integer CAPABILITY_DBYTES = 16;
constant integer LOG2_CAPABILITY_DBYTES = 4;

// Mem[] - non-assignment (read) form
// ==================================
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccType acctype]
    assert size IN {1, 2, 4, 8, 16};
    bits(size*8) value;
    boolean iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        value<63:0>   = AArch64.MemSingle[address,   8, acctype, aligned];
        value<127:64> = AArch64.MemSingle[address+8, 8, acctype, aligned];
    else
        value = AArch64.MemSingle[address, size, acctype, aligned];

    if BigEndian() then
        value = BigEndianReverse(value);
    return value;

// Mem[] - assignment (write) form
// ===============================
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

```

```
Mem[bits(64) address, integer size, AccType acctype] = bits(size*8) value
    boolean iswrite = TRUE;

    if BigEndian() then
        value = BigEndianReverse(value);

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        atomic = aligned;
    else
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);

    if !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        if !aligned then
            c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
            assert c IN {Constraint_FAULT, Constraint_NONE};
            if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        AArch64.MemSingle[address,   8, acctype, aligned] = value<63:0>;
        AArch64.MemSingle[address+8, 8, acctype, aligned] = value<127:64>;
    else
        AArch64.MemSingle[address, size, acctype, aligned] = value;
    return;

CheckCapabilityAlignment(bits(64) address, AccType acctype, boolean iswrite)

    if (address != Align(address, CAPABILITY_DBYTES)) then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

CheckCapabilityStorePairAlignment(bits(64) address, AccType acctype, boolean iswrite)

    boolean atomic = (acctype == AccType_ATOMIC) || (acctype == AccType_ORDEREDATOMIC);
    integer size = if atomic then CAPABILITY_DBYTES*2 else CAPABILITY_DBYTES;

    if (address != Align(address, size)) then
        secondstage = FALSE;
        AArch64.Abort(address, AArch64.AlignmentFault(acctype, iswrite, secondstage));

Capability MemC[bits(64) address, AccType acctype]
    boolean iswrite = FALSE;
    bits(8*CAPABILITY_DBYTES) data;
    bits(CAPABILITY_DBYTES DIV 16) tag;
    Capability cap;

    CheckCapabilityAlignment(address, acctype, iswrite);
    (tag, data) = AArch64.TaggedMemSingle(address, CAPABILITY_DBYTES, acctype, TRUE);

    cap = CapabilityFromData(CAPABILITY_DBITS, tag<0>, data<CAPABILITY_DBITS-1:0>);

    return cap;

MemC[bits(64) address, AccType acctype] = Capability value
    boolean iswrite = TRUE;
    bits(CAPABILITY_DBITS) data;
    bits(CAPABILITY_DBYTES DIV 16) tag;

    (tag<0>, data) = DataFromCapability(CAPABILITY_DBITS, value);

    CheckCapabilityAlignment(address, acctype, iswrite);
    AArch64.TaggedMemSingle(address, CAPABILITY_DBYTES, acctype, TRUE, tag, data<CAPABILITY_DBYTES*8-1:0>);

// At the time of writing, array form doesn't support tuple assignment

(Capability, Capability) MemCP(bits(64) address, AccType acctype)
    boolean iswrite = FALSE;
    integer size = CAPABILITY_DBYTES*2;
    bits(8*size) data;
    bits(size DIV 16) tags;
    Capability cap1;
    Capability cap2;

    CheckCapabilityAlignment(address, acctype, iswrite);
    (tags, data) = AArch64.TaggedMemSingle(address, size, acctype, TRUE);

    bits(CAPABILITY_DBITS) data1 = data<CAPABILITY_DBITS-1:0>;
    bits(CAPABILITY_DBITS) data2 = data<(CAPABILITY_DBITS*2)-1:CAPABILITY_DBITS>;
    cap1 = CapabilityFromData(CAPABILITY_DBITS, tags<0>, data1);
    cap2 = CapabilityFromData(CAPABILITY_DBITS, tags<1>, data2);

    return (cap1, cap2);

MemCP(bits(64) address, AccType acctype, Capability value1, Capability value2)
    boolean iswrite = TRUE;
    integer size = CAPABILITY_DBYTES*2;
    bits(size DIV 16) tags;
    bits(8*size) data;

    (tags<0>, data<CAPABILITY_DBITS-1:0>)                        = DataFromCapability(CAPABILITY_DBITS, value1);
    (tags<1>, data<(CAPABILITY_DBITS*2)-1:CAPABILITY_DBITS>) = DataFromCapability(CAPABILITY_DBITS, value2);

    CheckCapabilityStorePairAlignment(address, acctype, iswrite);
    AArch64.TaggedMemSingle(address, size, acctype, TRUE, tags, data);

constant integer CAPABILITY_DBITS = CAPABILITY_DBYTES * 8;
```

5.95 aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// ===========
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(VirtualAddress base, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacctype)
    bits(64) address = VAddress(base);
    VACheckAddress(base, address, size DIV 8, CAP_PERM_LOAD, ldacctype);
    VACheckAddress(base, address, size DIV 8, CAP_PERM_STORE, stacctype);
    bits(size) newvalue;
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc];
    if BigEndian() then
        oldvalue = BigEndianReverse(oldvalue);

    case op of
        when MemAtomicOp_ADD   newvalue = oldvalue + value;
        when MemAtomicOp_BIC   newvalue = oldvalue AND NOT(value);
        when MemAtomicOp_EOR   newvalue = oldvalue EOR value;
        when MemAtomicOp_ORR   newvalue = oldvalue OR value;
        when MemAtomicOp_SMAX  newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
        when MemAtomicOp_SMIN  newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
        when MemAtomicOp_UMAX  newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
        when MemAtomicOp_UMIN  newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
        when MemAtomicOp_SWP   newvalue = value;

    if BigEndian() then
        newvalue = BigEndianReverse(newvalue);
    _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;

    // Load operations return the old (pre-operation) value
    return oldvalue;
```

5.96 aarch64/functions/memory/MemAtomicC

```
// MemAtomicC()
// ============
// Performs load capability and store capability memory operations for a given virtual address.

Capability MemAtomicC(bits(64) address, MemAtomicOp op, Capability value, AccType ldacctype, AccType stacctype)
```

```

    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, CAPABILITY_DBYTES*8);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.

    // Check of SC
    integer size = CAPABILITY_DBYTES;
    // This is only used for Cap_SWP instruction in Morello
    assert(op == MemAtomicOp_SWP);
    bits(8*size) newdata;
    bits(size DIV 16) newtag;
    (newtag<0>, newdata) = DataFromCapability(8*size, value);
    if newtag != Zeros(size DIV 16) then
        CheckStoreTagsPermission(memaddrdesc, stacctype);

    // Device memory locations marked as faulting loads of valid capabilities
    // will fault and will not read the memory location.
    if memaddrdesc.memattrs.memtype == MemType_Device then
        CheckLoadTagsPermission(memaddrdesc, ldacctype);

    // Memory array access
    bits(8 * size) olddata;
    bits(size DIV 16) oldtag;
    if memaddrdesc.memattrs.readtagzero then
        olddata = _ReadMem(memaddrdesc, size, ldaccdesc);
        oldtag = Zeros(size DIV 16);
    else
        (oldtag, olddata) = _ReadTaggedMem(memaddrdesc, size, ldaccdesc);

        // Check of LC
        if oldtag != Zeros(size DIV 16) then
            CheckLoadTagsPermission(memaddrdesc, ldacctype);

    _WriteTaggedMem(memaddrdesc, size, staccdesc, newtag, newdata);

    // Load operations return the old (pre-operation) capability value
    return CapabilityFromData(CAPABILITY_DBITS, oldtag<0>, olddata<CAPABILITY_DBITS-1:0>);
```

5.97 aarch64/functions/memory/MemAtomicCompareAndSwap

```
// MemAtomicCompareAndSwap()
// =========================
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(VirtualAddress base, bits(size) expectedvalue,
                                   bits(size) newvalue, AccType ldacctype, AccType stacctype)
    bits(64) address = VAddress(base);
    VACheckAddress(base, address, size DIV 8, CAP_PERM_LOAD, ldacctype);
    VACheckAddress(base, address, size DIV 8, CAP_PERM_STORE, stacctype);
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    oldvalue = _Mem[memaddrdesc, size DIV 8, ldaccdesc];
    if BigEndian() then
        oldvalue = BigEndianReverse(oldvalue);

    if oldvalue == expectedvalue then
        if BigEndian() then
            newvalue = BigEndianReverse(newvalue);
        _Mem[memaddrdesc, size DIV 8, staccdesc] = newvalue;
    return oldvalue;
```

5.98 aarch64/functions/memory/MemAtomicCompareAndSwapC

```
// MemAtomicCompareAndSwapC()
// ==========================
// Compares the Capability stored at the passed-in memory address against the passed-in expected
// Capability. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

Capability MemAtomicCompareAndSwapC(VirtualAddress vaddr, bits(64) address, Capability expectedcap,
                                    Capability newcap, AccType ldacctype, AccType stacctype)
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, CAPABILITY_DBYTES*8);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // Check of SC
    integer size = CAPABILITY_DBYTES;
    bits(8*size) newdata;
    bits(size DIV 16) newtag;
    (newtag<0>, newdata) = DataFromCapability(8*size, newcap);
    if newtag != Zeros(size DIV 16) then
        CheckStoreTagsPermission(memaddrdesc, stacctype);

    // Device memory locations marked as faulting loads of valid capabilities
    // will fault and will not read the memory location.
    if memaddrdesc.memattrs.memtype == MemType_Device then
        CheckLoadTagsPermission(memaddrdesc, ldacctype);

    // Memory array access
    bits(8 * size) olddata;
    bits(size DIV 16) oldtag;
    if memaddrdesc.memattrs.readtagzero then
        olddata = _ReadMem(memaddrdesc, size, ldaccdesc);
        oldtag = Zeros(size DIV 16);
    else
        (oldtag, olddata) = _ReadTaggedMem(memaddrdesc, size, ldaccdesc);

        // Check of LC
        if oldtag != Zeros(size DIV 16) then
            CheckLoadTagsPermission(memaddrdesc, ldacctype);

    Capability oldcap = CapabilityFromData(CAPABILITY_DBITS, oldtag<0>, olddata<CAPABILITY_DBITS-1:0>);
    oldcap = CapSquashPostLoadCap(oldcap, vaddr);

    if CapIsEqual(oldcap,expectedcap) then
        _WriteTaggedMem(memaddrdesc, size, staccdesc, newtag, newdata);

    return oldcap;
```

5.99 aarch64/functions/ras/AArch64.ESBOperation

```
// AArch64.ESBOperation()
// ======================
// Perform the AArch64 ESB operation, either for ESB executed in AArch64 state, or for
// ESB in AArch32 state when SError interrupts are routed to an Exception level using
// AArch64

AArch64.ESBOperation()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.AMO == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = PSTATE.A == '1';
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending
    if IsPhysicalSErrorPending() && masked then
        implicit_esb = FALSE;
        syndrome = AArch64.PhysicalSErrorSyndrome(implicit_esb);
        DISR_EL1 = AArch64.ReportDeferredSError(syndrome)<31:0>;
        ClearPendingPhysicalSError();               // Set ISR_EL1.A to 0

    return;
```

5.100 aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

5.101 aarch64/functions/ras/AArch64.ReportDeferredSError

```
// AArch64.ReportDeferredSError()
// ==============================
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31>    = '1';               // A
    target<24>    = syndrome<24>;      // IDS
    target<23:0> = syndrome<23:0>;    // ISS
    return target;
```

5.102 aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =======================
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>)<31:0>;
        HCR_EL2.VSE = '0';                          // Clear pending virtual SError

    return;
```

5.103 aarch64/functions/registers/AArch64.MaybeZeroRegisterUppers

```
// AArch64.MaybeZeroRegisterUppers()
// =================================
// On taking an exception to  AArch64 from AArch32, it is CONSTRAINED UNPREDICTABLE whether the top
// 32 bits of registers visible at any lower Exception level using AArch32 are set to zero.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();         // Always called from AArch32 state before entering AArch64 state

    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0;  last = 14;  include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0;  last = 30;  include_R15 = FALSE;
    else
        first = 0;  last = 30;  include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable_ZEROUPPER) then
            _R[n]<63:32> = Zeros();

    return;
```

5.104 aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// ===============================

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        C[i] = CapNull();
    return;
```

5.105 aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// ==============================

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = bits(128) UNKNOWN;

    return;
```

5.106 aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// ===============================

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(129) UNKNOWN;
    SP_EL1 = bits(129) UNKNOWN;
    ELR_EL1  = bits(129) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(129) UNKNOWN;
        ELR_EL2  = bits(129) UNKNOWN;
        SPSR_EL2 = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SP_EL3 = bits(129) UNKNOWN;
        ELR_EL3  = bits(129) UNKNOWN;
        SPSR_EL3 = bits(32) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq = bits(32) UNKNOWN;
        SPSR_irq = bits(32) UNKNOWN;
        SPSR_abt = bits(32) UNKNOWN;
        SPSR_und = bits(32) UNKNOWN;

    // External debug special registers
    DSPSR_EL0 = bits(32) UNKNOWN;
    CDLR_EL0  = bits(129) UNKNOWN;

    return;
```

5.107 aarch64/functions/registers/AArch64.ResetSystemRegisters

```
AArch64.ResetSystemRegisters(boolean cold_reset);
```

5.108 aarch64/functions/registers/C

```
// C[] - assignment form
// =====================
// Write to capability register from a 129-bit value.

C[integer n] = Capability value
    assert n >= 0 && n <= 31;
    if n != 31 then
```

```
        _R[n] = ZeroExtend(value);
    return;

// C[] - non-assignment form
// =========================
// Read from capabiltiy register with implicit slice of 129 bits.

Capability C[integer n]
    assert n >= 0 && n <= 31;
    if n != 31 then
        return _R[n]<128:0>;
    else
        return CapNull();
```

5.109 aarch64/functions/registers/CSP

```
// CSP[] - assignment form
// =======================
// Write to stack pointer from a capability value.

CSP[] = Capability value
    if IsInRestricted() then
        RSP_EL0 = value;
    elsif PSTATE.SP == '0' then
        SP_EL0 = value;
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = value;
            when EL1  SP_EL1 = value;
            when EL2  SP_EL2 = value;
            when EL3  SP_EL3 = value;
    return;

// CSP[] - non-assignment form
// ===========================
// Read capability stack pointer

Capability CSP[]
    if IsInRestricted() then
        return RSP_EL0;
    elsif PSTATE.SP == '0' then
        return SP_EL0;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0;
            when EL1  return SP_EL1;
            when EL2  return SP_EL2;
            when EL3  return SP_EL3;
```

5.110 aarch64/functions/registers/CapIsSystemAccessEnabled

```
// CapIsSystemAccessEnabled()
// ==========================
// Returns whether access to system resources is enabled

boolean CapIsSystemAccessEnabled()
    if Halted() then
        return TRUE;
    else
        return CapIsSystemAccessPermitted(PCC[]);
```

5.111 aarch64/functions/registers/Capability

```
type Capability;
```

5.112 aarch64/functions/registers/DDC

```
// DDC[] - assignment form
// ======================
// Write to default data capability

DDC[] = Capability value
    DDC = value;
    if IsInRestricted() then
        RDDC_EL0 = value;
    elsif PSTATE.SP == '0' then
        DDC_EL0 = value;
    else
        case PSTATE.EL of
            when EL0  DDC_EL0 = value;
            when EL1  DDC_EL1 = value;
            when EL2  DDC_EL2 = value;
            when EL3  DDC_EL3 = value;

// DDC[] - non-assignment form
// ===========================
// Read default data capability

Capability DDC[]
    if IsInRestricted() then
        return RDDC_EL0;
    elsif PSTATE.SP == '0' then
        return DDC_EL0;
    else
        case PSTATE.EL of
            when EL0  return DDC_EL0;
            when EL1  return DDC_EL1;
            when EL2  return DDC_EL2;
            when EL3  return DDC_EL3;
```

5.113 aarch64/functions/registers/IsInRestricted

```
// IsInRestricted()
// ================
// Returns whether the PE is in Restricted state

boolean IsInRestricted()
    if Halted() then
        return FALSE;
    else
        return !CapIsExecutive(PCC[]);
```

5.114 aarch64/functions/registers/PC

```
// PC - non-assignment form
// ========================
// Read program counter.

bits(64) PC[]
    return CapGetValue(PCC);

VirtualAddress BaseReg[integer n, boolean is_prefetch]
    if !IsInC64() then
        bits(64) address;
        if n == 31 then
            if !is_prefetch then
                CheckSPAlignment();
            address = SP[];
        else
            address = X[n];
        return VAFromBits64(address);
    else
        Capability address;
        if n == 31 then
            if !is_prefetch then
                CheckSPAlignment();
            address = CSP[];
        else
            address = C[n];
        return VAFromCapability(address);

VirtualAddress AltBaseReg[integer n, boolean is_prefetch]
```

```
    if !IsInC64() then
        Capability address;
        if n == 31 then
            if !is_prefetch then
                CheckSPAlignment();
            address = CSP[];
        else
            address = C[n];
        return VAFromCapability(address);
    else
        bits(64) address;
        if n == 31 then
            if !is_prefetch then
                CheckSPAlignment();
            address = SP[];
        else
            address = X[n];
        return VAFromBits64(address);

VirtualAddress BaseReg[integer n]
    return BaseReg[n, FALSE];

VirtualAddress AltBaseReg[integer n]
    return AltBaseReg[n, FALSE];

BaseReg[integer n] = VirtualAddress address
    if !IsInC64() then
        if n == 31 then
            SP[] = VAToBits64(address);
        else
            X[n] = VAToBits64(address);
    else
        if n == 31 then
            CSP[] = VAToCapability(address);
        else
            C[n] = VAToCapability(address);

AltBaseReg[integer n] = VirtualAddress address
    if !IsInC64() then
        if n == 31 then
            CSP[] = VAToCapability(address);
        else
            C[n] = VAToCapability(address);
    else
        if n == 31 then
            SP[] = VAToBits64(address);
        else
            X[n] = VAToBits64(address);
```

5.115 aarch64/functions/registers/PCC

```
// PCC[] - assignment form
// =======================
// Write to program counter capability

PCC[] = Capability value
    PCC = ZeroExtend(value);

// PCC[] - non-assignment form
// ===========================
// Read program counter capability

Capability PCC[]
    return PCC;
```

5.116 aarch64/functions/registers/SP

```
// SP[] - assignment form
// ======================
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
    assert width IN {32,64};
    if IsInRestricted() then
        RSP_EL0 = ZeroExtend(value);
    elsif PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value);
    else
        case PSTATE.EL of
            when EL0  SP_EL0 = ZeroExtend(value);
            when EL1  SP_EL1 = ZeroExtend(value);
            when EL2  SP_EL2 = ZeroExtend(value);
            when EL3  SP_EL3 = ZeroExtend(value);

    return;

// SP[] - non-assignment form
// ==========================
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
    assert width IN {8,16,32,64};
    if IsInRestricted() then
        return RSP_EL0<width-1:0>;
    elsif PSTATE.SP == '0' then
        return SP_EL0<width-1:0>;
    else
        case PSTATE.EL of
            when EL0  return SP_EL0<width-1:0>;
            when EL1  return SP_EL1<width-1:0>;
            when EL2  return SP_EL2<width-1:0>;
            when EL3  return SP_EL3<width-1:0>;
```

5.117 aarch64/functions/registers/V

```
// V[] - assignment form
// =====================
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    _V[n] = ZeroExtend(value);
    return;

// V[] - non-assignment form
// =========================
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64,128};
    return _V[n]<width-1:0>;
```

5.118 aarch64/functions/registers/VirtualAddress

```
type VirtualAddress is (
    VirtualAddressType vatype,
    Capability base,
    bits(64) offset,
)
```

5.119 aarch64/functions/registers/VirtualAddressType

```
enumeration VirtualAddressType { VA_Bits64, VA_Capability };
```

5.120 aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =============================
// Reads a 128-bit SIMD&FP register in up to two parts:
```

```
//  part 0 returns the bottom 8, 16, 32 or 64 bits of a value held in the register;
//  part 1 returns the top half of the bottom 64 bits or the top half of the 128-bit
//  value held in the register.

bits(width) Vpart[integer n, integer part]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        return _V[n]<width-1:0>;
    else
        assert width IN {32,64};
        return _V[n]<(width * 2)-1:width>;

// Vpart[] - assignment form
// =========================
// Writes a 128-bit SIMD&FP register in up to two parts:
//  part 0 zero extends a 8, 16, 32, or 64-bit value to fill the whole register;
//  part 1 inserts a 64-bit value into the top half of the register.

Vpart[integer n, integer part] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width IN {8,16,32,64};
        _V[n] = ZeroExtend(value);
    else
        assert width == 64;
        _V[n]<(width * 2)-1:width> = value<width-1:0>;
```

5.121 aarch64/functions/registers/X

```
// X[] - assignment form
// =====================
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - non-assignment form
// =========================
// Read from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

5.122 aarch64/functions/sysregisters/CCTLR

```
// CCTLR[] - non-assignment form
// =============================

CCTLRType CCTLR[bits(2) el]
    bits(32) r;
    case el of
        when EL0  r = CCTLR_EL0;
        when EL1  r = CCTLR_EL1;
        when EL2  r = CCTLR_EL2;
        when EL3  r = CCTLR_EL3;
        otherwise Unreachable();
    return r;

// CCTLR[] - non-assignment form
// =============================

CCTLRType CCTLR[]
    return CCTLR[PSTATE.EL];
```

5.123 aarch64/functions/sysregisters/CELR

```
// CELR[] - non-assignment form
// ============================

Capability CELR[bits(2) el]
    Capability r;
    case el of
        when EL1  r = ELR_EL1;
        when EL2  r = ELR_EL2;
        when EL3  r = ELR_EL3;
        otherwise Unreachable();
    return r;

// CELR[] - assignment form
// ========================

CELR[bits(2) el] = Capability value
    case el of
        when EL1  ELR_EL1 = value;
        when EL2  ELR_EL2 = value;
        when EL3  ELR_EL3 = value;
        otherwise Unreachable();
    return;

// CELR[] - non-assignment form
// ============================

Capability CELR[]
    return CELR[PSTATE.EL];

// CELR[] - assignment form
// ========================

CELR[] = Capability value
    CELR[PSTATE.EL] = value;
    return;
```

5.124 aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// ===============================

CNTKCTLType CNTKCTL[]
    bits(32) r;
    if IsInHost() then
        r = CNTHCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;
```

5.125 aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

5.126 aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =============================

CPACRType CPACR[]
    bits(32) r;
    if IsInHost() then
        r = CPTR_EL2;
        return r;
    r = CPACR_EL1;
    return r;
```

5.127 aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

5.128 aarch64/functions/sysregisters/CVBAR

```
// CVBAR[] - non-assignment form
// =============================

Capability CVBAR[bits(2) regime]
    Capability r;
    case regime of
        when EL1  r = VBAR_EL1;
        when EL2  r = VBAR_EL2;
        when EL3  r = VBAR_EL3;
        otherwise Unreachable();
    return r;

// CVBAR[] - non-assignment form
// =============================

Capability CVBAR[]
    return CVBAR[PSTATE.EL];
```

5.129 aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// ===========================

bits(64) ELR[bits(2) el]
    bits(64) r;
    case el of
        when EL1  r = ELR_EL1<63:0>;
        when EL2  r = ELR_EL2<63:0>;
        when EL3  r = ELR_EL3<63:0>;
        otherwise Unreachable();
    return r;

// ELR[] - non-assignment form
// ===========================

bits(64) ELR[]
    assert PSTATE.EL != EL0;
    return ELR[PSTATE.EL];

// ELR[] - assignment form
// =======================

ELR[bits(2) el] = bits(64) value
    bits(64) r = value;
    case el of
        when EL1
            ELR_EL1 = ZeroExtend(r);
        when EL2
            ELR_EL2 = ZeroExtend(r);
        when EL3
            ELR_EL3 = ZeroExtend(r);
        otherwise Unreachable();
    return;

// ELR[] - assignment form
// =======================

ELR[] = bits(64) value
    assert PSTATE.EL != EL0;
    ELR[PSTATE.EL] = value;
    return;
```

5.130 aarch64/functions/sysregisters/ESR

```
type CCTLRType;

// ESR[] - non-assignment form
// ===========================

ESRType ESR[bits(2) regime]
    bits(32) r;
    case regime of
        when EL1  r = ESR_EL1;
        when EL2  r = ESR_EL2;
        when EL3  r = ESR_EL3;
        otherwise Unreachable();
    return r;

// ESR[] - non-assignment form
// ===========================

ESRType ESR[]
    return ESR[S1TranslationRegime()];

// ESR[] - assignment form
// =======================

ESR[bits(2) regime] = ESRType value
    bits(32) r = value;
    case regime of
        when EL1  ESR_EL1 = r;
        when EL2  ESR_EL2 = r;
        when EL3  ESR_EL3 = r;
        otherwise Unreachable();
    return;

// ESR[] - assignment form
// =======================

ESR[] = ESRType value
    ESR[S1TranslationRegime()] = value;
```

5.131 aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

5.132 aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// ===========================

bits(64) FAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = FAR_EL1;
        when EL2  r = FAR_EL2;
        when EL3  r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR[] - non-assignment form
// ===========================

bits(64) FAR[]
    return FAR[S1TranslationRegime()];

// FAR[] - assignment form
// =======================

FAR[bits(2) regime] = bits(64) value
    bits(64) r = value;
    case regime of
        when EL1  FAR_EL1 = r;
        when EL2  FAR_EL2 = r;
        when EL3  FAR_EL3 = r;
        otherwise Unreachable();
    return;

// FAR[] - assignment form
// =======================

FAR[] = bits(64) value
    FAR[S1TranslationRegime()] = value;
    return;
```

5.133 aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = MAIR_EL1;
        when EL2  r = MAIR_EL2;
        when EL3  r = MAIR_EL3;
        otherwise Unreachable();
    return r;

// MAIR[] - non-assignment form
// ============================

MAIRType MAIR[]
    return MAIR[S1TranslationRegime()];
```

5.134 aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

5.135 aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = SCTLR_EL1;
        when EL2  r = SCTLR_EL2;
        when EL3  r = SCTLR_EL3;
        otherwise Unreachable();
    return r;

// SCTLR[] - non-assignment form
// =============================

SCTLRType SCTLR[]
    return SCTLR[S1TranslationRegime()];
```

5.136 aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

5.137 aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1  r = VBAR_EL1<63:0>;
        when EL2  r = VBAR_EL2<63:0>;
        when EL3  r = VBAR_EL3<63:0>;
        otherwise Unreachable();
    return r;

// VBAR[] - non-assignment form
// ============================

bits(64) VBAR[]
    return VBAR[S1TranslationRegime()];
```

5.138 aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// ===========================
// Checks if an AArch64 MSR, MRS or SYS instruction is allowed from the current exception level and
    ↪security state.
// Also checks for traps by TIDCP and NV access.

AArch64.CheckSystemAccess(bits(2) op0, bits(3) op1, bits(4) crn, bits(4) crm, bits(3) op2, bits(5) rt, bit
    ↪read)
    boolean unallocated = FALSE;
    boolean need_secure = FALSE;
    bits(2) min_EL;

    // Check for traps by HCR_EL2.TIDCP
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && HCR_EL2.TIDCP == '1' && op0 == 'x1' && crn == '1x11' then
        // At EL0, it is IMPLEMENTATION_DEFINED whether attempts to execute system
        // register access instructions with reserved encodings are trapped to EL2 or UNDEFINED
        rcs_el0_trap = boolean IMPLEMENTATION_DEFINED "Reserved Control Space EL0 Trapped";
        if PSTATE.EL == EL1 || rcs_el0_trap then
            AArch64.SystemAccessTrap(EL2, 0x18);    // Exception_SystemRegisterTrap

    // Check for unallocated encodings
    case op1 of
        when '00x', '010'
            min_EL = EL1;
        when '011'
            min_EL = EL0;
        when '100'
            min_EL = EL2;
        when '101'
            if !HaveVirtHostExt() then UNDEFINED;
            min_EL = EL2;
        when '110'
            min_EL = EL3;
        when '111'
            min_EL = EL1;
            need_secure = TRUE;
            // RSP_EL0 and RCSP_EL0 are available from EL0, and not Secure-only
            if op0 == '11' && crn == '0100' && crm == '0001' && op2 == '011' then
                min_EL = EL0;
                need_secure = FALSE;

    if UInt(PSTATE.EL) < UInt(min_EL) then
        UNDEFINED;
    elsif need_secure && !IsSecure() then
        UNDEFINED;
```

5.139 aarch64/functions/system/AArch64.ExecutingATS1xPInstr

```
// AArch64.ExecutingATS1xPInstr()
// ==============================
// Return TRUE if current instruction is AT S1E1R/WP

boolean AArch64.ExecutingATS1xPInstr()
    if !HavePrivATExt() then return FALSE;

    instr = ThisInstr();
    if instr<22+:10> == '1101010100' then
        op1  = instr<16+:3>;
        CRn  = instr<12+:4>;
        CRm  = instr<8+:4>;
        op2  = instr<5+:3>;
        return op1 == '000' && CRn == '0111' && CRm == '1001' && op2 IN {'000','001'};
    else
        return FALSE;
```

5.140 aarch64/functions/system/AArch64.SysInstr

```
// Execute a system instruction with write (source operand).
AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

5.141 aarch64/functions/system/AArch64.SysInstrInputIsCapability

```
// AArch64.SysInstrInputIsCapability()
// ===================================

// Does the specified system instruction take a capability as input?

boolean AArch64.SysInstrInputIsCapability(integer op0, integer op1, integer crn, integer crm, integer op2)

    // This returns TRUE for the ZVA, IVAC, CVAC, CVAU, CVAP, CVADP, CIVAC operations for DC,
    // and IC IVAU.
    return (PSTATE.C64 == '1' &&
            ((op0 == 1 && op1 == 0 && crn == 7 && crm == 6) ||
             (op0 == 1 && op1 == 3 && crn == 7 && crm IN {4, 5, 10, 11, 12, 13, 14} && op2 == 1)));
```

5.142 aarch64/functions/system/AArch64.SysInstrWithCapability

```
// Execute a system instruction taking a source capability as input.
AArch64.SysInstrWithCapability(integer op0, integer op1, integer crn, integer crm, integer op2, Capability
    ↪val);
```

5.143 aarch64/functions/system/AArch64.SysInstrWithResult

```
// Execute a system instruction with read (result operand).
// Returns the result of the instruction.
bits(64) AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2);
```

5.144 aarch64/functions/system/AArch64.SysRegRead

```
// Read from a system register and return the contents of the register.
bits(64) AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2);
```

5.145 aarch64/functions/system/AArch64.SysRegWrite

```
// Write to a system register.
AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, bits(64) val);
```

5.146 aarch64/functions/virtualaddress/VAAdd

```
// VAAdd()
// =======

VirtualAddress VAAdd(VirtualAddress v, bits(64) offset)
    VirtualAddress r;
    if VAIsCapability(v) then
        r = VAFromCapability(CapAdd(VAToCapability(v), offset));
    else
        r = VAFromBits64(VAToBits64(v) + offset);

    return r;
```

5.147 aarch64/functions/virtualaddress/VACheckAddress

```
// VACheckAddress()
// ================
// Check Virtual Address against a 64-bit address. If any capability checks
// fail then an appropriate fault will be generated

VACheckAddress(VirtualAddress base, bits(64) addr64, integer size, bits(64) requested_perms, AccType acctype)

    Capability c;

    if VAIsBits64(base) then
        c = DDC[];
        // Note: The effects of CCTLR_ELx.DDCBO are applied in VAddress
    else
        c = VAToCapability(base);

    (-) = CheckCapability(c, addr64, size, requested_perms, acctype);
```

5.148 aarch64/functions/virtualaddress/VACheckPerm

```
// VACheckPerm()
// =============
// Check Virtual Address against a set of permissions.

boolean VACheckPerm(VirtualAddress base, bits(64) requested_perms)

    Capability c;

    if VAIsBits64(base) then
        c = DDC[];
        // Note: The effects of CCTLR_ELx.DDCBO are applied in VAddress
    else
        c = VAToCapability(base);

    return CapCheckPermissions(c, requested_perms);
```

5.149 aarch64/functions/virtualaddress/VAFromBits64

```
// VAFromBits64()
// ==============
// Create a VirtualAddress from a 64-bit value

VirtualAddress VAFromBits64(bits(64) b)
    VirtualAddress v;
    v.vatype = VA_Bits64;
    v.offset = b;

    return v;
```

5.150 aarch64/functions/virtualaddress/VAFromCapability

```
// VAFromCapability()
// ==================
// Create a virtual address from a capability

VirtualAddress VAFromCapability(Capability c)
    VirtualAddress v;

    v.vatype = VA_Capability;
    v.base = c;

    return v;
```

5.151 aarch64/functions/virtualaddress/VAIsBits64

```
// VAIsBits64()
// ============

boolean VAIsBits64(VirtualAddress v)
    return v.vatype == VA_Bits64;
```

5.152 aarch64/functions/virtualaddress/VAIsCapability

```
// VAIsCapability()
// ================

boolean VAIsCapability(VirtualAddress v)
    return v.vatype == VA_Capability;
```

5.153 aarch64/functions/virtualaddress/VAToBits64

```
// VAToBits64()
// ============

bits(64) VAToBits64(VirtualAddress v)
    assert VAIsBits64(v);
    return v.offset;
```

5.154 aarch64/functions/virtualaddress/VAToCapability

```
// VAToCapability()
// ================

Capability VAToCapability(VirtualAddress v)
    assert VAIsCapability(v);
    return v.base;
```

5.155 aarch64/functions/virtualaddress/VAddress

```
// VAddress()
// ==========
// Convert a VirtualAddress to a 64-bit address without checking for validity

bits(64) VAddress(VirtualAddress addr)

    bits(64) addr64;

    if VAIsBits64(addr) then
        if CCTLR[].DDCBO == '1' then
            addr64 = VAToBits64(addr) + CapGetBase(DDC[]);
        else
            addr64 = VAToBits64(addr);
    else
        Capability c = VAToCapability(addr);
        addr64 = CapGetValue(c)<63:0>;

    return addr64;
```

5.156 aarch64/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =========================

AArch64.ExceptionReturn(bits(64) new_pc, bits(32) spsr)

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' && spsr<4> == '1' && spsr<20> == '0' then
        // If the exception return is illegal, PC[63:32,1:0] are UNKNOWN
        new_pc<63:32> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    elsif UsingAArch32() then                    // Return to AArch32
        // ELR_ELx[1:0] or ELR_ELx[0] are treated as being 0, depending on the target instruction set state
        if PSTATE.T == '1' then
            new_pc<0> = '0';                     // T32
        else
            new_pc<1:0> = '00';                  // A32
    else                                         // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc);

    if UsingAArch32() then
        // 32 most significant bits are ignored.
        BranchTo(new_pc<31:0>, BranchType_ERET);
    else
        BranchToAddr(new_pc, BranchType_ERET);
```

5.157 aarch64/instrs/branch/eret/AArch64.ExceptionReturnToCapability

```
// AArch64.ExceptionReturnToCapability()
// =====================================

AArch64.ExceptionReturnToCapability(Capability new_pcc, bits(32) spsr)

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if sync_errors then
        SynchronizeErrors();
        iesb_req = TRUE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if !CapIsSystemAccessEnabled() then
        new_pcc = CapWithTagClear(new_pcc);
    if CapIsExponentOutOfRange(new_pcc) then
        new_pcc = CapWithTagClear(new_pcc);
    new_pcc = BranchAddr(new_pcc, PSTATE.EL);
    BranchToCapability(new_pcc, BranchType_ERET);
```

5.158 aarch64/instrs/countop/CountOp

```
enumeration CountOp     {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

5.159 aarch64/instrs/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =================
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    case op of
        when '000' return ExtendType_UXTB;
        when '001' return ExtendType_UXTH;
        when '010' return ExtendType_UXTW;
        when '011' return ExtendType_UXTX;
        when '100' return ExtendType_SXTB;
        when '101' return ExtendType_SXTH;
        when '110' return ExtendType_SXTW;
        when '111' return ExtendType_SXTX;
```

5.160 aarch64/instrs/extendreg/ExtendReg

```
// ExtendReg()
// ===========
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer len;

    case exttype of
        when ExtendType_SXTB unsigned = FALSE; len = 8;
        when ExtendType_SXTH unsigned = FALSE; len = 16;
        when ExtendType_SXTW unsigned = FALSE; len = 32;
        when ExtendType_SXTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE;  len = 8;
        when ExtendType_UXTH unsigned = TRUE;  len = 16;
        when ExtendType_UXTW unsigned = TRUE;  len = 32;
        when ExtendType_UXTX unsigned = TRUE;  len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    //   [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    //   LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

5.161 aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType  {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                         ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

5.162 aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp  {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                         FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

5.163 aarch64/instrs/float/arithmetic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp   {FPUnaryOp_ABS, FPUnaryOp_MOV,
                         FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

5.164 aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                         FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
};
```

5.165 aarch64/instrs/integer/bitfield/bfxpreferred/BFXPreferred

```
// BFXPreferred()
// ==============
//
// Return TRUE if UBFX or SBFX is the preferred disassembly of a
// UBFM or SBFM bitfield instruction. Must exclude more specific
// aliases UBFIZ, SBFIZ, UXT[BH], SXT[BHW], LSL, LSR and ASR.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)
```

```
    integer S = UInt(imms);
    integer R = UInt(immr);

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // must not match UXTx/SXTx alias
    if immr == '000000' then
        // must not match 32-bit UXT[BH] or SXT[BH]
        if sf == '0' && imms IN {'000111', '001111'} then
            return FALSE;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```

5.166 aarch64/instrs/integer/bitmasks/DecodeBitMasks

```
// DecodeBitMasks()
// ================

// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr, boolean immediate)
    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine S, R and S - R parameters
    levels = ZeroExtend(Ones(len), 6);

    // For logical immediates an all-ones value of S is reserved
    // since it would generate a useless all-ones result (many times)
    if immediate && (imms AND levels) == levels then
        UNDEFINED;

    S = UInt(imms AND levels);
    R = UInt(immr AND levels);
    diff = S - R;    // 6-bit subtract with borrow

    // From a software perspective, the remaining code is equivalant to:
    //   esize = 1 << len;
    //   d = UInt(diff<len-1:0>);
    //   welem = ZeroExtend(Ones(S + 1), esize);
    //   telem = ZeroExtend(Ones(d + 1), esize);
    //   wmask = Replicate(ROR(welem, R));
    //   tmask = Replicate(telem);
    //   return (wmask, tmask);

    // Compute "top mask"
    tmask_and = diff<5:0> OR NOT(levels);
    tmask_or  = diff<5:0> AND levels;

    tmask = Ones(64);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<0>, 1) : Ones(1), 32))
               OR  Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
               OR  Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
               OR  Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
               OR  Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
               OR  Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
              AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
               OR  Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask
              AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
               OR  Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
    // optimization of first step:
    // wmask = Replicate(wmask_or<0> : '0', 32);
    wmask = ((wmask
              AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
               OR  Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
    wmask = ((wmask
              AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
               OR  Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
    wmask = ((wmask
              AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
               OR  Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
    wmask = ((wmask
              AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
               OR  Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
    wmask = ((wmask
              AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
               OR  Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

    if diff<6> != '0' then // borrow from S - R
        wmask = wmask AND tmask;
    else
        wmask = wmask OR tmask;

    return (wmask<M-1:0>, tmask<M-1:0>);
```

5.167 aarch64/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```
enumeration MoveWideOp  {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};
```

5.168 aarch64/instrs/integer/logical/movwpreferred/MoveWidePreferred

```
// MoveWidePreferred()
// ===================
//
// Return TRUE if a bitmask immediate encoding would generate an immediate
// value that could also be represented by a single MOVZ or MOVN instruction.
// Used as a condition for the preferred MOV<-ORR alias.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && immN:imms != '1xxxxxx' then
        return FALSE;
    if sf == '0' && immN:imms != '00xxxxx' then
        return FALSE;

    // for MOVZ must contain no more than 16 ones
    if S < 16 then
        // ones must not span halfword boundary when rotated
        return (-R MOD 16) <= (15 - S);

    // for MOVN must contain no more than 16 zeros
    if S >= width - 15 then
        // zeros must not span halfword boundary when rotated
        return (R MOD 16) <= (S - (width - 15));

    return FALSE;
```

5.169 aarch64/instrs/integer/shiftreg/DecodeShift

```
// DecodeShift()
// =============
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00'  return ShiftType_LSL;
        when '01'  return ShiftType_LSR;
        when '10'  return ShiftType_ASR;
        when '11'  return ShiftType_ROR;
```

5.170 aarch64/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// ==========
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shiftype, integer amount)
    bits(N) result = X[reg];
    case shiftype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

5.171 aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType   {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

5.172 aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp   {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

5.173 aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
                         MemAtomicOp_BIC,
                         MemAtomicOp_EOR,
                         MemAtomicOp_ORR,
                         MemAtomicOp_SMAX,
                         MemAtomicOp_SMIN,
                         MemAtomicOp_UMAX,
                         MemAtomicOp_UMIN,
                         MemAtomicOp_SWP};
```

5.174 aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

5.175 aarch64/instrs/memory/prefetch/Prefetch

```
// Prefetch()
// ==========

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch_READ;         // PLD: prefetch for load
        when '01' hint = Prefetch_EXEC;         // PLI: preload instructions
        when '10' hint = Prefetch_WRITE;        // PST: prepare for store
        when '11' return;                       // unallocated hint
    target = UInt(prfop<2:1>);                  // target cache level
    stream = (prfop<0> != '0');                 // streaming (non-temporal)
    Hint_Prefetch(address, hint, target, stream);
    return;
```

5.176 aarch64/instrs/system/barriers/barrierop/MemBarrierOp

```
enumeration MemBarrierOp   {  MemBarrierOp_DSB         // Data Synchronization Barrier
                            , MemBarrierOp_DMB         // Data Memory Barrier
                            , MemBarrierOp_ISB         // Instruction Synchronization Barrier
                            , MemBarrierOp_SSBB        // Speculative Synchronization Barrier to VA
                            , MemBarrierOp_PSSBB       // Speculative Synchronization Barrier to PA
                            , MemBarrierOp_SB          // Speculation Barrier
                           };
```

5.177 aarch64/instrs/system/hints/syshintop/SystemHintOp

```
enumeration SystemHintOp {
    SystemHintOp_NOP,
    SystemHintOp_YIELD,
    SystemHintOp_WFE,
    SystemHintOp_WFI,
    SystemHintOp_SEV,
    SystemHintOp_SEVL,
    SystemHintOp_ESB,
    SystemHintOp_PSB,
    SystemHintOp_CSDB
};
```

5.178 aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UAO, // Armv8.2
                         PSTATEField_SSBS,
                         PSTATEField_SP
                         };
```

5.179 aarch64/instrs/system/sysops/sysop/SysOp

```
// SysOp()
// =======

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
    case op1:CRn:CRm:op2 of
        when '000 0111 1000 000' return Sys_AT;   // S1E1R
        when '100 0111 1000 000' return Sys_AT;   // S1E2R
        when '110 0111 1000 000' return Sys_AT;   // S1E3R
        when '000 0111 1000 001' return Sys_AT;   // S1E1W
        when '100 0111 1000 001' return Sys_AT;   // S1E2W
        when '110 0111 1000 001' return Sys_AT;   // S1E3W
        when '000 0111 1000 010' return Sys_AT;   // S1E0R
        when '000 0111 1000 011' return Sys_AT;   // S1E0W
        when '100 0111 1000 100' return Sys_AT;   // S12E1R
        when '100 0111 1000 101' return Sys_AT;   // S12E1W
        when '100 0111 1000 110' return Sys_AT;   // S12E0R
        when '100 0111 1000 111' return Sys_AT;   // S12E0W
        when '011 0111 0100 001' return Sys_DC;   // ZVA
        when '000 0111 0110 001' return Sys_DC;   // IVAC
        when '000 0111 0110 010' return Sys_DC;   // ISW
        when '011 0111 1010 001' return Sys_DC;   // CVAC
        when '000 0111 1010 010' return Sys_DC;   // CSW
        when '011 0111 1011 001' return Sys_DC;   // CVAU
        when '011 0111 1110 001' return Sys_DC;   // CIVAC
        when '000 0111 1110 010' return Sys_DC;   // CISW
        when '011 0111 1101 001' return Sys_DC;   // CVADP
        when '000 0111 0001 000' return Sys_IC;   // IALLUIS
        when '000 0111 0101 000' return Sys_IC;   // IALLU
        when '011 0111 0101 001' return Sys_IC;   // IVAU
        when '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
        when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
        when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
        when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
        when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
        when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
        when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
        when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
        when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
        when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
        when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
        when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
        when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
        when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
        when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
        when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
        when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
        when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
        when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
        when '100 1000 0111 000' return Sys_TLBI; // ALLE2
        when '110 1000 0111 000' return Sys_TLBI; // ALLE3
        when '000 1000 0111 001' return Sys_TLBI; // VAE1
        when '100 1000 0111 001' return Sys_TLBI; // VAE2
        when '110 1000 0111 001' return Sys_TLBI; // VAE3
        when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
        when '000 1000 0111 011' return Sys_TLBI; // VAAE1
        when '100 1000 0111 100' return Sys_TLBI; // ALLE1
        when '000 1000 0111 101' return Sys_TLBI; // VALE1
        when '100 1000 0111 101' return Sys_TLBI; // VALE2
        when '110 1000 0111 101' return Sys_TLBI; // VALE3
        when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
        when '000 1000 0111 111' return Sys_TLBI; // VAALE1
    return Sys_SYS;
```

5.180 aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

5.181 aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp      {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

5.182 aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp   {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                         CompareOp_LE, CompareOp_LT};
```

5.183 aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                         ImmediateOp_ORR, ImmediateOp_BIC};
```

5.184 aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// ========

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize);
    lo = Reduce(op, input<half-1:0>, esize);

    case op of
        when ReduceOp_FMINNUM
            result = FPMinNum(lo, hi, FPCR);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

5.185 aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMINNUM, ReduceOp_FMAXNUM,
                      ReduceOp_FMIN, ReduceOp_FMAX,
                      ReduceOp_FADD, ReduceOp_ADD};
```

5.186 aarch64/translation/attrs/AArch64.CombineS1S2Desc

```
// AArch64.CombineS1S2Desc()
// =========================
// Combines the address descriptors from stage 1 and stage 2

AddressDescriptor AArch64.CombineS1S2Desc(AddressDescriptor s1desc, AddressDescriptor s2desc)

    AddressDescriptor result;

    result.paddress = s2desc.paddress;

    if IsFault(s1desc) || IsFault(s2desc) then
        result = if IsFault(s1desc) then s1desc else s2desc;
    else
        result.fault = AArch64.NoFault();
        if s2desc.memattrs.memtype == MemType_Device || s1desc.memattrs.memtype == MemType_Device then
            result.memattrs.memtype = MemType_Device;
            if s1desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s2desc.memattrs.device;
            elsif s2desc.memattrs.memtype == MemType_Normal then
                result.memattrs.device = s1desc.memattrs.device;
            else                    // Both Device
                result.memattrs.device = CombineS1S2Device(s1desc.memattrs.device,
                                                       s2desc.memattrs.device);
        else                        // Both Normal
            result.memattrs.memtype = MemType_Normal;
            result.memattrs.device = DeviceType UNKNOWN;
            result.memattrs.inner = CombineS1S2AttrHints(s1desc.memattrs.inner, s2desc.memattrs.inner);
            result.memattrs.outer = CombineS1S2AttrHints(s1desc.memattrs.outer, s2desc.memattrs.outer);
            result.memattrs.shareable = (s1desc.memattrs.shareable || s2desc.memattrs.shareable);
            result.memattrs.outershareable = (s1desc.memattrs.outershareable ||
                                              s2desc.memattrs.outershareable);
    result.memattrs = CombineS1S2LCSC(result.memattrs, s1desc.memattrs, s2desc.memattrs);

    result.memattrs = CanonicalizeMemoryAttributes(result.memattrs);

    return result;
```

5.187 aarch64/translation/attrs/AArch64.InstructionDevice

```
// AArch64.InstructionDevice()
// ===========================
// Instruction fetches from memory marked as Device but not execute-never might generate a
// Permission Fault but are otherwise treated as if from Normal Non-cacheable memory.

AddressDescriptor AArch64.InstructionDevice(AddressDescriptor addrdesc, bits(64) vaddress,
                                            bits(48) ipaddress, integer level,
                                            AccType acctype, boolean iswrite, boolean secondstage,
                                            boolean s2fs1walk)

    c = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
    assert c IN {Constraint_NONE, Constraint_FAULT};

    if c == Constraint_FAULT then
        addrdesc.fault = AArch64.PermissionFault(ipaddress,   level, acctype, iswrite,
                                                 secondstage, s2fs1walk);
    else
        addrdesc.memattrs.memtype = MemType_Normal;
        addrdesc.memattrs.inner.attrs = MemAttr_NC;
        addrdesc.memattrs.inner.hints = MemHint_No;
        addrdesc.memattrs.outer = addrdesc.memattrs.inner;
        addrdesc.memattrs = CanonicalizeMemoryAttributes(addrdesc.memattrs);

    return addrdesc;
```

5.188 aarch64/translation/attrs/AArch64.S1AttrDecode

```
// AArch64.S1AttrDecode()
// ======================
// Converts the Stage 1 attribute fields, using the MAIR, to orthogonal
// attributes and hints.

MemoryAttributes AArch64.S1AttrDecode(bits(2) SH, bits(3) attr, AccType acctype)

    MemoryAttributes memattrs;

    mair = MAIR[];
    index = 8 * UInt(attr);
    attrfield = mair<index+7:index>;

    if ((attrfield<7:4> != '0000' && attrfield<3:0> == '0000') ||
        (attrfield<7:4> == '0000' && attrfield<3:0> != 'xx00')) then
        // Reserved, maps to an allocated value
        (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

    if attrfield<7:4> == '0000' then            // Device
        memattrs.memtype = MemType_Device;
        case attrfield<3:0> of
            when '0000'  memattrs.device = DeviceType_nGnRnE;
            when '0100'  memattrs.device = DeviceType_nGnRE;
            when '1000'  memattrs.device = DeviceType_nGRE;
            when '1100'  memattrs.device = DeviceType_GRE;
            otherwise    Unreachable();         // Reserved, handled above

    elsif attrfield<3:0> != '0000'  then        // Normal
        memattrs.memtype = MemType_Normal;
        memattrs.outer = LongConvertAttrsHints(attrfield<7:4>, acctype);
        memattrs.inner = LongConvertAttrsHints(attrfield<3:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    else
        Unreachable();                          // Reserved, handled above

    return CanonicalizeMemoryAttributes(memattrs);
```

5.189 aarch64/translation/attrs/AArch64.TranslateAddressS1Off

```
// AArch64.TranslateAddressS1Off()
// ===============================
// Called for stage 1 translations when translation is disabled to supply a default translation.
// Note that there are additional constraints on instruction prefetching that are not described in
// this pseudocode.

TLBRecord AArch64.TranslateAddressS1Off(bits(64) vaddress, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    TLBRecord result;

    Top = AddrTop(vaddress, PSTATE.EL);
    if !IsZero(vaddress<Top:PAMax()>) then
        level = 0;
        ipaddress = bits(48) UNKNOWN;
        secondstage = FALSE;
        s2fs1walk = FALSE;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,  level, acctype,
                                                          iswrite, secondstage, s2fs1walk);
        return result;

    default_cacheable = (HasS2Translation() && HCR_EL2.DC == '1');

    if default_cacheable then
        // Use default cacheable settings
        result.addrdesc.memattrs.memtype = MemType_Normal;
        result.addrdesc.memattrs.inner.attrs = MemAttr_WB;      // Write-back
        result.addrdesc.memattrs.inner.hints = MemHint_RWA;
        result.addrdesc.memattrs.shareable = FALSE;
        result.addrdesc.memattrs.outershareable = FALSE;
    elsif acctype != AccType_IFETCH then
        // Treat data as Device
        result.addrdesc.memattrs.memtype = MemType_Device;
        result.addrdesc.memattrs.device = DeviceType_nGnRnE;
        result.addrdesc.memattrs.inner = MemAttrHints UNKNOWN;
    else
        // Instruction cacheability controlled by SCTLR_ELx.I
        cacheable = SCTLR[].I == '1';
        result.addrdesc.memattrs.memtype = MemType_Normal;
        if cacheable then
            result.addrdesc.memattrs.inner.attrs = MemAttr_WT;
            result.addrdesc.memattrs.inner.hints = MemHint_RA;
        else
            result.addrdesc.memattrs.inner.attrs = MemAttr_NC;
            result.addrdesc.memattrs.inner.hints = MemHint_No;
        result.addrdesc.memattrs.shareable = TRUE;
        result.addrdesc.memattrs.outershareable = TRUE;

    result.addrdesc.memattrs.outer = result.addrdesc.memattrs.inner;
    result.addrdesc.memattrs = CanonicalizeMemoryAttributes(result.addrdesc.memattrs);

    // CDBM, LC and SC behavior is defined such that there is no
    // zeroing of tags, no faults and no tracking of stores.
    result.addrdesc.memattrs.readtagzero = FALSE;
    result.addrdesc.memattrs.writetagfault = FALSE;
    result.addrdesc.memattrs.readtagfault = FALSE;
    result.addrdesc.memattrs.readtagfaulttgen = bit UNKNOWN;
    result.addrdesc.memattrs.iss2writetagfault = FALSE;

    result.perms.ap = bits(3) UNKNOWN;
    result.perms.xn = '0';
    result.perms.pxn = '0';

    result.nG = bit UNKNOWN;
    result.contiguous = boolean UNKNOWN;
    result.domain = bits(4) UNKNOWN;
    result.level = integer UNKNOWN;
    result.blocksize = integer UNKNOWN;
    result.addrdesc.paddress.address = vaddress<47:0>;
    result.addrdesc.paddress.NS = if IsSecure() then '0' else '1';
    result.addrdesc.fault = AArch64.NoFault();
    return result;
```

5.190 aarch64/translation/checks/AArch64.AccessIsPrivileged

```
// AArch64.AccessIsPrivileged()
// ============================

boolean AArch64.AccessIsPrivileged(AccType acctype)

    el = AArch64.AccessUsesEL(acctype);

    if el == EL0 then
        ispriv = FALSE;
    elsif el == EL3 then
        ispriv = TRUE;
    elsif el == EL2 && (!IsInHost() || HCR_EL2.TGE == '0') then
        ispriv = TRUE;
    elsif HaveUAOExt() && PSTATE.UAO == '1' then
        ispriv = TRUE;
    else
        ispriv = (acctype != AccType_UNPRIV);

    return ispriv;
```

5.191 aarch64/translation/checks/AArch64.AccessUsesEL

```
// AArch64.AccessUsesEL()
// ======================
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

5.192 aarch64/translation/checks/AArch64.CheckLoadTagsPermission

```
// AArch64.CheckLoadTagsPermission()
// =================================
// Function used for load tag checking

CheckLoadTagsPermission(AddressDescriptor desc, AccType acctype)
    if desc.memattrs.readtagfault then
        bit fault_tgen = desc.memattrs.readtagfaulttgen;
        if EffectiveTGEN(desc.vaddress, PSTATE.EL) == fault_tgen then
            secondstage = FALSE;
            is_store = FALSE;
            FaultRecord fault = AArch64.CapabilityPagePermissionFault(acctype, secondstage, is_store);
            AArch64.Abort(desc.vaddress, fault);
```

5.193 aarch64/translation/checks/AArch64.CheckPermission

```
// AArch64.CheckPermission()
// =========================
// Function used for permission checking from AArch64 stage 1 translations

FaultRecord AArch64.CheckPermission(Permissions perms, bits(64) vaddress, integer level,
                                    bit NS, AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());

    wxn = SCTLR[].WXN == '1';

    if (PSTATE.EL == EL0 ||
        IsInHost() ||
        PSTATE.EL == EL1) then
        priv_r = TRUE;
        priv_w = perms.ap<2> == '0';
        user_r = perms.ap<1> == '1';
        user_w = perms.ap<2:1> == '01';

        ispriv = AArch64.AccessIsPrivileged(acctype);

        pan = if HavePANExt() then PSTATE.PAN else '0';
        is_ldst   = !(acctype IN {AccType_DC, AccType_DC_UNPRIV, AccType_AT, AccType_IFETCH});
        is_ats1xp = (acctype == AccType_AT && AArch64.ExecutingATS1xPInstr());
        if pan == '1' && user_r && ispriv && (is_ldst || is_ats1xp) then
            priv_r = FALSE;
            priv_w = FALSE;

        user_xn = perms.xn == '1' || (user_w && wxn);
        priv_xn = perms.pxn == '1' || (priv_w && wxn) || user_w;

        if ispriv then
            (r, w, xn) = (priv_r, priv_w, priv_xn);
        else
            (r, w, xn) = (user_r, user_w, user_xn);
    else
        // Access from EL2 or EL3
        r = TRUE;
        w = perms.ap<2> == '0';
        xn = perms.xn == '1' || (w && wxn);

    // Restriction on Secure instruction fetch
    if HaveEL(EL3) && IsSecure() && NS == '1' && SCR_EL3.SIF == '1' then
        xn = TRUE;

    if acctype == AccType_IFETCH then
        fail = xn;
        failedread = TRUE;
    elsif acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW } then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC && PSTATE.EL != EL0 then
        // DC maintenance instructions operating by VA, cannot fault from stage 1 translation,
        // other than DC IVAC, which requires write permission, and operations executed at EL0,
        // which require read permission.
        fail = FALSE;
    else
        fail = !r;
        failedread = TRUE;

    if fail then
        secondstage = FALSE;
        s2fs1walk = FALSE;
        ipaddress = bits(48) UNKNOWN;
        return AArch64.PermissionFault(ipaddress,  level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

5.194 aarch64/translation/checks/AArch64.CheckS2Permission

```
// AArch64.CheckS2Permission()
// ===========================
// Function used for permission checking from AArch64 stage 2 translations

FaultRecord AArch64.CheckS2Permission(Permissions perms, bits(64) vaddress, bits(48) ipaddress,
                                      integer level, AccType acctype, boolean iswrite,
                                      boolean s2fs1walk, boolean hwupdatewalk)

    assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

    r = perms.ap<1> == '1';
    w = perms.ap<2> == '1';
    if HaveExtendedExecuteNeverExt() then
        case perms.xn:perms.xxn of
            when '00'  xn = FALSE;
            when '01'  xn = PSTATE.EL == EL1;
            when '10'  xn = TRUE;
            when '11'  xn = PSTATE.EL == EL0;
    else
        xn = perms.xn == '1';
    // Stage 1 walk is checked as a read, regardless of the original type
    if acctype == AccType_IFETCH && !s2fs1walk then
        fail = xn;
        failedread = TRUE;
    elsif (acctype IN { AccType_ATOMICRW, AccType_ORDEREDRW, AccType_ORDEREDATOMICRW }) && !s2fs1walk then
        fail = !r || !w;
        failedread = !r;
    elsif iswrite && !s2fs1walk then
        fail = !w;
        failedread = FALSE;
    elsif acctype == AccType_DC && PSTATE.EL != EL0 && !s2fs1walk then
        // DC maintenance instructions operating by VA, with the exception of DC IVAC, do
        // not generate Permission faults from stage 2 translation, other than when
        // performing a stage 1 translation table walk.
        fail = FALSE;
    elsif hwupdatewalk then
        fail = !w;
        failedread = !iswrite;
    else
        fail = !r;
        failedread = !iswrite;

    if fail then
        domain = bits(4) UNKNOWN;
        secondstage = TRUE;
        return AArch64.PermissionFault(ipaddress,  level, acctype,
                                       !failedread, secondstage, s2fs1walk);
    else
        return AArch64.NoFault();
```

5.195 aarch64/translation/checks/AArch64.CheckStoreTagsPermission

```
// AArch64.CheckStoreTagsPermission()
// ==================================
// Function used for store tag checking

CheckStoreTagsPermission(AddressDescriptor desc, AccType acctype)
    if desc.memattrs.writetagfault then
        is_store = TRUE;
        FaultRecord fault = AArch64.CapabilityPagePermissionFault(acctype,
            ↪desc.memattrs.iss2writetagfault, is_store);
        AArch64.Abort(desc.vaddress, fault);
```

5.196 aarch64/translation/debug/AArch64.CheckBreakpoint

```
// AArch64.CheckBreakpoint()
// =========================
// Called before executing the instruction of length "size" bytes at "vaddress" in an AArch64
// translation regime, when either debug exceptions are enabled, or halting debug is enabled
// and halting is allowed.

FaultRecord AArch64.CheckBreakpoint(bits(64) vaddress,  integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    match = FALSE;

    for i = 0 to UInt(ID_AA64DFR0_EL1.BRPs)
        match_i = AArch64.BreakpointMatch(i, vaddress, size);
        match = match || match_i;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Breakpoint;
        Halt(reason);
    elsif match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

5.197 aarch64/translation/debug/AArch64.CheckDebug

```
// AArch64.CheckDebug()
// ====================
// Called on each access to check for a debug exception or entry to Debug state.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = AArch64.NoFault();

    d_side = (acctype != AccType_IFETCH);
    generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = HaltOnBreakpointOrWatchpoint();

    if generate_exception || halt then
        if d_side then
            fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
        else
            fault = AArch64.CheckBreakpoint(vaddress, size);

    return fault;
```

5.198 aarch64/translation/debug/AArch64.CheckWatchpoint

```
// AArch64.CheckWatchpoint()
// =========================
// Called before accessing the memory location of "size" bytes at "address",
// when either debug exceptions are enabled for the access, or halting debug
// is enabled and halting is allowed.

FaultRecord AArch64.CheckWatchpoint(bits(64) vaddress, AccType acctype,
                                    boolean iswrite, integer size)
    assert !ELUsingAArch32(S1TranslationRegime());

    match = FALSE;
    ispriv = AArch64.AccessIsPrivileged(acctype);

    for i = 0 to UInt(ID_AA64DFR0_EL1.WRPs)
        match = match || AArch64.WatchpointMatch(i, vaddress, size, ispriv, iswrite);

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        Halt(reason);
    elsif match then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return AArch64.NoFault();
```

5.199 aarch64/translation/faults/AArch64.AccessFlagFault

```
// AArch64.AccessFlagFault()
// =========================

FaultRecord AArch64.AccessFlagFault(bits(48) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AccessFlag, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.200 aarch64/translation/faults/AArch64.AddressSizeFault

```
// AArch64.AddressSizeFault()
// ==========================

FaultRecord AArch64.AddressSizeFault(bits(48) ipaddress, integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_AddressSize, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.201 aarch64/translation/faults/AArch64.AlignmentFault

```
// AArch64.AlignmentFault()
// ========================

FaultRecord AArch64.AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    s2fs1walk = boolean UNKNOWN;

    return AArch64.CreateFaultRecord(Fault_Alignment, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.202 aarch64/translation/faults/AArch64.AsynchExternalAbort

```
// AArch64.AsynchExternalAbort()
// =============================
// Wrapper function for asynchronous external aborts

FaultRecord AArch64.AsynchExternalAbort(boolean parity, bits(2) errortype, bit extflag)

    faulttype = if parity then Fault_AsyncParity else Fault_AsyncExternal;
    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(faulttype, ipaddress,  level, acctype, iswrite, extflag,
                                      errortype, secondstage, s2fs1walk);

FaultRecord AArch64.CapabilityPagePermissionFault(AccType acctype, boolean secondstage, boolean is_store)

    ipaddress = bits(48) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_CapPagePerm, ipaddress,  level, acctype, is_store,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.203 aarch64/translation/faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// ====================

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)

    ipaddress = bits(48) UNKNOWN;
    errortype = bits(2) UNKNOWN;
    level = integer UNKNOWN;
    extflag = bit UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_Debug, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.204 aarch64/translation/faults/AArch64.NoFault

```
// AArch64.NoFault()
// =================

FaultRecord AArch64.NoFault()

    ipaddress = bits(48) UNKNOWN;
    level = integer UNKNOWN;
    acctype = AccType_NORMAL;
    iswrite = boolean UNKNOWN;
    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    return AArch64.CreateFaultRecord(Fault_None, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.205 aarch64/translation/faults/AArch64.PermissionFault

```
// AArch64.PermissionFault()
// =========================

FaultRecord AArch64.PermissionFault(bits(48) ipaddress, integer level,
                                    AccType acctype, boolean iswrite, boolean secondstage,
                                    boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Permission, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.206 aarch64/translation/faults/AArch64.TranslationFault

```
// AArch64.TranslationFault()
// ==========================

FaultRecord AArch64.TranslationFault(bits(48) ipaddress,  integer level,
                                     AccType acctype, boolean iswrite, boolean secondstage,
                                     boolean s2fs1walk)

    extflag = bit UNKNOWN;
    errortype = bits(2) UNKNOWN;
    return AArch64.CreateFaultRecord(Fault_Translation, ipaddress,  level, acctype, iswrite,
                                     extflag, errortype, secondstage, s2fs1walk);
```

5.207 aarch64/translation/translation/AArch64.CheckAndUpdateDescriptor

```
// AArch64.CheckAndUpdateDescriptor()
// ==================================
// Check and update translation table descriptor if hardware update is configured

FaultRecord AArch64.CheckAndUpdateDescriptor(DescriptorUpdate result, FaultRecord fault,
                                             boolean secondstage, bits(64) vaddress, AccType acctype,
                                             boolean iswrite, boolean s2fs1walk, boolean hwupdatewalk,
                                                  ↪boolean iswritevalidcap)

    boolean hw_update_AF = FALSE;
    boolean hw_update_AP = FALSE;
    boolean hw_update_SC = FALSE;

    // Check if access flag can be updated
    // Address translation instructions are permitted to update AF but not required
    if result.AF then
        if fault.statuscode == Fault_None || ConstrainUnpredictable(Unpredictable_AFUPDATE) ==
             ↪Constraint_TRUE then
            hw_update_AF = TRUE;

    write_perm_req = (iswrite || acctype IN {AccType_ATOMICRW,AccType_ORDEREDRW, AccType_ORDEREDATOMICRW
         ↪}) && !s2fs1walk;
    if result.AP && fault.statuscode == Fault_None then
        hw_update_AP = (write_perm_req && !(acctype IN {AccType_AT, AccType_DC, AccType_DC_UNPRIV})) ||
             ↪hwupdatewalk;

    if result.SC && fault.statuscode == Fault_None && iswritevalidcap && write_perm_req then
        hw_update_SC = TRUE;

    if hw_update_AF || hw_update_AP || hw_update_SC then
        if secondstage || !HasS2Translation() then
            descaddr2 = result.descaddr;
        else
            hwupdatewalk = TRUE;
            descaddr2 = AArch64.SecondStageWalk(result.descaddr, vaddress, acctype, iswrite, 8,
                ↪hwupdatewalk);
            if IsFault(descaddr2) then
                return descaddr2.fault;

        accdesc = CreateAccessDescriptor(AccType_ATOMICRW);
        desc = _Mem[descaddr2, 8, accdesc];
        el = AArch64.AccessUsesEL(acctype);
        case el of
            when EL3
                reversedescriptors = SCTLR_EL3.EE == '1';
            when EL2
                reversedescriptors = SCTLR_EL2.EE == '1';
            otherwise
                reversedescriptors = SCTLR_EL1.EE == '1';
        if reversedescriptors then
            desc = BigEndianReverse(desc);

        if hw_update_AF then
            desc<10> = '1';
        if hw_update_AP then
            desc<7> = (if secondstage then '1' else '0');
        if hw_update_SC then
            desc<60> = '1';

        _Mem[descaddr2,8,accdesc] = if reversedescriptors then BigEndianReverse(desc) else desc;

    return fault;
```

5.208 aarch64/translation/translation/AArch64.FirstStageTranslate

```
// AArch64.FirstStageTranslate()
// =============================
// Perform a stage 1 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.FirstStageTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                              boolean wasaligned, integer size)
    boolean iswritevalidcap = FALSE;
    return AArch64.FirstStageTranslateWithTag(vaddress, acctype, iswrite, wasaligned, size,
        ↪iswritevalidcap);
```

5.209 aarch64/translation/translation/AArch64.FirstStageTranslateWithTag

```
// AArch64.FirstStageTranslateWithTag()
// ====================================
// Perform a stage 1 translation walk.
// An additional argument specifies whether the translation is used for writing a valid capability.

AddressDescriptor AArch64.FirstStageTranslateWithTag(bits(64) vaddress, AccType acctype, boolean iswrite,
                                                     boolean wasaligned, integer size, boolean
                                                          ↪iswritevalidcap)

    s1_enabled = AArch64.IsStageOneEnabled(acctype);
    ipaddress = bits(48) UNKNOWN;
    secondstage = FALSE;
    s2fs1walk = FALSE;

    if s1_enabled then                          // First stage enabled
        S1 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                            s2fs1walk, size);
        permissioncheck = TRUE;
    else
        S1 = AArch64.TranslateAddressS1Off(vaddress, acctype, iswrite);
        permissioncheck = FALSE;

    // Check for unaligned data accesses to Device memory
    if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
        && !IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device then
        S1.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);
    if !IsFault(S1.addrdesc) && permissioncheck then
        S1.addrdesc.fault = AArch64.CheckPermission(S1.perms, vaddress, S1.level,
                                                    S1.addrdesc.paddress.NS,
                                                    acctype, iswrite);

    // Check for instruction fetches from Device memory not marked as execute-never. If there has
    // not been a Permission Fault then the memory is not marked execute-never.
    if (!IsFault(S1.addrdesc) && S1.addrdesc.memattrs.memtype == MemType_Device &&
        acctype == AccType_IFETCH) then
        S1.addrdesc = AArch64.InstructionDevice(S1.addrdesc, vaddress, ipaddress, S1.level,
                                                acctype, iswrite,
                                                secondstage, s2fs1walk);
    // Check and update translation table descriptor if required
    hwupdatewalk = FALSE;
    s2fs1walk = FALSE;
    S1.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S1.descupdate, S1.addrdesc.fault,
                                                         secondstage, vaddress, acctype,
                                                         iswrite, s2fs1walk, hwupdatewalk,
                                                              ↪iswritevalidcap);

    return S1.addrdesc;
```

5.210 aarch64/translation/translation/AArch64.FullTranslate

```
// AArch64.FullTranslate()
// =======================
// Perform both stage 1 and stage 2 translation walks for the current translation regime. The
// function used by Address Translation operations is similar except it uses the translation
// regime specified for the instruction.

AddressDescriptor AArch64.FullTranslate(bits(64) vaddress, AccType acctype, boolean iswrite,
                                        boolean wasaligned, integer size)
    boolean iswritevalidcap = FALSE;
    return AArch64.FullTranslateWithTag(vaddress, acctype, iswrite, wasaligned, size, iswritevalidcap);
```

5.211 aarch64/translation/translation/AArch64.FullTranslateWithTag

```
// AArch64.FullTranslateWithTag()
// ==============================
// Perform both stage 1 and stage 2 translation walks for the current translation regime.
// An additional argument specifies whether the translation is used for writing a valid capability.

AddressDescriptor AArch64.FullTranslateWithTag(bits(64) vaddress, AccType acctype, boolean iswrite,
                                               boolean wasaligned, integer size, boolean iswritevalidcap)

    // First Stage Translation
    S1 = AArch64.FirstStageTranslateWithTag(vaddress, acctype, iswrite, wasaligned, size, iswritevalidcap);
    if !IsFault(S1) && HasS2Translation() then
        s2fs1walk = FALSE;
        hwupdatewalk = FALSE;
        // If the first stage of translation will fault a write of a valid capability
        // the second stage of translation should not perform any hardware update due to
        // a store of a valid capability.
        if S1.memattrs.writetagfault then
            iswritevalidcap = FALSE;
        result = AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                              size, hwupdatewalk, iswritevalidcap);
    else
        result = S1;

    return result;
```

5.212 aarch64/translation/translation/AArch64.IsStageOneEnabled

```
// AArch64.IsStageOneEnabled()
// ===========================

boolean AArch64.IsStageOneEnabled(AccType acctype)

    if HasS2Translation() then
        s1_enabled = HCR_EL2.TGE == '0' && HCR_EL2.DC == '0' && SCTLR_EL1.M == '1';
    else
        s1_enabled = SCTLR[].M == '1';

    return s1_enabled;
```

5.213 aarch64/translation/translation/AArch64.SecondStageTranslate

```
// AArch64.SecondStageTranslate()
// ==============================
// Perform a stage 2 translation walk. The function used by Address Translation operations is
// similar except it uses the translation regime specified for the instruction.

AddressDescriptor AArch64.SecondStageTranslate(AddressDescriptor S1, bits(64) vaddress,
                                               AccType acctype, boolean iswrite, boolean wasaligned,
                                               boolean s2fs1walk, integer size, boolean hwupdatewalk,
                                                    ↪boolean iswritevalidcap)
    assert HasS2Translation();

    s2_enabled = HCR_EL2.VM == '1' || HCR_EL2.DC == '1';
    secondstage = TRUE;

    if s2_enabled then                        // Second stage enabled
        ipaddress = S1.paddress.address<47:0>;
        S2 = AArch64.TranslationTableWalk(ipaddress, vaddress, acctype, iswrite, secondstage,
                                            s2fs1walk, size);

        // Check for unaligned data accesses to Device memory
        if ((!wasaligned && acctype != AccType_IFETCH) || (acctype == AccType_DCZVA))
            && S2.addrdesc.memattrs.memtype == MemType_Device && !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.AlignmentFault(acctype, iswrite, secondstage);

        // Check for permissions on Stage2 translations
        if !IsFault(S2.addrdesc) then
            S2.addrdesc.fault = AArch64.CheckS2Permission(S2.perms, vaddress, ipaddress, S2.level,
                                                          acctype, iswrite, s2fs1walk, hwupdatewalk);

        // Check for instruction fetches from Device memory not marked as execute-never. As there
        // has not been a Permission Fault then the memory is not marked execute-never.
        if (!s2fs1walk && !IsFault(S2.addrdesc) && S2.addrdesc.memattrs.memtype == MemType_Device &&
            acctype == AccType_IFETCH) then
            S2.addrdesc = AArch64.InstructionDevice(S2.addrdesc, vaddress, ipaddress, S2.level,
                                                    acctype, iswrite,
                                                    secondstage, s2fs1walk);

        // Check for protected table walk
        if (s2fs1walk && !IsFault(S2.addrdesc) && HCR_EL2.PTW == '1' &&
            S2.addrdesc.memattrs.memtype == MemType_Device) then
            S2.addrdesc.fault = AArch64.PermissionFault(ipaddress,   S2.level, acctype,
                                                        iswrite, secondstage, s2fs1walk);

        // Check and update translation table descriptor if required
        S2.addrdesc.fault = AArch64.CheckAndUpdateDescriptor(S2.descupdate, S2.addrdesc.fault,
                                                             secondstage, vaddress, acctype,
                                                             iswrite, s2fs1walk, hwupdatewalk,
                                                                  ↪iswritevalidcap);
        result = AArch64.CombineS1S2Desc(S1, S2.addrdesc);
    else
        result = S1;

    return result;
```

5.214 aarch64/translation/translation/AArch64.SecondStageWalk

```
// AArch64.SecondStageWalk()
// =========================
// Perform a stage 2 translation on a stage 1 translation page table walk access.

AddressDescriptor AArch64.SecondStageWalk(AddressDescriptor S1, bits(64) vaddress, AccType acctype,
                                          boolean iswrite, integer size, boolean hwupdatewalk)

    assert HasS2Translation();

    s2fs1walk = TRUE;
    wasaligned = TRUE;
    iswritevalidcap = FALSE;
    return AArch64.SecondStageTranslate(S1, vaddress, acctype, iswrite, wasaligned, s2fs1walk,
                                        size, hwupdatewalk, iswritevalidcap);
```

5.215 aarch64/translation/translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// ==========================
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) vaddress, AccType acctype, boolean iswrite,
                                           boolean wasaligned, integer size)
    boolean iswritevalidcap = FALSE;
    return AArch64.TranslateAddressWithTag(vaddress, acctype, iswrite, wasaligned, size, iswritevalidcap);
```

5.216 aarch64/translation/translation/AArch64.TranslateAddressWithTag

```
// AArch64.TranslateAddressWithTag()
// =================================
// Entry point for translating an address with an additional argument specifying if the translation
// is for writing a valid capability

AddressDescriptor AArch64.TranslateAddressWithTag(bits(64) vaddress, AccType acctype, boolean iswrite,
                                                  boolean wasaligned, integer size, boolean
                                                      ↪iswritevalidcap)
    assert(iswrite || !iswritevalidcap);
    result = AArch64.FullTranslateWithTag(vaddress, acctype, iswrite, wasaligned, size, iswritevalidcap);

    if !(acctype IN {AccType_PTW, AccType_IC, AccType_AT}) && !IsFault(result) then
        result.fault = AArch64.CheckDebug(vaddress, acctype, iswrite, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(vaddress);

    return result;
```

5.217 aarch64/translation/walk/AArch64.TranslationTableWalk

```
// AArch64.TranslationTableWalk()
// ==============================
// Returns a result of a translation table walk
//
// Implementations might cache information from memory in any number of non-coherent TLB
// caching structures, and so avoid memory accesses that have been expressed in this
// pseudocode. The use of such TLBs is not expressed in this pseudocode.

TLBRecord AArch64.TranslationTableWalk(bits(48) ipaddress, bits(64) vaddress,
                                       AccType acctype, boolean iswrite, boolean secondstage,
                                       boolean s2fs1walk, integer size)
    if !secondstage then
        assert !ELUsingAArch32(S1TranslationRegime());
    else
        assert HaveEL(EL2) && !IsSecure() && !ELUsingAArch32(EL2) && HasS2Translation();

    TLBRecord result;
    AddressDescriptor descaddr;
    bits(64) baseregister;
    bits(64) inputaddr;        // Input Address is 'vaddress' for stage 1, 'ipaddress' for stage 2

    descaddr.memattrs.memtype = MemType_Normal;

    // Derived parameters for the page table walk:
    //  grainsize = Log2(Size of Table)         - Size of Table is 4KB, 16KB or 64KB in AArch64
    //  stride = Log2(Address per Level)        - Bits of address consumed at each level
    //  firstblocklevel = First level where a block entry is allowed
    //  ps = Physical Address size as encoded in TCR_EL1.IPS or TCR_ELx/VTCR_EL2.PS
    //  inputsize = Log2(Size of Input Address) - Input Address size in bits
    //  level = Level to start walk from
    // This means that the number of levels after start level = 3-level

    if !secondstage then
        // First stage translation
        inputaddr = ZeroExtend(vaddress);
        el = AArch64.AccessUsesEL(acctype);
        top = AddrTop(inputaddr, el);
        if el == EL3 then
            largegrain = TCR_EL3.TG0 == '01';
            midgrain = TCR_EL3.TG0 == '10';
            inputsize = 64 - UInt(TCR_EL3.T0SZ);
            inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            ps = TCR_EL3.PS;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
                 ↪IsZero(inputaddr<top:inputsize>);
            disabled = FALSE;
            baseregister = TTBR0_EL3;
            descaddr.memattrs = WalkAttrDecode(TCR_EL3.SH0, TCR_EL3.ORGN0, TCR_EL3.IRGN0, secondstage);
            reversedescriptors = SCTLR_EL3.EE == '1';
            lookupsecure = TRUE;
            singlepriv = TRUE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL3.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL3.HD == '1';
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL3.HPD == '1';
        elsif ELIsInHost(el) then
            if inputaddr<top> == '0' then
                largegrain = TCR_EL2.TG0 == '01';
                midgrain = TCR_EL2.TG0 == '10';
                inputsize = 64 - UInt(TCR_EL2.T0SZ);
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - 39;
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
                     ↪IsZero(inputaddr<top:inputsize>);
                disabled = TCR_EL2.EPD0 == '1';
                baseregister = TTBR0_EL2;
                descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD0 == '1';
            else
                inputsize = 64 - UInt(TCR_EL2.T1SZ);
                largegrain = TCR_EL2.TG1 == '11';       // TG1 and TG0 encodings differ
                midgrain = TCR_EL2.TG1 == '01';
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - 39;
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
                     ↪IsOnes(inputaddr<top:inputsize>);
                disabled = TCR_EL2.EPD1 == '1';
                baseregister = TTBR1_EL2;
                descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH1, TCR_EL2.ORGN1, TCR_EL2.IRGN1, secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD1 == '1';
            ps = TCR_EL2.IPS;
            reversedescriptors = SCTLR_EL2.EE == '1';
            lookupsecure = FALSE;
            singlepriv = FALSE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
        elsif el == EL2 then
            inputsize = 64 - UInt(TCR_EL2.T0SZ);
            largegrain = TCR_EL2.TG0 == '01';
            midgrain = TCR_EL2.TG0 == '10';
            inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
            inputsize_min = 64 - 39;
            if inputsize < inputsize_min then
                c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                assert c IN {Constraint_FORCE, Constraint_FAULT};
                if c == Constraint_FORCE then inputsize = inputsize_min;
            ps = TCR_EL2.PS;
            basefound = inputsize >= inputsize_min && inputsize <= inputsize_max &&
                 ↪IsZero(inputaddr<top:inputsize>);
            disabled = FALSE;
            baseregister = TTBR0_EL2;
            descaddr.memattrs = WalkAttrDecode(TCR_EL2.SH0, TCR_EL2.ORGN0, TCR_EL2.IRGN0, secondstage);
            reversedescriptors = SCTLR_EL2.EE == '1';
            lookupsecure = FALSE;
            singlepriv = TRUE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL2.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL2.HD == '1';
            hierattrsdisabled = AArch64.HaveHPDExt() && TCR_EL2.HPD == '1';
        else
```

```
            if inputaddr<top> == '0' then
                inputsize = 64 - UInt(TCR_EL1.T0SZ);
                largegrain = TCR_EL1.TG0 == '01';
                midgrain = TCR_EL1.TG0 == '10';
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - 39;
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<top:inputsize>);
                disabled = TCR_EL1.EPD0 == '1';
                baseregister = TTBR0_EL1;
                descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH0, TCR_EL1.ORGN0, TCR_EL1.IRGN0, secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD0 == '1';
            else
                inputsize = 64 - UInt(TCR_EL1.T1SZ);
                largegrain = TCR_EL1.TG1 == '11';       // TG1 and TG0 encodings differ
                midgrain = TCR_EL1.TG1 == '01';
                inputsize_max = if Have52BitVAExt() && largegrain then 52 else 48;
                inputsize_min = 64 - 39;
                if inputsize < inputsize_min then
                    c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
                    assert c IN {Constraint_FORCE, Constraint_FAULT};
                    if c == Constraint_FORCE then inputsize = inputsize_min;
                basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsOnes(inputaddr<top:inputsize>);
                disabled = TCR_EL1.EPD1 == '1';
                baseregister = TTBR1_EL1;
                descaddr.memattrs = WalkAttrDecode(TCR_EL1.SH1, TCR_EL1.ORGN1, TCR_EL1.IRGN1, secondstage);
                hierattrsdisabled = AArch64.HaveHPDExt() &&  TCR_EL1.HPD1 == '1';
            ps = TCR_EL1.IPS;
            reversedescriptors = SCTLR_EL1.EE == '1';
            lookupsecure = IsSecure();
            singlepriv = FALSE;
            update_AF = HaveAccessFlagUpdateExt() && TCR_EL1.HA == '1';
            update_AP = HaveDirtyBitModifierExt() && update_AF && TCR_EL1.HD == '1';
        if largegrain then
            grainsize = 16;                                             // Log2(64KB page size)
            firstblocklevel = 2;                                        // Largest block is 512MB (2^29 bytes)
        elsif midgrain then
            grainsize = 14;                                             // Log2(16KB page size)
            firstblocklevel = 2;                                        // Largest block is 32MB (2^25 bytes)
        else // Small grain
            grainsize = 12;                                             // Log2(4KB page size)
            firstblocklevel = 1;                                        // Largest block is 1GB (2^30 bytes)
        stride = grainsize - 3;                                         // Log2(page size / 8 bytes)
        // The starting level is the number of strides needed to consume the input address
        level = 4 - (1 + ((inputsize - grainsize - 1) DIV stride));

    else
        // Second stage translation
        inputaddr = ZeroExtend(ipaddress);
        inputsize = 64 - UInt(VTCR_EL2.T0SZ);
        largegrain = VTCR_EL2.TG0 == '01';
        midgrain = VTCR_EL2.TG0 == '10';

        inputsize_max = 48;
        if inputsize > inputsize_max then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_max;
        inputsize_min = 64 - 39;
        if inputsize < inputsize_min then
            c = ConstrainUnpredictable(Unpredictable_RESTnSZ);
            assert c IN {Constraint_FORCE, Constraint_FAULT};
            if c == Constraint_FORCE then inputsize = inputsize_min;
        ps = VTCR_EL2.PS;
        basefound = inputsize >= inputsize_min && inputsize <= inputsize_max && IsZero(inputaddr<63:inputsize>);
        disabled = FALSE;
        descaddr.memattrs = WalkAttrDecode(VTCR_EL2.SH0, VTCR_EL2.ORGN0, VTCR_EL2.IRGN0, secondstage);
        reversedescriptors = SCTLR_EL2.EE == '1';
        singlepriv = TRUE;
        update_AF = HaveAccessFlagUpdateExt() && VTCR_EL2.HA == '1';
        update_AP = HaveDirtyBitModifierExt() && update_AF && VTCR_EL2.HD == '1';

        lookupsecure = FALSE;
        baseregister = VTTBR_EL2;
        startlevel = UInt(VTCR_EL2.SL0);
        if largegrain then
            grainsize = 16;                                         // Log2(64KB page size)
            level = 3 - startlevel;
            firstblocklevel = 2;                                    // Largest block is 512MB (2^29 bytes)
        elsif midgrain then
            grainsize = 14;                                         // Log2(16KB page size)
            level = 3 - startlevel;
            firstblocklevel = 2;                                    // Largest block is 32MB (2^25 bytes)
        else // Small grain
            grainsize = 12;                                         // Log2(4KB page size)
            level = 2 - startlevel;
            firstblocklevel = 1;                                    // Largest block is 1GB (2^30 bytes)
        stride = grainsize - 3;                                     // Log2(page size / 8 bytes)

        // Limits on IPA controls based on implemented PA size. Level 0 is only
        // supported by small grain translations
        if largegrain then                              // 64KB pages
            // Level 1 only supported if implemented PA size is greater than 2^42 bytes
            if level == 0 || (level == 1 && PAMax() <= 42) then basefound = FALSE;
        elsif midgrain then                             // 16KB pages
            // Level 1 only supported if implemented PA size is greater than 2^40 bytes
            if level == 0 || (level == 1 && PAMax() <= 40) then basefound = FALSE;
        else                                            // Small grain, 4KB pages
            // Level 0 only supported if implemented PA size is greater than 2^42 bytes
            if level < 0 || (level == 0 && PAMax() <= 42) then basefound = FALSE;

        // If the inputsize exceeds the PAMax value, the behavior is CONSTRAINED UNPREDICTABLE
        inputsizecheck = inputsize;
        if inputsize > PAMax() && (!ELUsingAArch32(EL1) || inputsize > 40) then
            case ConstrainUnpredictable(Unpredictable_LARGEIPA) of
                when Constraint_FORCE
                    // Restrict the inputsize to the PAMax value
                    inputsize = PAMax();
                    inputsizecheck = PAMax();
                when Constraint_FORCENOSLCHECK
                    // As FORCE, except use the configured inputsize in the size checks below
                    inputsize = PAMax();
                when Constraint_FAULT
                    // Generate a translation fault
                    basefound = FALSE;
                otherwise
                    Unreachable();

        // Number of entries in the starting level table =
        //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
        startsizecheck = inputsizecheck - ((3 - level)*stride + grainsize); // Log2(Num of entries)

        // Check for starting level table with fewer than 2 entries or longer than 16 pages.
        // Lower bound check is:  startsizecheck < Log2(2 entries)
        // Upper bound check is:  startsizecheck > Log2(pagesize/8*16)
        if startsizecheck < 1 || startsizecheck > stride + 4 then basefound = FALSE;

    if !basefound || disabled then
        level = 0;           // AArch32 reports this as a level 1 fault
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress,   level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    case ps of
        when '000'  outputsize = 32;
        when '001'  outputsize = 36;
        when '010'  outputsize = 40;
        when '011'  outputsize = 42;
        when '100'  outputsize = 44;
        when '101'  outputsize = 48;
        otherwise   outputsize = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address
            ↪size value";

    if outputsize > PAMax() then outputsize = PAMax();

    if outputsize < 48 && !IsZero(baseregister<47:outputsize>) then
        level = 0;
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,  level, acctype, iswrite,
                                                         secondstage, s2fs1walk);
        return result;

    // Bottom bound of the Base address is:
    //     Log2(8 bytes per entry)+Log2(Number of entries in starting level table)
    // Number of entries in starting level table =
    //     (Size of Input Address)/((Address per level)^(Num levels remaining)*(Size of Table))
    baselowerbound = 3 + inputsize - ((3-level)*stride + grainsize);  // Log2(Num of entries*8)
    baseaddress = baseregister<47:baselowerbound>:Zeros(baselowerbound);

    ns_table = if lookupsecure then '0' else '1';
    ap_table = '00';
    xn_table = '0';
    pxn_table = '0';

    addrselecttop = inputsize - 1;

    repeat
        addrselectbottom = (3-level)*stride + grainsize;

        bits(48) index = ZeroExtend(inputaddr<addrselecttop:addrselectbottom>:'000');
        descaddr.paddress.address = baseaddress OR index;
        descaddr.paddress.NS = ns_table;

        // If there are two stages of translation, then the first stage table walk addresses
        // are themselves subject to translation
        if secondstage || !HasS2Translation() then
            descaddr2 = descaddr;
        else
            hwupdatewalk = FALSE;
            descaddr2 = AArch64.SecondStageWalk(descaddr, vaddress, acctype, iswrite, 8, hwupdatewalk);
            // Check for a fault on the stage 2 walk
            if IsFault(descaddr2) then
                result.addrdesc.fault = descaddr2.fault;
                return result;

        // Update virtual address for abort functions
        descaddr2.vaddress = ZeroExtend(vaddress);

        accdesc = CreateAccessDescriptorPTW(acctype, secondstage, s2fs1walk, level);
        desc = _Mem[descaddr2, 8, accdesc];

        if reversedescriptors then desc = BigEndianReverse(desc);

        if desc<0> == '0' || (desc<1:0> == '01' && level == 3) then
             // Fault (00), Reserved (10), or Block (01) at level 3.
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress,  level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

        // Valid Block, Page, or Table entry
        if desc<1:0> == '01' || level == 3 then                 // Block (01) or Page (11)
            blocktranslate = TRUE;
        else                                                    // Table (11)
            if outputsize != 48 && !IsZero(desc<47:outputsize>) then
                result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,  level, acctype,
                                                                 iswrite, secondstage, s2fs1walk);
                return result;

            baseaddress = desc<47:grainsize>:Zeros(grainsize);
            if !secondstage then
                // Unpack the upper and lower table attributes
                ns_table    = ns_table    OR desc<63>;
            if !secondstage && !hieratrsdisabled then
                ap_table<1> = ap_table<1> OR desc<62>;       // read-only

                xn_table    = xn_table    OR desc<60>;
                // pxn_table and ap_table[0] apply in EL1&0 or EL2&0 translation regimes
                if !singlepriv then
                    pxn_table   = pxn_table   OR desc<59>;
                    ap_table<0> = ap_table<0> OR desc<61>;   // privileged

            level = level + 1;
            addrselecttop = addrselectbottom - 1;
            blocktranslate = FALSE;
    until blocktranslate;

    // Check block size is supported at this level
    if level < firstblocklevel then
        result.addrdesc.fault = AArch64.TranslationFault(ipaddress,  level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Check for misprogramming of the contiguous bit
    if largegrain then
        num_ch_entries = 5;
    elsif midgrain then
        if level == 3 then
            num_ch_entries = 7;
        else num_ch_entries = 5;
    else num_ch_entries = 4;

    contiguousbitcheck = inputsize < (addrselectbottom + num_ch_entries);

    if contiguousbitcheck && desc<52> == '1' then
        if boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit" then
            result.addrdesc.fault = AArch64.TranslationFault(ipaddress,  level, acctype,
                                                             iswrite, secondstage, s2fs1walk);
            return result;

    // Unpack the descriptor into address and upper and lower block attributes
    outputaddress = desc<47:addrselectbottom>:inputaddr<addrselectbottom-1:0>;

    // Check the output address is inside the supported range
    if outputsize != 48 && !IsZero(desc<47:outputsize>) then
        result.addrdesc.fault = AArch64.AddressSizeFault(ipaddress,  level, acctype,
                                                         iswrite, secondstage, s2fs1walk);
        return result;

    // Check Access Flag
    if desc<10> == '0' then
        if !update_AF then
            result.addrdesc.fault = AArch64.AccessFlagFault(ipaddress,  level, acctype,
                                                            iswrite, secondstage, s2fs1walk);
            return result;
        else
            result.descupdate.AF = TRUE;

    if update_AP && desc<51> == '1' then
        // If hw update of access permission field is configured consider AP[2] as '0' / S2AP[2] as '1'
        if !secondstage && desc<7> == '1' then
            desc<7> = '0';
            result.descupdate.AP = TRUE;
        elsif secondstage && desc<7> == '0' then
            desc<7> = '1';
            result.descupdate.AP = TRUE;
    bits(4) ehwu = EffectiveHWU(PSTATE.EL, secondstage, vaddress<55>);
    bit current_cdbm = ehwu<0> AND desc<59>;
    bit current_sc = ehwu<1> AND desc<60>;
    if current_cdbm == '1' && current_sc == '0' then
        result.descupdate.SC = TRUE;
    // Required descriptor if AF, AP[2]/S2AP[2] or SC needs update
    result.descupdate.descaddr = descaddr;

    xn = desc<54>;                                            // Bit[54] of the block/page descriptor
         ↪holds UXN
    pxn = desc<53>;                                           // Bit[53] of the block/page descriptor
         ↪holds PXN
    ap = desc<7:6>:'1';                                       // Bits[7:6] of the block/page descriptor
         ↪hold AP[2:1]
    contiguousbit = desc<52>;
    nG = desc<11>;
    sh = desc<9:8>;
    memattr = desc<5:2>;                                      // AttrIndx and NS bit in stage 1

    result.domain = bits(4) UNKNOWN;                          // Domains not used
    result.level = level;
    result.blocksize = 2^((3-level)*stride + grainsize);

    // Stage 1 translation regimes also inherit attributes from the tables
    if !secondstage then
        result.perms.xn      = xn OR xn_table;
        result.perms.ap<2>   = ap<2> OR ap_table<1>;          // Force read-only
        // PXN, nG and AP[1] apply in EL1&0 or EL2&0 stage 1 translation regimes
        if !singlepriv then
            result.perms.ap<1> = ap<1> AND NOT(ap_table<0>);  // Force privileged only
            result.perms.pxn   = pxn OR pxn_table;
            // Pages from Non-secure tables are marked non-global in Secure EL1&0
            if IsSecure() then
                result.nG = nG OR ns_table;
            else
                result.nG = nG;
        else
            result.perms.ap<1> = '1';
            result.perms.pxn   = '0';
            result.nG          = '0';
        result.perms.ap<0>   = '1';
        result.addrdesc.memattrs = AArch64.S1AttrDecode(sh, memattr<2:0>, acctype);
        result.addrdesc.paddress.NS = memattr<3> OR ns_table;
    else
        result.perms.ap<2:1> = ap<2:1>;
        result.perms.ap<0>   = '1';
        result.perms.xn      = xn;
        if HaveExtendedExecuteNeverExt() then result.perms.xxn = desc<53>;
        result.perms.pxn     = '0';
        result.nG            = '0';
        if s2fs1walk then
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, AccType_PTW);
        else
            result.addrdesc.memattrs = S2AttrDecode(sh, memattr, acctype);
        result.addrdesc.paddress.NS = '1';

    // Read descriptor bits which control loads and stores of valid capabilities:
    // LC 62:61, SC 60, CDBM 59
    if secondstage then
        result.addrdesc.memattrs.readtagzero = (ehwu<2> AND desc<61>) == '0';
        result.addrdesc.memattrs.readtagfault = FALSE;
        result.addrdesc.memattrs.readtagfaulttgen = '0';
    else
        result.addrdesc.memattrs.readtagzero = (ehwu<3:2> AND desc<62:61>) == '00';
        result.addrdesc.memattrs.readtagfault = (ehwu<3> AND desc<62>) == '1';
        result.addrdesc.memattrs.readtagfaulttgen = NOT (ehwu<2> AND desc<61>);
    bit cdbm = ehwu<0> AND desc<59>;
    boolean writetagfault = (cdbm == '0') && (ehwu<1> AND desc<60>) == '0';
    result.addrdesc.memattrs.writetagfault = writetagfault;
    result.addrdesc.memattrs.iss2writetagfault = secondstage && writetagfault;

    result.addrdesc.paddress.address = outputaddress;
    result.addrdesc.fault = AArch64.NoFault();
    result.contiguous = contiguousbit == '1';
    if HaveCommonNotPrivateTransExt() then result.CnP = baseregister<0>;

    return result;
```

5.218 aarch64/translation/walk/EffectiveHWU

```
// EffectiveHWU()
// ==============
// Effective (V)TCR_ELx.HWU bits

bits(4) EffectiveHWU(bits(2) el, boolean secondstage, bit vaddr55)
    if secondstage then
        return VTCR_EL2.<HWU62,HWU61,HWU60,HWU59>;
    else
        regime = S1TranslationRegime(el);

        case regime of
            when EL1
                if vaddr55 == '1' then
                    if TCR_EL1.HPD1 == '1' then
                        return TCR_EL1.<HWU162,HWU161,HWU160,HWU159>;
                    else
                        return Zeros(4);
                elsif TCR_EL1.HPD0 == '1' then
                    return TCR_EL1.<HWU062,HWU061,HWU060,HWU059>;
                else
                    return Zeros(4);
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    if vaddr55 == '1' then
                        if TCR_EL2.HPD1 == '1' then
                            return TCR_EL2.<HWU162,HWU161,HWU160,HWU159>;
                        else
                            return Zeros(4);
                    elsif TCR_EL2.HPD0 == '1' then
                        return TCR_EL2.<HWU062,HWU061,HWU060,HWU059>;
                    else
                        return Zeros(4);
                else
                    if TCR_EL2.HPD == '1' then
                        return TCR_EL2.<HWU62,HWU61,HWU60,HWU59>;
                    else
                        return Zeros(4);
            when EL3
                if TCR_EL3.HPD == '1' then
                    return TCR_EL3.<HWU62,HWU61,HWU60,HWU59>;
                else
                    return Zeros(4);
```

5.219 shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// ===================

ClearStickyErrors()
    EDSCR.TXU = '0';            // Clear TX underrun flag
    EDSCR.RXO = '0';            // Clear RX overrun flag

    if Halted() then            // in Debug state
        EDSCR.ITO = '0';        // Clear ITR overrun flag

    // If halted and the ITR is not empty then it is UNPREDICTABLE whether the EDSCR.ERR is cleared.
    // The UNPREDICTABLE behavior also affects the instructions in flight, but this is not described
    // in the pseudocode.
    if Halted() && EDSCR.ITE == '0' && ConstrainUnpredictableBool(Unpredictable_CLEARERRITEZERO) then
        return;
    EDSCR.ERR = '0';            // Clear cumulative error flag

    return;
```

5.220 shared/debug/DebugTarget/DebugTarget

```
// DebugTarget()
// =============
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    secure = IsSecure();
    return DebugTargetFrom(secure);
```

5.221 shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =================

bits(2) DebugTargetFrom(boolean secure)
    if HaveEL(EL2) && !secure then
        route_to_el2 = (MDCR_EL2.TDE == '1' || HCR_EL2.TGE == '1');
    else
        route_to_el2 = FALSE;

    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && HighestELUsingAArch32() && secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

5.222 shared/debug/DoubleLockStatus/DoubleLockStatus

```
// DoubleLockStatus()
// ==================
// Returns the state of the OS Double Lock.
//    FALSE if OSDLR_EL1.DLK == 0 or DBGPRCR_EL1.CORENPDRQ == 1 or the PE is in Debug state.
//    TRUE if OSDLR_EL1.DLK == 1 and DBGPRCR_EL1.CORENPDRQ == 0 and the PE is in Non-debug state.

boolean DoubleLockStatus()
    if ELUsingAArch32(EL1) then
        Unreachable();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

5.223 shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// ==========================
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if ExternalInvasiveDebugEnabled() then
        if ExternalSecureInvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return MDCR_EL3.EDAD == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

5.224 shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// ========================
// Returns TRUE if an external debug interface access to the PMU registers is allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    if ExternalNoninvasiveDebugEnabled() then
        if ExternalSecureNoninvasiveDebugEnabled() then
            return TRUE;
        elsif HaveEL(EL3) then
            return MDCR_EL3.EPMAD == '0';
        else
            return !IsSecure();
    else
        return FALSE;
```

5.225 shared/debug/authentication/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

5.226 shared/debug/authentication/ExternalInvasiveDebugEnabled

```
// ExternalInvasiveDebugEnabled()
// ==============================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the DBGEN signal.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == HIGH;
```

5.227 shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =================================
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return (ExternalNoninvasiveDebugEnabled() &&
            (!IsSecure() || ExternalSecureNoninvasiveDebugEnabled()));
```

5.228 shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, ExternalNoninvasiveDebugEnabled returns the state of the (DBGEN
// OR NIDEN) signal.

boolean ExternalNoninvasiveDebugEnabled()
    return ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

5.229 shared/debug/authentication/ExternalSecureInvasiveDebugEnabled

```
// ExternalSecureInvasiveDebugEnabled()
// ====================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN AND SPIDEN) signal.
// CoreSight allows asserting SPIDEN without also asserting DBGEN, but this is not recommended.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == HIGH;
```

5.230 shared/debug/authentication/ExternalSecureNoninvasiveDebugEnabled

```
// ExternalSecureNoninvasiveDebugEnabled()
// =======================================
// The definition of this function is IMPLEMENTATION DEFINED.
// In the recommended interface, this function returns the state of the (DBGEN OR NIDEN) AND
// (SPIDEN OR SPNIDEN) signal.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveEL(EL3) && !IsSecure() then return FALSE;
    return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
```

5.231 shared/debug/authentication/IsCorePowered

```
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();
```

5.232 shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// ======================
// Checks for an invalid state match that will generate Constrained Unpredictable behaviour, otherwise
// returns Constraint_NONE.

(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC, bit HMC, bits(2) PxC, boolean
    ↪isbreakpnt)
    boolean reserved = FALSE;

    // Match 'Usr/Sys/Svc' only valid for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && HMC:PxC == '000' && SSC != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (HMC != '0' || SSC != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && SSC IN {'01','10'} && HMC:SSC:PxC != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || HighestELUsingAArch32()) && HMC:SSC:PxC == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && HMC:SSC:PxC == '11100' then
        reserved = TRUE;

    // Values that are not allocated in any architecture version
    if (HMC:SSC:PxC) IN {'01110','100x0','10110','11x10'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        (c, <HMC,SSC,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, SSC, HMC, PxC);
```

5.233 shared/debug/cti/CTI_SetEventLevel

```
// Set a Cross Trigger multi-cycle input event trigger to the specified level.
CTI_SetEventLevel(CrossTriggerIn id, signal level);
```

5.234 shared/debug/cti/CTI_SignalEvent

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

5.235 shared/debug/cti/CrossTrigger

```
enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,          CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,  CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,  CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,     CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,         CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,  CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,  CrossTriggerIn_TraceExtOut3};
```

5.236 shared/debug/dccanditr/CDBGDTR_EL0

```
// CDBGDTR_EL0[] (write)
// =====================
// System register writes to CDBGDTR_EL0

CDBGDTR_EL0[] = bits(129) value
    // For MSR CDBGDTR_EL0,<Ct>
    if EDSCR.TXfull == '1' then
        value = bits(129) UNKNOWN;
    EDSCR2.DTRTAG = value<128>;
    DBGDTR2B = value<127:96>;
    DBGDTR2A = value<95:64>;
    DTRRX = value<63:32>;
    DTRTX = value<31:0>;

    EDSCR.TXfull = '1';
    return;

// CDBGDTR_EL0[] (read)
// ====================
// System register reads of CDBGDTR_EL0

bits(129) CDBGDTR_EL0[]
    // For MRS <Ct>,CDBGDTR_EL0
    bits(129) result;
    if EDSCR.RXfull == '0' then
        result = Capability UNKNOWN;
    else
```

```
        // NOTE: the word order is reversed on reads with regards to writes
        result<63:32> = DTRTX;
        result<31:0> = DTRRX;
        result<95:64> = DBGDTR2A;
        result<127:96> = DBGDTR2B;
        result<128> = EDSCR2.DTRTAG;
    EDSCR.RXfull = '0';
    return result;
```

5.237 shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =======================

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then HIGH else LOW);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

    // The value to be driven onto the common COMMIRQ signal.
    commirq = ((commrx && MDCCINT_EL1.RX == '1') ||
               (commtx && MDCCINT_EL1.TX == '1'));
    SetInterruptRequestLevel(InterruptID_COMMIRQ, if commirq then HIGH else LOW);

    return;
```

5.238 shared/debug/dccanditr/DBGDTRRX_EL0

```
// DBGDTRRX_EL0[] (external write)
// ===============================
// Called on writes to debug register 0x08C.

DBGDTRRX_EL0[boolean memory_mapped] = bits(32) value

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1';  EDSCR.ERR = '1';              // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                            // See comments in EDITR[] (external write)
        ExecuteA64(0xD5330501<31:0>);               // A64 "MRS X1,DBGDTRRX_EL0"
        ExecuteA64(0xB8004401<31:0>);               // A64 "STR W1,[X0],#4"
        X[1] = bits(64) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(32) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls DBGDTR_EL0[] (read) which clears RXfull.
            assert EDSCR.RXfull == '0';

        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_EL0[] (external read)
// ==============================

bits(32) DBGDTRRX_EL0[boolean memory_mapped]
    return DTRRX;
```

5.239 shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// ==============================
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;              // Error flag set: no side-effects

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then           // Software lock locked: no side-effects
        return value;

    if underrun then
        EDSCR.TXU = '1';  EDSCR.ERR = '1';              // Underrun condition: block side-effects
        return value;                                   // Return UNKNOWN

    EDSCR.TXfull = '0';
    if Halted() && EDSCR.MA == '1' then
        EDSCR.ITE = '0';                                // See comments in EDITR[] (external write)

        if !UsingAArch32() then
            ExecuteA64(0xB8404401<31:0>);               // A64 "LDR W1,[X0],#4"
        else
            ExecuteT32(0xF850<15:0> /*hw1*/, 0x1B04<15:0> /*hw2*/);      // T32 "LDR R1,[R0],#4"
        // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.TXfull = bit UNKNOWN;
            DBGDTRTX_EL0 = bits(32) UNKNOWN;
        else
            if !UsingAArch32() then
                ExecuteA64(0xD5130501<31:0>);           // A64 "MSR DBGDTRTX_EL0,X1"
            else
                ExecuteT32(0xEE00<15:0> /*hw1*/, 0x1E15<15:0> /*hw2*/);  // T32 "MSR DBGDTRTXint,R1"
            // "MSR DBGDTRTX_EL0,X1" calls DBGDTR_EL0[] (write) which sets TXfull.
            assert EDSCR.TXfull == '1';
        X[1] = bits(64) UNKNOWN;
        EDSCR.ITE = '1';                                // See comments in EDITR[] (external write)

    return value;

// DBGDTRTX_EL0[] (external write)
// ===============================

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write
    DTRTX = value;
    return;
```

5.240 shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// ====================
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

DBGDTR_EL0[] = bits(N) value
    // For MSR DBGDTRTX_EL0,<Rt>  N=32, value=X[t]<31:0>, X[t]<63:32> is ignored
    // For MSR DBGDTR_EL0,<Xt>    N=64, value=X[t]<63:0>
    assert N IN {32,64};
    if EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>;        // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;

// DBGDTR_EL0[] (read)
// ===================
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
    // For MRS <Rt>,DBGDTRTX_EL0  N=32, X[t]=Zeros(32):result
    // For MRS <Xt>,DBGDTR_EL0    N=64, X[t]=result
    assert N IN {32,64};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    else
        // On a 64-bit read, implement a half-duplex channel
        // NOTE: the word order is reversed on reads with regards to writes
        if N == 64 then result<63:32> = DTRTX;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

5.241 shared/debug/dccanditr/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

5.242 shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// ========================
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return;

    if EDSCR.ERR == '1' then return;                    // Error flag set: ignore write

    // The Software lock is OPTIONAL.
    if memory_mapped && EDLSR.SLK == '1' then return;   // Software lock locked: ignore write

    if !Halted() then return;                           // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1';  EDSCR.ERR = '1';              // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // may support multiple outstanding instructions. Unlike the "InstrCompl" flag in [v7A] there
    // is no indication that the pipeline is empty (all instructions have completed). In this
    // pseudocode, the assumption is that only one instruction can be executed at a time,
    // meaning ITE acts like "InstrCompl".
    EDSCR.ITE = '0';

    if !UsingAArch32() then
        ExecuteA64(value);
    else
        ExecuteT32(value<15:0>/*hw1*/, value<31:16> /*hw2*/);

    EDSCR.ITE = '1';

    return;
```

5.243 shared/debug/halting/DCPSInstruction

```
// DCPSInstruction()
// =================
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

    case target_el of
```

```
        when EL1
            if PSTATE.EL == EL2 || (PSTATE.EL == EL3 && !UsingAArch32()) then handle_el = PSTATE.EL;
            elsif EL2Enabled() && HCR_EL2.TGE == '1' then UNDEFINED;
            else handle_el = EL1;

        when EL2
            if !HaveEL(EL2) then UNDEFINED;
            elsif PSTATE.EL == EL3 && !UsingAArch32() then handle_el = EL3;
            elsif IsSecure() then UNDEFINED;
            else handle_el = EL2;
        when EL3
            if EDSCR.SDD == '1' || !HaveEL(EL3) then UNDEFINED;
            handle_el = EL3;
        otherwise
            Unreachable();

    from_secure = IsSecure();
    PSTATE.nRW = '0';  PSTATE.SP = '1';  PSTATE.EL = handle_el;
    if (HavePANExt() && ((handle_el == EL1 && SCTLR_EL1.SPAN == '0') ||
                         (handle_el == EL2 && HCR_EL2.E2H == '1' &&
                          HCR_EL2.TGE == '1' && SCTLR_EL2.SPAN == '0'))) then
        PSTATE.PAN = '1';
    ELR[] = bits(64) UNKNOWN;  SPSR[] = bits(32) UNKNOWN;  ESR[] = bits(32) UNKNOWN;
    if !HaveCapabilitiesExt() then
        DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;
    if HaveUAOExt() then PSTATE.UAO = '0';
    if HaveCapabilitiesExt() then PSTATE.C64 = CCTLR[].C64E;

    UpdateEDSCRFields();                        // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;
```

5.244 shared/debug/halting/DRPSInstruction

```
// DRPSInstruction()
// =================
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SetPSTATEFromPSR(SPSR[]);

    // PSTATE.{N,Z,C,V,Q,GE,SS,D,A,I,F} are not observable and ignored in Debug state, so
    // behave as if UNKNOWN.
    if UsingAArch32() then
        PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';  PSTATE.T = '1';        // PSTATE.J is RES0
        DLR = bits(32) UNKNOWN;  DSPSR = bits(32) UNKNOWN;
    else
        PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
        if !HaveCapabilitiesExt() then
            DLR_EL0 = bits(64) UNKNOWN;
        DSPSR_EL0 = bits(32) UNKNOWN;

    UpdateEDSCRFields();                                // Update EDSCR PE state flags

    return;
```

5.245 shared/debug/halting/DebugHalt

```
constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ          = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';
```

5.246 shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
DisableITRAndResumeInstructionPrefetch();
```

5.247 shared/debug/halting/ExecuteA64

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
```

5.248 shared/debug/halting/ExecuteT32

```
// Execute a T32 instruction in Debug state.
ExecuteT32(bits(16) hw1, bits(16) hw2);
```

5.249 shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// ================

ExitDebugState()
    assert Halted();
    SynchronizeContext();

    // Although EDSCR.STATUS signals that the PE is restarting, debuggers must use EDPRSR.SDR to
    // detect that the PE has restarted.
    EDSCR.STATUS = '000001';                           // Signal restarting
    EDESR<2:0> = '000';                                // Clear any pending Halting debug events

    bits(64) new_pc;
    bits(32) spsr;

    Capability new_pcc = CDLR_EL0;
    spsr = DSPSR_EL0;
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr);                            // Can update privileged bits, even at EL0

    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable_RESTARTALIGNPC) then new_pc<0> = '0';
        BranchTo(new_pc<31:0>, BranchType_DBGEXIT);    // AArch32 branch
    else
        // If targeting AArch32 then possibly zero the 32 most significant bits of the target PC
        if spsr<4> == '1' && ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
            new_pc<63:32> = Zeros();
        BranchToCapability(new_pcc, BranchType_DBGEXIT);

    (EDSCR.STATUS,EDPRSR.SDR) = ('000010','1');        // Atomically signal restarted
    UpdateEDSCRFields();                               // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```

5.250 shared/debug/halting/Halt

```
// Halt()
// ======

Halt(bits(6) reason)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt);  // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    Capability preferred_restart_cap = PCC[];
    spsr = GetPSRFromPSTATE();

    if UsingAArch32() then
        spsr<21> = PSTATE.SS;                    // Always save the SS bit

    CDLR_EL0 = preferred_restart_cap;
    DSPSR_EL0 = spsr;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if IsSecure() then
        EDSCR.SDD = '0';                        // If entered in Secure state, allow debug
    elsif HaveEL(EL3) then
        EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
    else
        assert EDSCR.SDD == '1';                // Otherwise EDSCR.SDD is RES1
    EDSCR.MA = '0';

    // PSTATE.{SS,D,A,I,F} are not observable and ignored in Debug state, so behave as if
    // UNKNOWN. PSTATE.{N,Z,C,V,Q,GE} are also not observable, but since these are not changed on
    // exception entry, this function also leaves them unchanged. PSTATE.{E,M,nRW,EL,SP} are
    // unchanged. PSTATE.IL is set to 0.
    if UsingAArch32() then
        PSTATE.<SS,A,I,F> = bits(4) UNKNOWN;
        //  In AArch32, all instructions are T32 and unconditional.
        PSTATE.IT = '00000000';
        PSTATE.T = '1';                         // PSTATE.J is RES0
    else
        PSTATE.<SS,D,A,I,F> = bits(5) UNKNOWN;
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason;                      // Signal entered Debug state
    UpdateEDSCRFields();                        // Update EDSCR PE state flags.

    return;
```

5.251 shared/debug/halting/HaltOnBreakpointOrWatchpoint

```
// HaltOnBreakpointOrWatchpoint()
// ==============================
// Returns TRUE if the Breakpoint and Watchpoint debug events should be considered for Debug
// state entry, FALSE if they should be considered for a debug exception.

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';
```

5.252 shared/debug/halting/Halted

```
// Halted()
// ========

boolean Halted()
    return !(EDSCR.STATUS IN {'000001', '000010'});                     // Halted
```

5.253 shared/debug/halting/HaltingAllowed

```
// HaltingAllowed()
// ================
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.

boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
```

```
        return FALSE;
    elsif IsSecure() then
        return ExternalSecureInvasiveDebugEnabled();
    else
        return ExternalInvasiveDebugEnabled();
```

5.254 shared/debug/halting/Restarting

```
// Restarting()
// ============

boolean Restarting()
    return EDSCR.STATUS == '000001';                                     // Restarting
```

5.255 shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

5.256 shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// ===================
// Update EDSCR PE state fields

UpdateEDSCRFields()

    if !Halted() then
        EDSCR.EL = '00';
        EDSCR.NS = bit UNKNOWN;
        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        EDSCR.NS = if IsSecure() then '0' else '1';

        bits(4) RW;
        RW<1> = if ELUsingAArch32(EL1) then '0' else '1';
        if PSTATE.EL != EL0 then
            RW<0> = RW<1>;
        else
            RW<0> = if UsingAArch32() then '0' else '1';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_GEN[].NS == '0') then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

5.257 shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====================
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // Called after an exception entry or exit, that is, such that IsSecure() and PSTATE.EL are correct
    // for the exception target.
    base = if IsSecure() then 0 else 4;
    if HaltingAllowed() then
        if HaveExtendedECDebugEvents() then
```

```
            exception_exit = !exception_entry;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + 8>:EDECCR<UInt(PSTATE.EL) + base>;
            case ctrl of
                when '00'  halt = FALSE;
                when '01'  halt = TRUE;
                when '10'  halt = (exception_exit == TRUE);
                when '11'  halt = (exception_entry == TRUE);
        else
            halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');
        if halt then Halt(DebugHalt_ExceptionCatch);
```

5.258 shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// ==================
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep()
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive);
        else
            Halt(DebugHalt_Step_Normal);
```

5.259 shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// ====================
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if EDECR.OSUCE == '1' then
        if !Halted() then EDESR.OSUC = '1';
```

5.260 shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// ===========================
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        Halt(DebugHalt_OSUnlockCatch);
```

5.261 shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// ========================
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        Halt(DebugHalt_ResetCatch);
```

5.262 shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =================
// Called after reset

CheckResetCatch()
    if EDECR.RCE == '1' then
```

```
        EDESR.RC = '1';
        // If halting is allowed then halt immediately
        if HaltingAllowed() then Halt(DebugHalt_ResetCatch);
```

5.263 shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====================================
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = OSLSR_EL1.OSLK;
    if HaltingAllowed() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

5.264 shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// ======================

ExternalDebugRequest()
    if HaltingAllowed() then
        Halt(DebugHalt_EDBGRQ);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

5.265 shared/debug/haltingevents/HaltingStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
boolean HaltingStep_DidNotStep();
```

5.266 shared/debug/haltingevents/HaltingStep_SteppedEX

```
// Returns TRUE if the previously executed instruction was a Load-Exclusive class instruction
// executed in the active-not-pending state.
boolean HaltingStep_SteppedEX();
```

5.267 shared/debug/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// ================

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();             // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then                     // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

5.268 shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =================================
// Determine whether EDSCR disables interrupts routed to 'target'

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    case target of
        when EL3
            int_dis = EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled();
        when EL2
            int_dis = EDSCR.INTdis == '1x' && ExternalInvasiveDebugEnabled();
        when EL1
            if IsSecure() then
                int_dis = EDSCR.INTdis == '1x' && ExternalSecureInvasiveDebugEnabled();
            else
                int_dis = EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled();
    return int_dis;
```

5.269 shared/debug/interrupts/InterruptID

```
enumeration InterruptID {InterruptID_PMUIRQ, InterruptID_COMMIRQ, InterruptID_CTIIRQ,
                         InterruptID_COMMRX, InterruptID_COMMTX};
```

5.270 shared/debug/interrupts/SetInterruptRequestLevel

```
// Set a level-sensitive interrupt to the specified level.
SetInterruptRequestLevel(InterruptID id, signal level);
```

5.271 shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// ================

CreatePCSample()
    // In a simple sequential execution of the program, CreatePCSample is executed each time the PE
    // executes an instruction that can be sampled. An implementation is not constrained such that
    // reads of EDPCSRlo return the current values of PC, etc.

    pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
    pc_sample.pc = ThisInstrAddr();
    pc_sample.el = PSTATE.EL;
    pc_sample.rw = if UsingAArch32() then '0' else '1';
    pc_sample.ns = if IsSecure() then '0' else '1';
    pc_sample.contextidr = CONTEXTIDR_EL1;
    pc_sample.has_el2 = EL2Enabled();

    if EL2Enabled() then
        pc_sample.vmid = VTTBR_EL2.VMID;
        pc_sample.contextidr_el2 = CONTEXTIDR_EL2;
        pc_sample.el0h = FALSE;
    return;
```

5.272 shared/debug/samplebasedprofiling/EDPCSRlo

```
// EDPCSRlo[] (read)
// =================

bits(32) EDPCSRlo[boolean memory_mapped]

    sample = bits(32) UNKNOWN;

    return sample;
```

5.273 shared/debug/samplebasedprofiling/PCSample

```
type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    bit ns,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)

PCSample pc_sample;
```

5.274 shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// ===============

bits(32) PMPCSR[boolean memory_mapped]

    if EDPRSR<6:5,0> != '001' then                      // Check DLK, OSLK and PU bits
        IMPLEMENTATION_DEFINED "generate error response";
        return bits(32) UNKNOWN;

    // The Software lock is OPTIONAL.
    update = !memory_mapped || PMLSR.SLK == '0';        // Software locked: no side-effects

    if pc_sample.valid then
        sample = pc_sample.pc<31:0>;
        if update then
            PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            PMPCSR.EL = pc_sample.el;
            PMPCSR.NS = pc_sample.ns;

            PMCID1SR = pc_sample.contextidr;
            PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

            PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0} && !pc_sample.el0h
                            then pc_sample.vmid else bits(16) UNKNOWN);
    else
        sample = Ones(32);
        if update then
            PMPCSR<55:32>  = bits(24) UNKNOWN;
            PMPCSR.EL = bits(2) UNKNOWN;
            PMPCSR.NS = bit UNKNOWN;

            PMCID1SR = bits(32) UNKNOWN;
            PMCID2SR = bits(32) UNKNOWN;

            PMVIDSR.VMID = bits(16) UNKNOWN;

    return sample;
```

5.275 shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// ===================
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // Other self-hosted debug functions will call AArch32.GenerateDebugExceptions() if called from
    // AArch32 state. However, because Software Step is only active when the debug target Exception
    // level is using AArch64, CheckSoftwareStep only calls AArch64.GenerateDebugExceptions().
    if !ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions() then
        if MDSCR_EL1.SS == '1' && PSTATE.SS == '0' then
            AArch64.SoftwareStepException();
```

5.276 shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// ========================
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(32) spsr)
    assert Halted() || Restarting() ||  PSTATE.EL != EL0;

    SS_bit = '0';

    if MDSCR_EL1.SS == '1' then
        if Restarting() then
            enabled_at_source = FALSE;
        else
            enabled_at_source = AArch64.GenerateDebugExceptions();

        if IllegalExceptionReturn(spsr) then
            dest = PSTATE.EL;
        else
            (valid, dest) = ELFromSPSR(spsr);  assert valid;

        secure = IsSecureBelowEL3() || dest == EL3;
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest, secure, mask);
        ELd = DebugTargetFrom(secure);
        if !ELUsingAArch32(ELd) && !enabled_at_source && enabled_at_dest then
            SS_bit = spsr<21>;
    return SS_bit;
```

5.277 shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// ===========
// Advance the Software Step state machine.

SSAdvance()

    // A simpler implementation of this function just clears PSTATE.SS to zero regardless of the
    // current Software Step state machine. However, this check is made to illustrate that the
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    target = DebugTarget();
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';

    if active_not_pending then PSTATE.SS = '0';

    return;
```

5.278 shared/debug/softwarestep/SoftwareStep_DidNotStep

```
// Returns TRUE if the previously executed instruction was executed in the inactive state, that is,
// if it was not itself stepped.
// Might return TRUE or FALSE if the previously executed instruction was an ISB or ERET executed
// in the active-not-pending state, or if another exception was taken before the Software Step exception.
// Returns FALSE otherwise, indicating that the previously executed instruction was executed in the
// active-not-pending state, that is, the instruction was stepped.
boolean SoftwareStep_DidNotStep();
```

5.279 shared/debug/softwarestep/SoftwareStep_SteppedEX

```
// Returns a value that describes the previously executed instruction. The result is valid only if
// SoftwareStep_DidNotStep() returns FALSE.
// Might return TRUE or FALSE if the instruction was an AArch32 LDREX that failed its condition code test.
// Otherwise returns TRUE if the instruction was a Load-Exclusive class instruction, and FALSE if the
// instruction was not a Load-Exclusive class instruction.
boolean SoftwareStep_SteppedEX();
```

5.280 shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// ===================
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32() then
        cond = AArch32.CurrentCond();
        if PSTATE.T == '0' then             // A32
            syndrome<4> = '1';
            // A conditional A32 instruction that is known to pass its condition code check
            // can be presented either with COND set to 0xE, the value for unconditional, or
            // the COND value held in the instruction.
            if ConditionHolds(cond) && ConstrainUnpredictableBool(Unpredictable_ESRCONDPASS) then
                syndrome<3:0> = '1110';
            else
                syndrome<3:0> = cond;
        else                                // T32
            // When a T32 instruction is trapped, it is IMPLEMENTATION DEFINED whether:
            //  * CV set to 0 and COND is set to an UNKNOWN value
            //  * CV set to 1 and COND is set to the condition code for the condition that
            //    applied to the instruction.
            if boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';

    return syndrome;
```

5.281 shared/exceptions/exceptions/Exception

```
enumeration Exception {Exception_Uncategorized,       // Uncategorized or unknown reason
                       Exception_WFxTrap,             // Trapped WFI or WFE instruction
                       Exception_CP15RTTrap,          // Trapped AArch32 MCR or MRC access to CP15
                       Exception_CP15RRTTrap,         // Trapped AArch32 MCRR or MRRC access to CP15
                       Exception_CP14RTTrap,          // Trapped AArch32 MCR or MRC access to CP14
                       Exception_CP14DTTrap,          // Trapped AArch32 LDC or STC access to CP14
                       Exception_AdvSIMDFPAccessTrap, // HCPTR-trapped access to SIMD or FP
                       Exception_FPIDTrap,            // Trapped access to SIMD or FP ID register
                       // Trapped BXJ instruction not supported in Armv8
                       Exception_CP14RRTTrap,         // Trapped MRRC access to CP14 from AArch32
                       Exception_IllegalState,        // Illegal Execution state
                       Exception_SupervisorCall,      // Supervisor Call
                       Exception_HypervisorCall,      // Hypervisor Call
                       Exception_MonitorCall,         // Monitor Call or Trapped SMC instruction
                       Exception_SystemRegisterTrap,  // Trapped MRS or MSR system register access
                       Exception_InstructionAbort,    // Instruction Abort or Prefetch Abort
                       Exception_PCAlignment,         // PC alignment fault
                       Exception_DataAbort,           // Data Abort
                       Exception_SPAlignment,         // SP alignment fault
                       Exception_FPTrappedException,  // IEEE trapped FP exception
                       Exception_SError,              // SError interrupt
                       Exception_Breakpoint,          // (Hardware) Breakpoint
                       Exception_SoftwareStep,        // Software Step
                       Exception_Watchpoint,          // Watchpoint
                       Exception_SoftwareBreakpoint,  // Software Breakpoint Instruction
                       Exception_VectorCatch,         // AArch32 Vector Catch
                       Exception_IRQ,                 // IRQ interrupt
                       Exception_CapabilitySysRegTrap,// Trapped MRS or MSR access to Capability System
                            ↪register
                       Exception_CapabilityAccess,    // Trapped access to Capability functionality
                       Exception_FIQ};                  // FIQ interrupt
```

5.282 shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (Exception exceptype,          // Exception class
                         bits(25)  syndrome,           // Syndrome record
                         bits(64)  vaddress,           // Virtual fault address
                         boolean   ipavalid,           // Physical fault address for second stage faults is valid
                         bits(48)  ipaddress)          // Physical fault address for second stage faults
```

5.283 shared/exceptions/exceptions/ExceptionSyndrome

```
// ExceptionSyndrome()
// ===================
// Return a blank exception syndrome record for an exception of the given type.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    r.exceptype = exceptype;

    // Initialize all other fields
    r.syndrome = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.ipaddress = Zeros();

    return r;
```

5.284 shared/exceptions/traps/ReservedValue

```
// ReservedValue()
// ===============

ReservedValue()
    AArch64.UndefinedFault();
```

5.285 shared/exceptions/traps/UnallocatedEncoding

```
// UnallocatedEncoding()
// =====================

UnallocatedEncoding()
    AArch64.UndefinedFault();
```

5.286 shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =============
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)

    bits(6) result;
    case statuscode of
        when Fault_AddressSize         result = '0000':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AccessFlag          result = '0010':level<1:0>; assert level IN {1,2,3};
        when Fault_Permission          result = '0011':level<1:0>; assert level IN {1,2,3};
        when Fault_Translation         result = '0001':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncExternal        result = '010000';
        when Fault_SyncExternalOnWalk  result = '0101':level<1:0>; assert level IN {0,1,2,3};
        when Fault_SyncParity          result = '011000';
        when Fault_SyncParityOnWalk    result = '0111':level<1:0>; assert level IN {0,1,2,3};
        when Fault_AsyncParity         result = '011001';
        when Fault_AsyncExternal       result = '010001';
        when Fault_Alignment           result = '100001';
        when Fault_Debug               result = '100010';
        when Fault_TLBConflict         result = '110000';
        when Fault_HWUpdateAccessFlag  result = '110001';
        when Fault_CapTag              result = '101000';
        when Fault_CapSeal              result = '101001';
        when Fault_CapBounds            result = '101010';
        when Fault_CapPerm              result = '101011';
        when Fault_CapPagePerm          result = '101100';
        when Fault_Lockdown             result = '110100';  // IMPLEMENTATION DEFINED
        when Fault_Exclusive            result = '110101';  // IMPLEMENTATION DEFINED
        otherwise                       Unreachable();

    return result;
```

5.287 shared/functions/aborts/IPAValid

```
// IPAValid()
// ==========
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    if fault.s2fs1walk then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Permission, Fault_Translation,
                                    Fault_AddressSize};
    elsif fault.secondstage then
        return fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_AddressSize};
    else
        return FALSE;
```

5.288 shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// ==============
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsAsyncAbort()
// ==============

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

5.289 shared/functions/aborts/IsDebugException

```
// IsDebugException()
// ==================

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault_None;
    return fault.statuscode == Fault_Debug;
```

5.290 shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =================
// Returns TRUE if the abort currently being processed is an external abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk,
         ↪Fault_SyncParityOnWalk,
                           Fault_AsyncExternal, Fault_AsyncParity });

// IsExternalAbort()
// =================

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode);
```

5.291 shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====================
// Returns TRUE if the abort currently being processed is an external synchronous abort and FALSE
    ↪otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_SyncExternal, Fault_SyncParity, Fault_SyncExternalOnWalk,
        ↪Fault_SyncParityOnWalk});

// IsExternalSyncAbort()
// =====================

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode);
```

5.292 shared/functions/aborts/IsFault

```
// IsFault()
// =========
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault_None;
```

5.293 shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// ===================
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault_None;

    return (statuscode IN {Fault_AsyncExternal, Fault_AsyncParity});

// IsSErrorInterrupt()
// ===================

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);
```

5.294 shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// ===============

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault_None;

    return fault.secondstage;
```

5.295 shared/functions/aborts/LSInstructionSyndrome

```
bits(11) LSInstructionSyndrome();
```

5.296 shared/functions/capability/CAP_BASE_EXP_HI_BIT

```
constant integer CAP_BASE_EXP_HI_BIT        = 66;
```

5.297 shared/functions/capability/CAP_BASE_HI_BIT

```
constant integer CAP_BASE_HI_BIT            = 79;
```

5.298 shared/functions/capability/CAP_BASE_LO_BIT

```
constant integer CAP_BASE_LO_BIT            = 64;
```

5.299 shared/functions/capability/CAP_BASE_MANTISSA_LO_BIT

```
constant integer CAP_BASE_MANTISSA_LO_BIT   = 67;
```

5.300 shared/functions/capability/CAP_BASE_MANTISSA_NUM_BITS

```
constant integer CAP_BASE_MANTISSA_NUM_BITS = CAP_BASE_HI_BIT-CAP_BASE_MANTISSA_LO_BIT+1;
```

5.301 shared/functions/capability/CAP_BOUND_MAX

```
constant bits(CAP_BOUND_NUM_BITS) CAP_BOUND_MAX = (1<<CAP_VALUE_NUM_BITS)<0+:CAP_BOUND_NUM_BITS>;
```

5.302 shared/functions/capability/CAP_BOUND_MIN

```
constant bits(CAP_BOUND_NUM_BITS) CAP_BOUND_MIN = 0x0<0+:CAP_BOUND_NUM_BITS>;
```

5.303 shared/functions/capability/CAP_BOUND_NUM_BITS

```
constant integer CAP_BOUND_NUM_BITS  = CAP_VALUE_NUM_BITS+1;
```

5.304 shared/functions/capability/CAP_FLAGS_HI_BIT

```
constant integer CAP_FLAGS_HI_BIT           = 63;
```

5.305 shared/functions/capability/CAP_FLAGS_LO_BIT

```
constant integer CAP_FLAGS_LO_BIT           = 56;
```

5.306 shared/functions/capability/CAP_IE_BIT

```
constant integer CAP_IE_BIT        = 94;
```

5.307 shared/functions/capability/CAP_LENGTH_NUM_BITS

```
constant integer CAP_LENGTH_NUM_BITS = CAP_VALUE_NUM_BITS+1;
```

5.308 shared/functions/capability/CAP_LIMIT_EXP_HI_BIT

```
constant integer CAP_LIMIT_EXP_HI_BIT       = 82;
```

5.309 shared/functions/capability/CAP_LIMIT_HI_BIT

```
constant integer CAP_LIMIT_HI_BIT           = 93;
```

5.310 shared/functions/capability/CAP_LIMIT_LO_BIT

```
constant integer CAP_LIMIT_LO_BIT           = 80;
```

5.311 shared/functions/capability/CAP_LIMIT_MANTISSA_LO_BIT

```
constant integer CAP_LIMIT_MANTISSA_LO_BIT  = 83;
```

5.312 shared/functions/capability/CAP_LIMIT_MANTISSA_NUM_BITS

```
constant integer CAP_LIMIT_MANTISSA_NUM_BITS = CAP_LIMIT_HI_BIT-CAP_LIMIT_MANTISSA_LO_BIT+1;
```

5.313 shared/functions/capability/CAP_LIMIT_NUM_BITS

```
constant integer CAP_LIMIT_NUM_BITS = CAP_LIMIT_HI_BIT-CAP_LIMIT_LO_BIT+1;
```

5.314 shared/functions/capability/CAP_MAX_ENCODEABLE_EXPONENT

```
constant integer CAP_MAX_ENCODEABLE_EXPONENT = 63;
```

5.315 shared/functions/capability/CAP_MAX_EXPONENT

```
constant integer CAP_MAX_EXPONENT = CAP_VALUE_NUM_BITS-CAP_MW+2;
```

5.316 shared/functions/capability/CAP_MAX_FIXED_SEAL_TYPE

```
constant integer    CAP_MAX_FIXED_SEAL_TYPE = 3;
```

5.317 shared/functions/capability/CAP_MAX_OBJECT_TYPE

```
constant integer CAP_MAX_OBJECT_TYPE = (1<<CAP_OTYPE_NUM_BITS)-1;
```

5.318 shared/functions/capability/CAP_MW

```
constant integer CAP_MW = CAP_BASE_HI_BIT-CAP_BASE_LO_BIT+1;
```

5.319 shared/functions/capability/CAP_NO_SEALING

```
constant bits(64)   CAP_NO_SEALING = Ones(64);
```

5.320 shared/functions/capability/CAP_OTYPE_HI_BIT

```
constant integer CAP_OTYPE_HI_BIT  = 109;
```

5.321 shared/functions/capability/CAP_OTYPE_LO_BIT

```
constant integer CAP_OTYPE_LO_BIT  = 95;
```

5.322 shared/functions/capability/CAP_OTYPE_NUM_BITS

```
constant integer CAP_OTYPE_NUM_BITS = CAP_OTYPE_HI_BIT-CAP_OTYPE_LO_BIT+1;
```

5.323 shared/functions/capability/CAP_PERMS_HI_BIT

```
constant integer CAP_PERMS_HI_BIT  = 127;
```

5.324 shared/functions/capability/CAP_PERMS_LO_BIT

```
constant integer CAP_PERMS_LO_BIT  = 110;
```

5.325 shared/functions/capability/CAP_PERMS_NUM_BITS

```
constant integer CAP_PERMS_NUM_BITS = CAP_PERMS_HI_BIT-CAP_PERMS_LO_BIT+1;
```

5.326 shared/functions/capability/CAP_PERM_BRANCH_SEALED_PAIR

```
constant bits(64) CAP_PERM_BRANCH_SEALED_PAIR = (1<<8)<63:0>;
```

5.327 shared/functions/capability/CAP_PERM_COMPARTMENT_ID

```
constant bits(64) CAP_PERM_COMPARTMENT_ID = (1<<7)<63:0>;
```

5.328 shared/functions/capability/CAP_PERM_EXECUTE

```
constant bits(64) CAP_PERM_EXECUTE       = (1<<15)<63:0>;
```

5.329 shared/functions/capability/CAP_PERM_EXECUTIVE

```
constant bits(64) CAP_PERM_EXECUTIVE     = (1<<1)<63:0>;
```

5.330 shared/functions/capability/CAP_PERM_GLOBAL

```
constant bits(64) CAP_PERM_GLOBAL        = 1<63:0>;
```

5.331 shared/functions/capability/CAP_PERM_LOAD

```
constant bits(64) CAP_PERM_LOAD          = (1<<17)<63:0>;
```

5.332 shared/functions/capability/CAP_PERM_LOAD_CAP

```
constant bits(64) CAP_PERM_LOAD_CAP      = (1<<14)<63:0>;
```

5.333 shared/functions/capability/CAP_PERM_MUTABLE_LOAD

```
constant bits(64) CAP_PERM_MUTABLE_LOAD  = (1<<6)<63:0>;
```

5.334 shared/functions/capability/CAP_PERM_NONE

```
constant bits(64) CAP_PERM_NONE          = 0<63:0>;
```

5.335 shared/functions/capability/CAP_PERM_SEAL

```
constant bits(64) CAP_PERM_SEAL          = (1<<11)<63:0>;
```

5.336 shared/functions/capability/CAP_PERM_STORE

```
constant bits(64) CAP_PERM_STORE         = (1<<16)<63:0>;
```

5.337 shared/functions/capability/CAP_PERM_STORE_CAP

```
constant bits(64) CAP_PERM_STORE_CAP     = (1<<13)<63:0>;
```

5.338 shared/functions/capability/CAP_PERM_STORE_LOCAL

```
constant bits(64) CAP_PERM_STORE_LOCAL   = (1<<12)<63:0>;
```

5.339 shared/functions/capability/CAP_PERM_SYSTEM

```
constant bits(64) CAP_PERM_SYSTEM        = (1<<9)<63:0>;
```

5.340 shared/functions/capability/CAP_PERM_UNSEAL

```
constant bits(64) CAP_PERM_UNSEAL        = (1<<10)<63:0>;
```

5.341 shared/functions/capability/CAP_SEAL_TYPE_LB

```
constant bits(64)   CAP_SEAL_TYPE_LB  = ZeroExtend('11',64);
```

5.342 shared/functions/capability/CAP_SEAL_TYPE_LPB

```
constant bits(64)   CAP_SEAL_TYPE_LPB = ZeroExtend('10',64);
```

5.343 shared/functions/capability/CAP_SEAL_TYPE_RB

```
constant bits(64)   CAP_SEAL_TYPE_RB  = ZeroExtend('01',64);
```

5.344 shared/functions/capability/CAP_TAG_BIT

```
constant integer CAP_TAG_BIT       = 128;
```

5.345 shared/functions/capability/CAP_VALUE_FOR_BOUND_HI_BIT

```
constant integer CAP_VALUE_FOR_BOUND_HI_BIT = 55;
```

5.346 shared/functions/capability/CAP_VALUE_FOR_BOUND_NUM_BITS

```
constant integer CAP_VALUE_FOR_BOUND_NUM_BITS = CAP_VALUE_FOR_BOUND_HI_BIT-CAP_VALUE_LO_BIT+1;
```

5.347 shared/functions/capability/CAP_VALUE_HI_BIT

```
constant integer CAP_VALUE_HI_BIT           = 63;
```

5.348 shared/functions/capability/CAP_VALUE_LO_BIT

```
constant integer CAP_VALUE_LO_BIT           = 0;
```

5.349 shared/functions/capability/CAP_VALUE_NUM_BITS

```
constant integer CAP_VALUE_NUM_BITS = CAP_VALUE_HI_BIT-CAP_VALUE_LO_BIT+1;
```

5.350 shared/functions/capability/CapAdd

```
// CapAdd()
// ========
// Returns the input capability with the value adjusted by a given delta, if
// this results in the bounds no longer being representable the tag is cleared

Capability CapAdd(Capability c, bits(CAP_VALUE_NUM_BITS) increment)
    Capability newc = c;
    newc<CAP_VALUE_HI_BIT:CAP_VALUE_LO_BIT> = CapGetValue(c) + increment;
    if !CapIsRepresentableFast(c, increment) then
        newc<CAP_TAG_BIT> = '0';

    if CapIsExponentOutOfRange(c) then
        newc<CAP_TAG_BIT> = '0';

    // if any bounds bits are taken from the value, ensure the top address bit doesn't change
    if (CapBoundsUsesValue(CapGetExponent(c)) &&
        CapGetValue(c)<CAP_FLAGS_LO_BIT-1> != CapGetValue(newc)<CAP_FLAGS_LO_BIT-1>) then
        newc<CAP_TAG_BIT> = '0';

    return newc;

// CapAdd()
// ========
// Integer version of CapAdd to simplify pseudocode for computing the link
// register

Capability CapAdd(Capability c, integer increment)
    return CapAdd(c,increment<CAP_VALUE_NUM_BITS-1:0>);
```

5.351 shared/functions/capability/CapBoundsAddress

```
// CapBoundsAddress()
// ==================
// Return a possibly modified address suitable for generating bounds

bits(CAP_VALUE_NUM_BITS) CapBoundsAddress(bits(CAP_VALUE_NUM_BITS) address)
    return SignExtend(address<CAP_FLAGS_LO_BIT-1:0>, CAP_VALUE_NUM_BITS);
```

5.352 shared/functions/capability/CapBoundsEqual

```
// CapBoundsEqual()
// ================
// Return if the bounds of two capbilities are equal

boolean CapBoundsEqual(Capability a, Capability b)
    (abase,alimit,avalid) = CapGetBounds(a);
    (bbase,blimit,bvalid) = CapGetBounds(b);
    // The bounds are never equal if there is an out of range exponent involved.
    return (abase == bbase) && (alimit == blimit) && avalid && bvalid;
```

5.353 shared/functions/capability/CapBoundsUsesValue

```
// CapBoundsUsesValue()
// ====================
// Return whether the capability bounds use value bits in the calculation

boolean CapBoundsUsesValue(integer exp)
    return exp + CAP_MW < CAP_VALUE_NUM_BITS;
```

5.354 shared/functions/capability/CapCheckPermissions

```
// CapCheckPermissions()
// =====================
// Returns true if a capability has all permissions in a given bit mask, false
// otherwise

boolean CapCheckPermissions(Capability c, bits(64) mask)
    bits(CAP_PERMS_NUM_BITS) perms = CapGetPermissions(c);
    return (perms OR NOT mask<CAP_PERMS_NUM_BITS-1:0>) == Ones(CAP_PERMS_NUM_BITS);
```

5.355 shared/functions/capability/CapClearPerms

```
// CapClearPerms()
// ===============
// Returns the input capability with permissions cleared
// according to a given bit mask

Capability CapClearPerms(Capability c, bits(64) mask)
    bits(CAP_PERMS_NUM_BITS) old_perms = CapGetPermissions(c);
    bits(CAP_PERMS_NUM_BITS) new_perms = old_perms AND NOT mask<CAP_PERMS_NUM_BITS-1:0>;
    c<CAP_PERMS_HI_BIT:CAP_PERMS_LO_BIT> = new_perms<CAP_PERMS_NUM_BITS-1:0>;
    return c;
```

5.356 shared/functions/capability/CapGetBase

```
// CapGetBase()
// ============
// Get the capability base in a form of the right type to use in arithmetic
// involving the Capability Value.

bits(CAP_VALUE_NUM_BITS) CapGetBase(Capability c)
    (base, - , - ) = CapGetBounds(c);

    return base<0+:CAP_VALUE_NUM_BITS>;
```

5.357 shared/functions/capability/CapGetBottom

```
// CapGetBottom()
// ==============
// Returns the bottom value

bits(CAP_MW) CapGetBottom(Capability c)
    if CapIsInternalExponent(c) then
        return c<CAP_BASE_HI_BIT:CAP_BASE_MANTISSA_LO_BIT>:'000';
    else
        return c<CAP_BASE_HI_BIT:CAP_BASE_LO_BIT>;
```

5.358 shared/functions/capability/CapGetBounds

```
// CapGetBounds()
// ==============
// Returns the bounds tuple. The tuple is composed of
// (base,limit,isExponentValid). As the top bound depends on the calculation of
// the bottom bound it better to always calculate them together The base can
// never have the CAP_BOUND_NUM_BITSth bit set.  However in order to do
// arithmetic combining them base and limit must be of the same type.

(bits(CAP_BOUND_NUM_BITS), bits(CAP_BOUND_NUM_BITS), boolean) CapGetBounds(Capability c)
    integer exp = CapGetExponent(c);

    if exp == CAP_MAX_ENCODEABLE_EXPONENT then
        return (CAP_BOUND_MIN,CAP_BOUND_MAX,TRUE);

    if CapIsExponentOutOfRange(c) then
        return (CAP_BOUND_MIN,CAP_BOUND_MAX,FALSE);

    bits(66) base;
    bits(66) limit;
    bits(CAP_MW) bottom = CapGetBottom(c);
    bits(CAP_MW) top    = CapGetTop(c);
    // alow is filled with zeros
    base<0+:exp> = Zeros(exp);
    limit<0+:exp> = Zeros(exp);
    // amid is the recovered value of T or B. As exp cannot be greater than 50
    // we cannot do an out of range bitslice with MW = 16 and 66 bit
    // arithmetic.
    base<exp+CAP_MW-1:exp> = bottom;
    limit<exp+CAP_MW-1:exp> = top;

    // Calculate inputs to correction calculations
    bits(66) a =  '00':CapBoundsAddress(CapGetValue(c));
    bits(3)  A3 = a<exp+CAP_MW-1:exp+CAP_MW-3>;
    bits(3)  B3 = bottom<CAP_MW-1:CAP_MW-3>;
    bits(3)  T3 = top<CAP_MW-1:CAP_MW-3>;
    bits(3)  R3 = B3 - '001';

    integer aHi;
    if CapUnsignedLessThan(A3,R3) then
        aHi = 1;
    else
        aHi = 0;

    integer bHi;
    if CapUnsignedLessThan(B3,R3) then
        bHi = 1;
    else
        bHi = 0;

    integer tHi;
    if CapUnsignedLessThan(T3,R3) then
        tHi = 1;
    else
        tHi = 0;

    correction_base = bHi - aHi;
    correction_limit  = tHi - aHi;

    // Determine if we need any atop bits or if they have all been shifted off
    // the top of the calculation.
    if exp+CAP_MW < CAP_MAX_EXPONENT+CAP_MW then
        atop = a<65:exp+CAP_MW>;
        base<65:exp+CAP_MW> = atop + correction_base;
        limit<65:exp+CAP_MW> = atop + correction_limit;

    // Final correction for limit for capabilities which wrap the address space
    bits(2) l2 = limit<64:63>;
    bits(2) b2 = '0':base<63>;
    if exp < (CAP_MAX_EXPONENT-1) && CapUnsignedGreaterThan(l2 - b2,'01') then
        limit<64> = NOT(limit<64>);

    return ('0':base<63:0>, limit<64:0>, TRUE);
```

5.359 shared/functions/capability/CapGetExponent

```
// CapGetExponent()
// ================
// Returns the exponent in the range 0 to 63
// The Te and Be bits are stored inverted

integer CapGetExponent(Capability c)
    if CapIsInternalExponent(c) then
        bits(6) nexp = c<CAP_LIMIT_EXP_HI_BIT:CAP_LIMIT_LO_BIT>:c<CAP_BASE_EXP_HI_BIT:CAP_BASE_LO_BIT>;
        return UInt(NOT(nexp));
    else
        return 0;

// CapIsExponentOutOfRange()
// Returns true if the exponent is not in the legal range, false otherwise.

boolean CapIsExponentOutOfRange(Capability c)
    integer exp = CapGetExponent(c);
    // To ensure 0 is a legal capability CAP_MAX_ENCODEABLE_EXPONENT is valid
    // and is handled specially.
    return (exp > CAP_MAX_EXPONENT) && (exp < CAP_MAX_ENCODEABLE_EXPONENT);
```

5.360 shared/functions/capability/CapGetLength

```
// CapGetLength()
// ==============
// Returns the length of the capability

bits(CAP_LENGTH_NUM_BITS) CapGetLength(Capability c)
    (base, limit, - ) = CapGetBounds(c);
    return limit - base;
```

5.361 shared/functions/capability/CapGetObjectType

```
// CapGetObjectType()
// ==================
// Returns the object type

bits(CAP_VALUE_NUM_BITS) CapGetObjectType(Capability c)
    return ZeroExtend(c<CAP_OTYPE_HI_BIT:CAP_OTYPE_LO_BIT>,CAP_VALUE_NUM_BITS);
```

5.362 shared/functions/capability/CapGetOffset

```
// CapGetOffset()
// ==============
// Returns the offset of the capability value
// relative to the capability base address

bits(CAP_VALUE_NUM_BITS) CapGetOffset(Capability c)
    (base, - , - ) = CapGetBounds(c);
    offset = '0':CapGetValue(c) - base;
    return offset<0+:CAP_VALUE_NUM_BITS>;
```

5.363 shared/functions/capability/CapGetPermissions

```
// CapGetPermissions()
// ===================
// Returns a bit vector of capability permissions

bits(CAP_PERMS_NUM_BITS) CapGetPermissions(Capability c)
    return c<CAP_PERMS_HI_BIT:CAP_PERMS_LO_BIT>;
```

5.364 shared/functions/capability/CapGetRepresentableMask

```
// CapGetRepresentableMask()
// =========================
// Return a mask that can be used to align down addresses to a value that is
// sufficient to set precise bounds for the given nearest representable length

bits(CAP_VALUE_NUM_BITS) CapGetRepresentableMask(bits(CAP_VALUE_NUM_BITS) len)
    // CapNull if interpreted as a capability has maximum bounds and it is
    // defined that introspection does not depend on the tag. Therefore it can
    // be used here.
    Capability c = CapNull();
    bits(CAP_VALUE_NUM_BITS)  test_base = Ones(CAP_VALUE_NUM_BITS) - len;
    bits(CAP_LENGTH_NUM_BITS) test_length = ZeroExtend(len,CAP_LENGTH_NUM_BITS);
    c<CAP_VALUE_HI_BIT:CAP_VALUE_LO_BIT> = test_base;
    c = CapSetBounds(c,test_length,FALSE);

    // CapSetBounds provably cannot create an exponent greater than
    // CAP_MAX_EXPONENT therefore a bad exponent check does not need to be done
    // in this case.
    integer exp1 = 0;
    if CapIsInternalExponent(c) then
        exp1 = CapGetExponent(c) + 3;

    return Ones(CAP_VALUE_NUM_BITS-exp1):Zeros(exp1);
```

5.365 shared/functions/capability/CapGetTag

```
// CapGetTag()
// ===========
// Returns the tag bit in bit<0> of the return value

bits(64) CapGetTag(Capability c)
    return ZeroExtend(c<CAP_TAG_BIT>,64);
```

5.366 shared/functions/capability/CapGetTop

```
// CapGetTop()
// ===========
// Returns the top value

bits(CAP_MW) CapGetTop(Capability c)
    bits(2) lmsb = '00';
    bits(2) lcarry = '00';
    bits(CAP_MW) b = CapGetBottom(c);
    bits(CAP_MW) t;
    if CapIsInternalExponent(c) then
        lmsb = '01';
        t = '00':c<CAP_LIMIT_HI_BIT:CAP_LIMIT_MANTISSA_LO_BIT>:'000';
    else
        t = '00':c<CAP_LIMIT_HI_BIT:CAP_LIMIT_LO_BIT>;
    if CapUnsignedLessThan(t<CAP_MW-3:0>,b<CAP_MW-3:0>) then
        lcarry = '01';
    t<CAP_MW-1:CAP_MW-2> = b<CAP_MW-1:CAP_MW-2> + lmsb + lcarry;
    return t;
```

5.367 shared/functions/capability/CapGetValue

```
// CapGetValue()
// =============
// Returns value field of a capability

bits(CAP_VALUE_NUM_BITS) CapGetValue(Capability c)
    return c<CAP_VALUE_HI_BIT:CAP_VALUE_LO_BIT>;
```

5.368 shared/functions/capability/CapIsBaseAboveLimit

```
// CapIsBaseAboveLimit()
// =====================
// Returns true if the base is strictly greater than the limit, false otherwise

boolean CapIsBaseAboveLimit(Capability c)
    (base,limit,-) = CapGetBounds(c);
    return CapUnsignedGreaterThan(base,limit);
```

5.369 shared/functions/capability/CapIsEqual

```
// CapIsEqual()
// ============
// Returns true if two capabilities are bitwise identical, false otherwise.

boolean CapIsEqual(Capability c1, Capability c2)
    return c1 == c2;
```

5.370 shared/functions/capability/CapIsExecutePermitted

```
// CapIsExecutePermitted()
// =======================
// Returns true if the capability permits code execution, false otherwise

boolean CapIsExecutePermitted(Capability c)
    return CapCheckPermissions(c, CAP_PERM_EXECUTE);
```

5.371 shared/functions/capability/CapIsExecutive

```
// CapIsExecutive()
// ================
// Returns true if the capability has Executive permission, false otherwise
```

```

boolean CapIsExecutive(Capability c)
    return CapCheckPermissions(c, CAP_PERM_EXECUTIVE);
```

5.372 shared/functions/capability/CapIsInBounds

```
// CapIsInBounds()
// ===============
// Returns true if the capability value is within the capability bounds, false
// otherwise.

boolean CapIsInBounds(Capability c)
    (base, limit, valid) = CapGetBounds(c);
    value65 = '0':CapGetValue(c);
    // Never in bounds if there is an out of range exponent involved
    return CapUnsignedGreaterThanOrEqual(value65,base) && CapUnsignedLessThan(value65,limit) && valid;
```

5.373 shared/functions/capability/CapIsInternalExponent

```
// CapIsInternalExponent()
// =======================
// Returns true if an internal exponent is in use, false otherwise.
// The Ie bit is stored inverted.

boolean CapIsInternalExponent(Capability c)
    return c<CAP_IE_BIT> == '0';
```

5.374 shared/functions/capability/CapIsLocal

```
// CapIsLocal()
// ============
// Returns true if the capability is local, false otherwise

boolean CapIsLocal(Capability c)
    return !CapCheckPermissions(c, CAP_PERM_GLOBAL);
```

5.375 shared/functions/capability/CapIsMutableLoadPermitted

```
// CapIsMutableLoadPermitted()
// ===========================
// Returns true if the capability is capable of loading capabilities
// for use in store operations, false otherwise

boolean CapIsMutableLoadPermitted(Capability c)
    return CapCheckPermissions(c, CAP_PERM_MUTABLE_LOAD);
```

5.376 shared/functions/capability/CapIsRangeInBounds

```
// CapIsRangeInBounds()
// ====================
// Returns true if a range of values is within capability bounds, false otherwise

boolean CapIsRangeInBounds(Capability c, bits(CAP_VALUE_NUM_BITS) start_address, bits(CAP_VALUE_NUM_BITS+1) length)
    (base, limit, valid) = CapGetBounds(c);
    start_ext = '0':start_address;
    limit_ext = start_ext + length;
    // Never in bounds if there is an out of range exponent involved
    return CapUnsignedGreaterThanOrEqual(start_ext,base) && CapUnsignedLessThanOrEqual(limit_ext,limit) && valid;
```

5.377 shared/functions/capability/CapIsRepresentable

```
// CapIsRepresentable()
// ====================
// Return if the bounds are still representable if a new value is applied to an
// an existing capability.

boolean CapIsRepresentable(Capability c, bits(CAP_VALUE_NUM_BITS) address)
    Capability newc = c;
    newc<CAP_VALUE_HI_BIT:CAP_VALUE_LO_BIT> = address;
    return CapBoundsEqual(c,newc);
```

5.378 shared/functions/capability/CapIsRepresentableFast

```
// CapIsRepresentableFast()
// ========================
// Return if the bounds are still representable if a new value is applied to an
// an existing capability. This version is used for CapAdd only and may exhibit
// false negatives vs the full CapIsRepresentable check for values which which
// are outside bounds.

boolean CapIsRepresentableFast(Capability c, bits(CAP_VALUE_NUM_BITS) increment)
    integer exp = CapGetExponent(c);
    if exp >= (CAP_MAX_EXPONENT - 2) then
        return TRUE;
    else
        bits(CAP_VALUE_NUM_BITS) a = CapGetValue(c);
        // calculation needs to be done on address rather than the value
        a = CapBoundsAddress(a);
        increment = CapBoundsAddress(increment);

        i_top = ASR(increment,exp+CAP_MW);
        i_mid = LSR(increment,exp)<CAP_MW-1:0>;
        a_mid = LSR(a,exp)<CAP_MW-1:0>;
        B3   = CapGetBottom(c)<CAP_MW-1:CAP_MW-3>;
        R3   = B3 - '001';
        R    = R3:Zeros(CAP_MW-3);
        diff = R - a_mid;
        diff1 = diff - 1;

        // Comparing against Ones below is used as proxy for comparing against
        // -1 to avoid any issues with comparing a bits value against a signed
        // integer.
        if (i_top == 0) then
            return CapUnsignedLessThan(i_mid, diff1);
        elsif (i_top == Ones(CAP_VALUE_NUM_BITS)) then
            return CapUnsignedGreaterThanOrEqual(i_mid, diff) && (R != a_mid);
        else
            return FALSE;
```

5.379 shared/functions/capability/CapIsSealed

```
// CapIsSealed()
// =============
// Returns true if the input capability is sealed

boolean CapIsSealed(Capability c)
    return CapGetObjectType(c) != Zeros(CAP_VALUE_NUM_BITS);
```

5.380 shared/functions/capability/CapIsSubSetOf

```
// CapIsSubSetOf()
// ===============
// Returns true if capability a is a subset or equal to capability b

boolean CapIsSubSetOf(Capability a, Capability b)
    (abase,alimit,avalid) = CapGetBounds(a);
    (bbase,blimit,bvalid) = CapGetBounds(b);
    boolean boundsSubset = CapUnsignedGreaterThanOrEqual(abase,bbase) &&
        ↪CapUnsignedLessThanOrEqual(alimit,blimit);
    boolean permsSubset = (CapGetPermissions(a) AND NOT(CapGetPermissions(b))) ==
        ↪Zeros(CAP_PERMS_NUM_BITS);
    // Subset is never true if there is an out of range exponent involved
    return boundsSubset && permsSubset && avalid && bvalid;
```

5.381 shared/functions/capability/CapIsSystemAccessPermitted

```
// CapIsSystemAccessPermitted()
// ============================
// Returns true if the capability permits system register accesses, false otherwise.

boolean CapIsSystemAccessPermitted(Capability c)
    return CapCheckPermissions(c, CAP_PERM_EXECUTE OR CAP_PERM_SYSTEM);
```

5.382 shared/functions/capability/CapIsTagClear

```
// CapIsTagClear()
// ===============
// Return true if the tag is clear, false otherwise

boolean CapIsTagClear(Capability c)
    return CapGetTag(c)<0> == '0';
```

5.383 shared/functions/capability/CapIsTagSet

```
// CapIsTagSet()
// =============
// Return true if the tag is set, false otherwise

boolean CapIsTagSet(Capability c)
    return CapGetTag(c)<0> == '1';
```

5.384 shared/functions/capability/CapNull

```
// CapNull()
// =========
// Returns the null capability defined as all zeros

Capability CapNull()
    Capability c = Zeros(129);
    return c;
```

5.385 shared/functions/capability/CapPermsInclude

```
// CapPermsInclude()
// =================
// Returns true if the perms includes the permissions in mask, false otherwise

boolean CapPermsInclude(bits(64) perms, bits(64) mask)
    return (perms<CAP_PERMS_NUM_BITS-1:0> AND mask<CAP_PERMS_NUM_BITS-1:0>) ==
        ↪mask<CAP_PERMS_NUM_BITS-1:0>;
```

5.386 shared/functions/capability/CapSetBounds

```
// CapSetBounds
// ============
// Returns a capability, derived from the input capability, with base address
// set to the value of the input capability and the length set to a given
// value.  If precise bounds setting is not possible, either the bounds are
// rounded, or tag is cleared, depending on the input exact flag.

Capability CapSetBounds(Capability c, bits(CAP_LENGTH_NUM_BITS) req_len, boolean exact)
    // For this ASL to be valid according to the proved properties req_len must
    // be at most 2^64. Called from the ISA via a register it can never be more than 2^64-1.
    assert CapUnsignedLessThanOrEqual(req_len,CAP_BOUND_MAX);

    // Find a candidate exponent
    integer exp = CAP_MAX_EXPONENT - CountLeadingZeroBits(req_len<CAP_VALUE_NUM_BITS:CAP_MW-1>);
    // If the candidate exponent is non zero or the calculated part of 'T' for
    // bounds decoding is not zero then the internal exponent is used.
    boolean ie = (exp != 0) || req_len<CAP_MW-2> == '1';

    bits(CAP_VALUE_NUM_BITS) base = CapGetValue(c);
    // Choose the actual base based on whether the desired capability is 'Large' or 'Small'
    // As exp can be increased in some cases, some potentially large capabilties
    // will be classed as small.
    bits(CAP_VALUE_NUM_BITS) abase = if CapBoundsUsesValue(CapGetExponent(c)) then CapBoundsAddress(base)
         ↪else base;

    bits(CAP_VALUE_NUM_BITS+2) req_base = '00':abase;
    bits(CAP_VALUE_NUM_BITS+2) req_top  = req_base + ('0':req_len);

    // Caclulate for the non ie case
    bits(CAP_MW) Bbits = req_base<CAP_MW-1:0>;
    bits(CAP_MW) TBits = req_top<CAP_MW-1:0>;
    boolean lostTop = FALSE;
    boolean lostBottom = FALSE;
    boolean incrementE = FALSE;

    if ie then
        // Logically the upper bit address is exp+3+CAP_MW-3-1 but +3-3 can
        // trivially be omitted.
        bits(CAP_MW-3) B_ie = req_base<exp+CAP_MW-1:exp+3>;
        bits(CAP_MW-3) T_ie = req_top<exp+CAP_MW-1:exp+3>;

        // Have we lost any bits of base or top?
        bits(CAP_VALUE_NUM_BITS+2) maskLo = ZeroExtend(Ones(exp+3),CAP_VALUE_NUM_BITS+2);
        lostBottom = (req_base AND maskLo) != Zeros(CAP_VALUE_NUM_BITS+2);
        lostTop    = (req_top  AND maskLo) != Zeros(CAP_VALUE_NUM_BITS+2);

        if lostTop then
            // Increment T to make sure it is still above top even with lost bits.
            // It might wrap but if that makes B<T then decoding will compensate.
            T_ie = T_ie + 1;

        // We chose e so that the top two bits of the length should be 0b01
        // however we may have overflowed if T was incremented or we lost bits
        // of base.
        L_ie = T_ie - B_ie;
        if L_ie<CAP_MW-4> == '1' then
            incrementE = TRUE;

            lostBottom = lostBottom || B_ie[0] == '1';
            lostTop    = lostTop || T_ie[0] == '1';

            // Recalculate. This cannot produce an out of range slice as an SMT
            // proof exists that the algorithm can never produce an exponent
            // greater than CAP_MAX_EXPONENT and we are just about to increment
            // so exp can only be CAP_MAX_EXPONENT-1.
            assert exp < CAP_MAX_EXPONENT;
            B_ie = req_base<exp+CAP_MW:exp+4>;
            T_ie = req_top<exp+CAP_MW:exp+4>;
            if lostTop then
                T_ie = T_ie + 1;

        if incrementE == TRUE then
            exp = exp + 1;

        Bbits = B_ie:'000';
        TBits = T_ie:'000';

    // Now construct the return
    Capability newc = c;

    // We must check request was within the bounds of the original capability
    // and unset the tag if it was not. This must be done using the sign
    // extended address not including the flags field.
    (obase, olimit, ovalid) = CapGetBounds(c);
    if (!CapUnsignedGreaterThanOrEqual(req_base<0+:CAP_BOUND_NUM_BITS>,obase) ||
       !CapUnsignedLessThanOrEqual(req_top<0+:CAP_BOUND_NUM_BITS>,olimit) ||
       !ovalid) then
        newc<CAP_TAG_BIT> = '0';

    // The ie bit and the Te and Be bits are stored inverted
    if ie then
        newc<CAP_IE_BIT> = '0';
        newc<CAP_BASE_EXP_HI_BIT:CAP_BASE_LO_BIT>   = NOT(exp<2:0>);
        newc<CAP_LIMIT_EXP_HI_BIT:CAP_LIMIT_LO_BIT> = NOT(exp<5:3>);
    else
        newc<CAP_IE_BIT> = '1';
        newc<CAP_BASE_EXP_HI_BIT:CAP_BASE_LO_BIT>   = Bbits<2:0>;
        newc<CAP_LIMIT_EXP_HI_BIT:CAP_LIMIT_LO_BIT> = TBits<2:0>;

    newc<CAP_BASE_HI_BIT:CAP_BASE_MANTISSA_LO_BIT>   = Bbits<CAP_MW-1:3>;
    // The top two bits of T are recovered during decoding
    newc<CAP_LIMIT_HI_BIT:CAP_LIMIT_MANTISSA_LO_BIT> = TBits<CAP_MW-3:3>;

    // if reducing bounds from a large to a small capability, the original
    // base needs to have consistent bits at the top
    boolean from_large = !CapBoundsUsesValue(CapGetExponent(c));
    boolean to_small = CapBoundsUsesValue(exp);
    if from_large && to_small && SignExtend(base<CAP_FLAGS_LO_BIT-1:0>, 64) != base then
        newc<CAP_TAG_BIT> = '0';

    // If we were asked for an exact bound and could not provide it then we must clear the tag
    if exact && (lostBottom || lostTop) then
        newc<CAP_TAG_BIT> = '0';

    return newc;
```

5.387 shared/functions/capability/CapSetObjectType

```
// CapSetObjectType()
// ==================
// Returns the capability c with the object type set to o

Capability CapSetObjectType(Capability c, bits(64) o)
    c<CAP_OTYPE_HI_BIT:CAP_OTYPE_LO_BIT> = o<CAP_OTYPE_NUM_BITS-1:0>;
    return c;

// CapGetFlags()
// Returns the flags field

bits(CAP_VALUE_NUM_BITS) CapGetFlags(Capability c)
    bits(CAP_VALUE_NUM_BITS) r = c<CAP_FLAGS_HI_BIT:CAP_FLAGS_LO_BIT>:Zeros(CAP_VALUE_FOR_BOUND_NUM_BITS);
    return r;

// CapSetFlags()
// Sets the flags field from flags field of f

Capability CapSetFlags(Capability c, bits(CAP_VALUE_NUM_BITS) f)
    c<CAP_FLAGS_HI_BIT:CAP_FLAGS_LO_BIT> = f<CAP_FLAGS_HI_BIT:CAP_FLAGS_LO_BIT>;
    return c;
```

5.388 shared/functions/capability/CapSetOffset

```
// CapSetOffset()
// ==============
// Returns the input capability with the address offset set to a given value.
// If this results in the bounds not being representable then the tag is
// cleared

Capability CapSetOffset(Capability c, bits(CAP_VALUE_NUM_BITS) offset)
    // If the exponent is valid does not need to be checked here as CapAdd will
    // unset the tag if it is.
    (base, - , - ) = CapGetBounds(c);
    bits(CAP_VALUE_NUM_BITS) newvalue = base<CAP_VALUE_NUM_BITS-1:0> + offset;
    bits(CAP_VALUE_NUM_BITS) increment = newvalue - CapGetValue(c);
    return CapAdd(c, increment);
```

5.389 shared/functions/capability/CapSetTag

```
// CapSetTag()
// ===========
// Returns a capability formed by setting the tag bit of the argument c to
// bit<0> of the argument t

Capability CapSetTag(Capability c, bits(64) t)
    Capability r = c;
    r<CAP_TAG_BIT> = t<0>;
    return r;
```

5.390 shared/functions/capability/CapSetValue

```
// CapSetValue()
// =============
// Returns the input capability with the value set to v, if this results in the
// capability bounds not being respresentable the tag is cleared

Capability CapSetValue(Capability c, bits(CAP_VALUE_NUM_BITS) v)
    bits(CAP_VALUE_NUM_BITS) oldv = CapGetValue(c);
    if !CapIsRepresentable(c,v) then
        c = CapWithTagClear(c);
    c<CAP_VALUE_HI_BIT:CAP_VALUE_LO_BIT> = v;

    // if any bounds bits are taken from the value, ensure the top address bit doesn't change
    if (CapBoundsUsesValue(CapGetExponent(c)) &&
        v<CAP_FLAGS_LO_BIT-1> != oldv<CAP_FLAGS_LO_BIT-1>) then
        c = CapWithTagClear(c);

    return c;
```

5.391 shared/functions/capability/CapSquashPostLoadCap

```
// CapSquashPostLoadCap()
// ======================
// Perform the following processing
//  - If the Capability was loaded without LoadCap permission clear the tag
//  - Remove MutableLoad, Store, StoreCap and StoreLocalCap permissions
//    in a loaded capability if accessed without MutableLoad permission

Capability CapSquashPostLoadCap(Capability data, VirtualAddress addr)

    Capability base_cap;

    if VAIsBits64(addr) then
        base_cap = DDC[];
    else
        base_cap = VAToCapability(addr);

    if !CapCheckPermissions(base_cap, CAP_PERM_LOAD_CAP) then
        data = CapWithTagClear(data);

    if !CapIsMutableLoadPermitted(base_cap) && CapIsTagSet(data) && !CapIsSealed(data) then
        data = CapClearPerms(data, CAP_PERM_STORE OR CAP_PERM_STORE_CAP OR CAP_PERM_STORE_LOCAL OR
            ↪CAP_PERM_MUTABLE_LOAD);

    return data;
```

5.392 shared/functions/capability/CapUnseal

```
// CapUnseal()
// ===========
// Returns an unsealed version of the input capability

Capability CapUnseal(Capability c)
    return CapSetObjectType(c,Zeros(64));
```

5.393 shared/functions/capability/CapUnsignedGreaterThan

```
// CapUnsignedGreaterThan()
// ========================
// Returns true if a is greater than b under an unsigned greater than operation.

boolean CapUnsignedGreaterThan(bits(N) a, bits(N) b)
    return UInt(a) > UInt(b);
```

5.394 shared/functions/capability/CapUnsignedGreaterThanOrEqual

```
// CapUnsignedGreaterThanOrEqual()
// ===============================
// Returns true if a is greater than b under an unsigned greater than or equal operation.

boolean CapUnsignedGreaterThanOrEqual(bits(N) a, bits(N) b)
    return UInt(a) >= UInt(b);
```

5.395 shared/functions/capability/CapUnsignedLessThan

```
// CapUnsignedLessThan()
// =====================
// Returns true if a is less than b under an unsigned less than operation.

boolean CapUnsignedLessThan(bits(N) a, bits(N) b)
    return UInt(a) < UInt(b);
```

5.396 shared/functions/capability/CapUnsignedLessThanOrEqual

```
// CapUnsignedLessThanOrEqual()
// ============================
// Returns true if a is less than b under an unsigned less than or equal operation.

boolean CapUnsignedLessThanOrEqual(bits(N) a, bits(N) b)
    return UInt(a) <= UInt(b);
```

5.397 shared/functions/capability/CapWithTagClear

```
// CapWithTagClear()
// =================
// Returns the input capability with tag cleared

Capability CapWithTagClear(Capability c)
    return CapSetTag(c,ZeroExtend('0',64));
```

5.398 shared/functions/capability/CapWithTagSet

```
// CapWithTagSet()
// ===============
// Returns the input capability with tag set

Capability CapWithTagSet(Capability c)
    return CapSetTag(c,ZeroExtend('1',64));
```

5.399 shared/functions/capability/CapabilityFromData

```
// CapabilityFromData()
// ====================
// Converts a 1-bit tag and 128-bit data to a Capability

Capability CapabilityFromData(integer size, bits(1) tag, bits(size) data)
    Capability c;
    c<size-1:0> = data;
    c<CAP_TAG_BIT> = tag;
    return c;
```

5.400 shared/functions/capability/DataFromCapability

```
// DataFromCapability()
// ====================
// Converts a Capability to a 1-bit tag and data of a given size

(bits(1), bits(size)) DataFromCapability(integer size, Capability c)
    return (c<CAP_TAG_BIT>, c<size-1:0>);
```

5.401 shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR_C(x, shift);
    return result;
```

5.402 shared/functions/common/ASR_C

```
// ASR_C()
// =======

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

5.403 shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

5.404 shared/functions/common/Align

```
// Align()
// =======

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =======

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

5.405 shared/functions/common/BitCount

```
// BitCount()
// ==========

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

5.406 shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// ======================

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

5.407 shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// ======================

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

5.408 shared/functions/common/Elem

```
// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<e*size+size-1 : e*size>;

// Elem[] - non-assignment form
// ============================

bits(size) Elem[bits(N) vector, integer e]
    return Elem[vector, e, size];

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;

// Elem[] - assignment form
// ========================

Elem[bits(N) &vector, integer e] = bits(size) value
    Elem[vector, e, size] = value;
    return;
```

5.409 shared/functions/common/Extend

```
// Extend()
// ========

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);

// Extend()
// ========

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

5.410 shared/functions/common/HighestSetBit

```
// HighestSetBit()
// ===============

integer HighestSetBit(bits(N) x)
```

```
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

5.411 shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

5.412 shared/functions/common/IsOnes

```
// IsOnes()
// ========

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

5.413 shared/functions/common/IsZero

```
// IsZero()
// ========

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

5.414 shared/functions/common/IsZeroBit

```
// IsZeroBit()
// ===========

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

5.415 shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL_C(x, shift);
    return result;
```

5.416 shared/functions/common/LSL_C

```
// LSL_C()
// =======

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

5.417 shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR_C(x, shift);
    return result;
```

5.418 shared/functions/common/LSR_C

```
// LSR_C()
// =======

(bits(N), bit) LSR_C(bits(N) x, integer shift)
    assert shift > 0;
    extended_x = ZeroExtend(x, shift+N);
    result = extended_x<shift+N-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

5.419 shared/functions/common/LowestSetBit

```
// LowestSetBit()
// ==============

integer LowestSetBit(bits(N) x)
    for i = 0 to N-1
        if x<i> == '1' then return i;
    return N;
```

5.420 shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
    return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
    return if a >= b then a else b;
```

5.421 shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
    return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
    return if a <= b then a else b;
```

5.422 shared/functions/common/Ones

```
// Ones()
// ======

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// ======

bits(N) Ones()
    return Ones(N);
```

5.423 shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR_C(x, shift);
    return result;
```

5.424 shared/functions/common/ROR_C

```
// ROR_C()
// =======

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

5.425 shared/functions/common/Replicate

```
// Replicate()
// ===========

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

5.426 shared/functions/common/RoundDown

```
integer RoundDown(real x);
```

5.427 shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// ==================

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

5.428 shared/functions/common/RoundUp

```
integer RoundUp(real x);
```

5.429 shared/functions/common/SInt

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

5.430 shared/functions/common/SignExtend

```
// SignExtend()
// ============

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;

// SignExtend()
// ============

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

5.431 shared/functions/common/UInt

```
// UInt()
// ======

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

5.432 shared/functions/common/ZeroExtend

```
// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;

// ZeroExtend()
// ============

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

5.433 shared/functions/common/Zeros

```
// Zeros()
// =======

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =======

bits(N) Zeros()
    return Zeros(N);
```

5.434 shared/functions/crc/BitReverse

```
// BitReverse()
// ============

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<N-i-1> = data<i>;
    return result;
```

5.435 shared/functions/crc/HaveCRCExt

```
// HaveCRCExt()
// ============

boolean HaveCRCExt()
    return HasArchVersion(ARMv8p1) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

5.436 shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// ============

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data, bits(32) poly)
    assert N > 32;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

5.437 shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// ==================
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
            ↪FFmul09(in3<c*8+:8>);
        out1<c*8+:8> =  FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
            ↪FFmul0D(in3<c*8+:8>);
        out2<c*8+:8> =  FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
            ↪FFmul0B(in3<c*8+:8>);
        out3<c*8+:8> =  FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
            ↪FFmul0E(in3<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
```

```
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

5.438 shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =================
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    return (
        op< 24+:8> : op< 48+:8> : op< 72+:8> : op< 96+:8> :
        op<120+:8> : op< 16+:8> : op< 40+:8> : op< 64+:8> :
        op< 88+:8> : op<112+:8> : op<  8+:8> : op< 32+:8> :
        op< 56+:8> : op< 80+:8> : op<104+:8> : op<  0+:8>
    );
```

5.439 shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// ================
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
    // Inverse S-box values
    bits(16*16*8) GF2_inv = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x7d0c2155631469e126d677ba7e042b17<127:0> :
        /*E*/ 0x619953833cbbebc8b0f52aae4d3be0a0<127:0> :
        /*D*/ 0xef9cc9939f7ae52d0d4ab519a97f5160<127:0> :
        /*C*/ 0x5fec8027591012b131c7078833a8dd1f<127:0> :
        /*B*/ 0xf45acd78fec0db9a2079d2c64b3e56fc<127:0> :
        /*A*/ 0x1bbe18aa0e62b76f89c5291d711af147<127:0> :
        /*9*/ 0x6edf751ce837f9e28535ade72274ac96<127:0> :
        /*8*/ 0x73e6b4f0cecff297eadc674f4111913a<127:0> :
        /*7*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<127:0> :
        /*6*/ 0x0645b3b80558e4f70ad3bc8c00abd890<127:0> :
        /*5*/ 0x849d8da75746155edab9edfd5048706c<127:0> :
        /*4*/ 0x92b6655dcc5ca4d41698688664f6f872<127:0> :
        /*3*/ 0x25d18b6d49a25b76b224d92866a12e08<127:0> :
        /*2*/ 0x4ec3fa420b954cee3d23c2a632947b54<127:0> :
        /*1*/ 0xcbe9dec444438e3487ff2f9b8239e37c<127:0> :
        /*0*/ 0xfbd7f3819ea340bf38a53630d56a0952<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2_inv<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

5.440 shared/functions/crypto/AESMixColumns

```
// AESMixColumns()
// ===============
// Transformation in the Cipher that takes all of the columns of the
// State and mixes their data (independently of one another) to
// produce new columns.

bits(128) AESMixColumns(bits (128) op)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> =  FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR         in2<c*8+:8>   EOR
            ↪in3<c*8+:8>;
        out1<c*8+:8> =          in0<c*8+:8>  EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
             ↪in3<c*8+:8>;
        out2<c*8+:8> =          in0<c*8+:8>  EOR         in1<c*8+:8>  EOR FFmul02(in2<c*8+:8>) EOR
             ↪FFmul03(in3<c*8+:8>);
        out3<c*8+:8> =  FFmul03(in0<c*8+:8>) EOR         in1<c*8+:8>  EOR         in2<c*8+:8>  EOR
             ↪FFmul02(in3<c*8+:8>);

    return (
        out3<3*8+:8> : out2<3*8+:8> : out1<3*8+:8> : out0<3*8+:8> :
        out3<2*8+:8> : out2<2*8+:8> : out1<2*8+:8> : out0<2*8+:8> :
        out3<1*8+:8> : out2<1*8+:8> : out1<1*8+:8> : out0<1*8+:8> :
        out3<0*8+:8> : out2<0*8+:8> : out1<0*8+:8> : out0<0*8+:8>
    );
```

5.441 shared/functions/crypto/AESShiftRows

```
// AESShiftRows()
// ==============
// Transformation in the Cipher that processes the State by cyclically
// shifting the last three rows of the State by different offsets.

bits(128) AESShiftRows(bits(128) op)
    return (
        op< 88+:8> : op< 48+:8> : op<  8+:8> : op< 96+:8> :
        op< 56+:8> : op< 16+:8> : op<104+:8> : op< 64+:8> :
        op< 24+:8> : op<112+:8> : op< 72+:8> : op< 32+:8> :
        op<120+:8> : op< 80+:8> : op< 40+:8> : op<  0+:8>
    );
```

5.442 shared/functions/crypto/AESSubBytes

```
// AESSubBytes()
// =============
// Transformation in the Cipher that processes the State using a nonlinear
// byte substitution table (S-box) that operates on each of the State bytes
// independently.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    bits(16*16*8) GF2 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x16bb54b00f2d99416842e6bf0d89a18c<127:0> :
        /*E*/ 0xdf2855cee9871e9b948ed9691198f8e1<127:0> :
        /*D*/ 0x9e1dc186b95735610ef6034866b53e70<127:0> :
        /*C*/ 0x8a8bbd4b1f74dde8c6b4a61c2e2578ba<127:0> :
        /*B*/ 0x08ae7a65eaf4566ca94ed58d6d37c8e7<127:0> :
        /*A*/ 0x79e4959162acd3c25c2406490a3a32e0<127:0> :
        /*9*/ 0xdb0b5ede14b8ee4688902a22dc4f8160<127:0> :
        /*8*/ 0x73195d643d7ea7c41744975fec130ccd<127:0> :
        /*7*/ 0xd2f3ff1021dab6bcf5389d928f40a351<127:0> :
        /*6*/ 0xa89f3c507f02f94585334d43fbaaefd0<127:0> :
        /*5*/ 0xcf584c4a39becb6a5bb1fc20ed00d153<127:0> :
        /*4*/ 0x842fe329b3d63b52a05a6e1b1a2c8309<127:0> :
        /*3*/ 0x75b227ebe28012079a059618c323c704<127:0> :
        /*2*/ 0x1531d871f1e5a534ccf73f362693fdb7<127:0> :
        /*1*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt(op<i*8+:8>)*8+:8>;
    return out;
```

5.443 shared/functions/crypto/FFmul02

```
// FFmul02()
// =========

bits(8) FFmul02(bits(8) b)
    bits(256*8) FFmul_02 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xE5E7E1E3EDEFE9EBF5F7F1F3FDFFF9FB<127:0> :
```

```
        /*E*/ 0xC5C7C1C3CDCFC9CBD5D7D1D3DDDFD9DB<127:0> :
        /*D*/ 0xA5A7A1A3ADAFA9ABB5B7B1B3BDBFB9BB<127:0> :
        /*C*/ 0x858781838D8F898B959791939D9F999B<127:0> :
        /*B*/ 0x656761636D6F696B757771737D7F797B<127:0> :
        /*A*/ 0x454741434D4F494B555751535D5F595B<127:0> :
        /*9*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECEAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt(b)*8+:8>;
```

5.444 shared/functions/crypto/FFmul03

```
// FFmul03()
// =========

bits(8) FFmul03(bits(8) b)
    bits(256*8) FFmul_03 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x1A191C1F16151013020104070E0D080B<127:0> :
        /*E*/ 0x2A292C2F262520233231343736E3D383B<127:0> :
        /*D*/ 0x7A797C7F767570736261646766E6D686B<127:0> :
        /*C*/ 0x4A494C4F464540435251545755E5D585B<127:0> :
        /*B*/ 0xDAD9DCDFD6D5D0D3C2C1C4C7CECDC8CB<127:0> :
        /*A*/ 0xEAE9ECEFE6E5E0E3F2F1F4F7FEFDF8FB<127:0> :
        /*9*/ 0xBAB9BCBFB6B5B0B3A2A1A4A7AEADA8AB<127:0> :
        /*8*/ 0x8A898C8F868580839291949797E9D989B<127:0> :
        /*7*/ 0x818287848D8E8B88999A9F9C95969390<127:0> :
        /*6*/ 0xB1B2B7B4BDBEBBB8A9AAAFACA5A6A3A0<127:0> :
        /*5*/ 0xE1E2E7E4EDEEEBE8F9FAFFFCF5F6F3F0<127:0> :
        /*4*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<127:0> :
        /*3*/ 0x414247444D4E4B48595A5F5C55565350<127:0> :
        /*2*/ 0x717277747D7E7B78696A6F6C65666360<127:0> :
        /*1*/ 0x212227242D2E2B28393A3F3C35363330<127:0> :
        /*0*/ 0x111217141D1E1B18090A0F0C05060300<127:0>
    );
    return FFmul_03<UInt(b)*8+:8>;
```

5.445 shared/functions/crypto/FFmul09

```
// FFmul09()
// =========

bits(8) FFmul09(bits(8) b)
    bits(256*8) FFmul_09 = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x464F545D626B70790E071C152A233831<127:0> :
        /*E*/ 0xD6DFC4CDF2FBE0E99E978C85BAB3A8A1<127:0> :
        /*D*/ 0x7D746F6659504B42353C272E1118030A<127:0> :
        /*C*/ 0xEDE4FFF6C9C0DBD2A5ACB7BE8188939A<127:0> :
        /*B*/ 0x3039222B141D060F78716A635C554E47<127:0> :
        /*A*/ 0xA0A9B2BB848D969FE8E1FAF3CCC5DED7<127:0> :
        /*9*/ 0x0B0219102F263D34434A5158676E757C<127:0> :
        /*8*/ 0x9B928980BFB6ADA4D3DAC1C8F7FEE5EC<127:0> :
        /*7*/ 0xAAA3B8B18E879C95E2EBF0F9C6CFD4DD<127:0> :
        /*6*/ 0x3A3328211E170C05727B6069565F444D<127:0> :
        /*5*/ 0x9198838AB5BCA7AED9D0CBC2FDF4EFE6<127:0> :
        /*4*/ 0x0108131A252C373E49405B526D647F76<127:0> :
        /*3*/ 0xDCD5CEC7F8F1EAE3949D868FB0B9A2AB<127:0> :
        /*2*/ 0x4C455E5768617A73040D161F2029323B<127:0> :
        /*1*/ 0xE7EEF5FCC3CAD1D8AFA6BDB48B829990<127:0> :
        /*0*/ 0x777E656C535A41483F362D241B120900<127:0>
    );
    return FFmul_09<UInt(b)*8+:8>;
```

5.446 shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =========

bits(8) FFmul0B(bits(8) b)
    bits(256*8) FFmul_0B = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
        /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
        /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
        /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
        /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
        /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
        /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
        /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
        /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
        /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
        /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
        /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
        /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
        /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
        /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
        /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
    );
    return FFmul_0B<UInt(b)*8+:8>;
```

5.447 shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =========

bits(8) FFmul0D(bits(8) b)
    bits(256*8) FFmul_0D = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x979A8D80A3AEB9B4FFF2E5E8CBC6D1DC<127:0> :
        /*E*/ 0x474A5D50737E69642F2235381B16010C<127:0> :
        /*D*/ 0x2C21363B1815020F44495E53707D6A67<127:0> :
        /*C*/ 0xFCF1E6EBC8C5D2DF94998E83A0ADBAB7<127:0> :
        /*B*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCB1<127:0> :
        /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
        /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
        /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
        /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
        /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
        /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt(b)*8+:8>;
```

5.448 shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =========

bits(8) FFmul0E(bits(8) b)
    bits(256*8) FFmul_0E = (
        /*       F E D C B A 9 8 7 6 5 4 3 2 1 0       */
        /*F*/ 0x8D83919FB5BBA9A7FDF3E1EFC5CBD9D7<127:0> :
        /*E*/ 0x6D63717F555B49471D13010F252B3937<127:0> :
        /*D*/ 0x56584A446E60727C26283A341E10020C<127:0> :
        /*C*/ 0xB6B8AAA48E80929CC6C8DAD4FEF0E2EC<127:0> :
        /*B*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<127:0> :
        /*9*/ 0xFBF5E7E9C3CDDFD18B859799B3BDAFA1<127:0> :
        /*8*/ 0x1B150709232D3F316B657779535D4F41<127:0> :
        /*7*/ 0xCCC2D0DEF4FAE8E6BCB2A0AE848A9896<127:0> :
        /*6*/ 0x2C22303E141A08065C52404E646A7876<127:0> :
        /*5*/ 0x17190B052F21333D67697B755F51434D<127:0> :
```

```
        /*4*/ 0xF7F9EBE5CFC1D3DD87899B95BFB1A3AD<127:0> :
        /*3*/ 0x616F7D735957454B111F0D032927353B<127:0> :
        /*2*/ 0x818F9D93B9B7A5ABF1FFEDE3C9C7D5DB<127:0> :
        /*1*/ 0xBAB4A6A8828C9E90CAC4D6D8F2FCEEE0<127:0> :
        /*0*/ 0x5A544648626C7E702A243638121C0E00<127:0>
    );
    return FFmul_0E<UInt(b)*8+:8>;
```

5.449 shared/functions/crypto/HaveAESExt

```
// HaveAESExt()
// ============
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
    return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

5.450 shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// ====================
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

5.451 shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =============
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

5.452 shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// ===============
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

5.453 shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =============
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

5.454 shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// ===============
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion(ARMv8p2) || !(HaveSHA1Ext() && HaveSHA256Ext()) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

5.455 shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// ============
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

5.456 shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// ============
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
    if !HasArchVersion(ARMv8p2) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

5.457 shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

5.458 shared/functions/crypto/SHA256hash

```
// SHA256hash()
// ============

bits(128) SHA256hash(bits (128) X, bits(128) Y, bits(128) W, boolean part1)
    bits(32) chs, maj, t;

    for e = 0 to 3
        chs = SHAchoose(Y<31:0>, Y<63:32>, Y<95:64>);
        maj = SHAmajority(X<31:0>, X<63:32>, X<95:64>);
        t = Y<127:96> + SHAhashSIGMA1(Y<31:0>) + chs + Elem[W, e, 32];
        X<127:96> = t + X<127:96>;
        Y<127:96> = t + SHAhashSIGMA0(X<31:0>) + maj;
        <Y, X> = ROL(Y : X, 32);
    return (if part1 then X else Y);
```

5.459 shared/functions/crypto/SHAchoose

```
// SHAchoose()
// ===========

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
    return (((y EOR z) AND x) EOR z);
```

5.460 shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// ===============

bits(32) SHAhashSIGMA0(bits(32) x)
    return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

5.461 shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// ===============

bits(32) SHAhashSIGMA1(bits(32) x)
    return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

5.462 shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =============

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
    return ((x AND y) OR ((x OR y) AND z));
```

5.463 shared/functions/crypto/SHAparity

```
// SHAparity()
// ===========

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
    return (x EOR y EOR z);
```

5.464 shared/functions/crypto/Sbox

```
// Sbox()
// ======
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    bits(2048) sboxstring =
        ↪0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491ef987a33540b43edcfac62e4b31ca

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

5.465 shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

5.466 shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

5.467 shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// ========================

// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

5.468 shared/functions/exclusive/ExclusiveMonitorsStatus

```
// Returns '0' to indicate success if the last memory write by this PE was to
// the same physical address region endorsed by ExclusiveMonitorsPass().
// Returns '1' to indicate failure if address translation resulted in a different
// physical address.
bit ExclusiveMonitorsStatus();
```

5.469 shared/functions/exclusive/IsExclusiveGlobal

```
// Return TRUE if the global Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

5.470 shared/functions/exclusive/IsExclusiveLocal

```
// Return TRUE if the local Exclusives monitor for processorid includes all of
// the physical address region of size bytes starting at paddress.
boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

5.471 shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at paddress in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

5.472 shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at paddress in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

5.473 shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

5.474 shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// ====================

boolean AArch32.HaveHPDExt()
    return HasArchVersion(ARMv8p2);
```

5.475 shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// ====================

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

5.476 shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// ================
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support";
```

5.477 shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// ====================
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension";
```

5.478 shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// ==========================
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension";
```

5.479 shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()
// ===============

boolean HaveAtomicExt()
    return HasArchVersion(ARMv8p1);
```

5.480 shared/functions/extension/HaveCapabilitiesExt

```
// HaveCapabilitiesExt()
// =====================
// Returns TRUE if the Capabilities extension is implemented and FALSE otherwise.

boolean HaveCapabilitiesExt()
    return TRUE;
```

5.481 shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// ==============================

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

5.482 shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =============
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has Dot Product extension";
```

5.483 shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// ================
// Returns TRUE if support for the OS Double Lock is implemented.

boolean HaveDoubleLock()
    return boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented";
```

5.484 shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()
// ===========================

boolean HaveExtendedECDebugEvents()
    return HasArchVersion(ARMv8p2);
```

5.485 shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()
// =============================

boolean HaveExtendedExecuteNeverExt()
    return HasArchVersion(ARMv8p2);
```

5.486 shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// ================================
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate to FP32 instructions,
// and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    if !HaveFP16Ext() then return FALSE;
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

5.487 shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()
// =============

boolean HaveHPMDExt()
    return HasArchVersion(ARMv8p1);
```

5.488 shared/functions/extension/HaveIESB

```
// HaveIESB()
// ==========

boolean HaveIESB()
    return (HaveRASExt() &&
            boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

5.489 shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =============
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has MPAM extension");
```

5.490 shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()
// ================================

boolean HaveNoSecurePMUDisableOverride()
    return HasArchVersion(ARMv8p2);
```

5.491 shared/functions/extension/HavePANExt

```
// HavePANExt()
// ============

boolean HavePANExt()
    return HasArchVersion(ARMv8p1);
```

5.492 shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()
// =================================

boolean HavePageBasedHardwareAttributes()
    return HasArchVersion(ARMv8p2);
```

5.493 shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()
// ===============

boolean HavePrivATExt()
    return HasArchVersion(ARMv8p2);
```

5.494 shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()
// ================

boolean HaveQRDMLAHExt()
    return HasArchVersion(ARMv8p1);

```

```
boolean HaveAccessFlagUpdateExt()
    return HasArchVersion(ARMv8p1);

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

5.495 shared/functions/extension/HaveRASExt

```
// HaveRASExt()
// ============

boolean HaveRASExt()
    return (HasArchVersion(ARMv8p2) ||
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

5.496 shared/functions/extension/HaveSBExt

```
// HaveSBExt()
// ===========
// Returns TRUE if support for SB is implemented, and FALSE otherwise.

boolean HaveSBExt()
    return boolean IMPLEMENTATION_DEFINED "Has SB extension";
```

5.497 shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()
// =============
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.

boolean HaveSSBSExt()
    return boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
```

5.498 shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// ==========================

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);
```

5.499 shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// ==============
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

5.500 shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// ============

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

5.501 shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =================

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

5.502 shared/functions/extension/InsertIESBBeforeException

```
// If SCTLR_ELx.IESB is 1 when an exception is generated to ELx, any pending Unrecoverable
// SError interrupt must be taken before executing any instructions in the exception handler.
// However, this can be before the branch to the exception handler is made.
boolean InsertIESBBeforeException(bits(2) el);
```

5.503 shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =======
// Single-precision add following BFloat16 computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) op2)
    bits(32) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN();
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0');    // Positive sign when Round to Odd
            else
                result = BFRound(result_value);

    return result;
```

5.504 shared/functions/float/bfloat/BFMatMulAdd

```
// BFMatMulAdd()
// =============
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2)
    assert N == 128;

    bits(N) result;
    bits(32) sum, prod0, prod1;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 1
                prod0 = BFMul(Elem[op1, 4*i + 2*k + 0, 16], Elem[op2, 4*j + 2*k + 0, 16]);
                prod1 = BFMul(Elem[op1, 4*i + 2*k + 1, 16], Elem[op2, 4*j + 2*k + 1, 16]);
                sum   = BFAdd(sum, BFAdd(prod0, prod1));
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

5.505 shared/functions/float/bfloat/BFMul

```
// BFMul()
// =======
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMul(bits(16) op1, bits(16) op2)
    bits(32) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN();
    else
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = BFRound(value1*value2);

    return result;
```

5.506 shared/functions/float/bfloat/BFRound

```
// BFRound()
// =========
// Converts a real number OP into a single-precision value using the
// Round to Odd rounding mode and following BFloat16 computation behaviors.

bits(32) BFRound(real op)
    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126;  E = 8;  F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign);      // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result;
```

5.507 shared/functions/float/bfloat/BFUnpack

```
// BFUnpack()
// ==========
// Unpacks a BFloat16 or single-precision value into its type,
// sign bit and real number that it represents.
// The real number result has the correct sign for numbers and infinities,
// is very large in magnitude for infinities, and is 0.0 for NaNs.
// (These values are chosen to simplify the description of
// comparisons and conversions.)

(FPType, bit, real) BFUnpack(bits(N) fpval)
    assert N IN {16,32};

    if N == 16 then
        sign   = fpval<15>;
        exp    = fpval<14:7>;
        frac   = fpval<6:0> : Zeros(16);
    else  // N == 32
        sign   = fpval<31>;
        exp    = fpval<30:23>;
        frac   = fpval<22:0>;

    if IsZero(exp) then
        fptype = FPType_Zero;  value = 0.0;    // Fixed Flush to Zero
    elsif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType_Infinity;  value = 2.0^1000000;
        else    // no SNaN for BF16 arithmetic
            fptype = FPType_QNaN; value = 0.0;
    else
        fptype = FPType_Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

5.508 shared/functions/float/bfloat/FPConvertBF

```
// FPConvertBF()
// =============
// Converts a single-precision OP to BFloat16 value with rounding controlled by ROUNDING.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr, FPRounding rounding)
    bits(32) result;    // BF16 value in top 16 bits

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if fptype == FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCVBF(value, fpcr, rounding);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<31:16>;
```

```

// FPConvertBF()
// =============
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRType fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

5.509 shared/functions/float/bfloat/FPRoundCVBF

```
// FPRoundCVBF()
// =============
// Converts a real number OP into a BFloat16 value using the supplied rounding mode RMODE.

bits(32) FPRoundCVBF(real op, FPCRType fpcr, FPRounding rounding)
    boolean isbfloat = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat);
```

5.510 shared/functions/float/fixedtofp/FixedToFP

```
// FixedToFP()
// ===========

// Convert M-bit fixed point OP with FBITS fractional bits to
// N-bit precision floating point, controlled by UNSIGNED and ROUNDING.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

5.511 shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =======

bits(N) FPAbs(bits(N) op)
    assert N IN {16,32,64};
    return '0' : op<N-2:0>;
```

5.512 shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =======

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);  inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);     zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

5.513 shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// ===========

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = '0011';
        if type1==FPType_SNaN || type2==FPType_SNaN || signal_nans then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else  // value1 > value2
            result = '0010';
    return result;
```

5.514 shared/functions/float/fpcompareeq/FPCompareEQ

```
// FPCompareEQ()
// =============

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        if type1==FPType_SNaN || type2==FPType_SNaN then
            FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 == value2);
    return result;
```

5.515 shared/functions/float/fpcomparege/FPCompareGE

```
// FPCompareGE()
// =============

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 >= value2);
    return result;
```

5.516 shared/functions/float/fpcomparegt/FPCompareGT

```
// FPCompareGT()
// =============

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    if type1==FPType_SNaN || type1==FPType_QNaN || type2==FPType_SNaN || type2==FPType_QNaN then
        result = FALSE;
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUnpack()
        result = (value1 > value2);
    return result;
```

5.517 shared/functions/float/fpconvert/FPConvert

```
// FPConvert()
// ===========

// Convert floating point OP with N-bit precision to M-bit precision,
// with rounding controlled by ROUNDING.
// This is used by the FP-to-FP conversion instructions and so for
// half-precision data ignores FZ16, but observes AHP.

bits(M) FPConvert(bits(N) op, FPCRType fpcr, FPRounding rounding)
    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        if alt_hp then
            result = FPZero(sign);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN();
        else
            result = FPConvertNaN(op);
        if fptype == FPType_SNaN || alt_hp then
            FPProcessException(FPExc_InvalidOp,fpcr);
    elsif fptype == FPType_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);
    return result;

// FPConvert()
// ===========

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

5.518 shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// ==============
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = op<N-1>;

    // Unpack payload from input NaN
    case N of
        when 64 frac = op<50:0>;
        when 32 frac = op<21:0>:Zeros(29);
        when 16 frac = op<8:0>:Zeros(42);

    // Repack payload into output NaN, while
    // converting an SNaN to a QNaN.
    case M of
        when 64 result = sign:Ones(M-52):frac;
        when 32 result = sign:Ones(M-23):frac<50:29>;
        when 16 result = sign:Ones(M-10):frac<50:42>;

    return result;
```

5.519 shared/functions/float/fpcrtype/FPCRType

```
type FPCRType;
```

5.520 shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// ============

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    case rm of
        when '00' return FPRounding_TIEAWAY; // A
        when '01' return FPRounding_TIEEVEN; // N
        when '10' return FPRounding_POSINF;  // P
        when '11' return FPRounding_NEGINF;  // M
```

5.521 shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// ==================

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    case rmode of
        when '00' return FPRounding_TIEEVEN; // N
        when '01' return FPRounding_POSINF;  // P
        when '10' return FPRounding_NEGINF;  // M
        when '11' return FPRounding_ZERO;    // Z
```

5.522 shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// ==============

bits(N) FPDefaultNaN()
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    sign = '0';
    bits(E) exp  = Ones(E);
    bits(F) frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

5.523 shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =======

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);
    return result;
```

5.524 shared/functions/float/fpexc/FPExc

```
enumeration FPExc       {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                         FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};
```

5.525 shared/functions/float/fpinfinity/FPInfinity

```
// FPInfinity()
// ============

bits(N) FPInfinity(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    bits(E) exp  = Ones(E);
    bits(F) frac = Zeros(F);
    return sign : exp : frac;
```

5.526 shared/functions/float/fpmax/FPMax

```
// FPMax()
// =======

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
```

```
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            result = FPRound(value, fpcr);
    return result;
```

5.527 shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =============

bits(N) FPMaxNormal(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

5.528 shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// ==========

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // treat a single quiet-NaN as -Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('1');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('1');

    return FPMax(op1, op2, fpcr);
```

5.529 shared/functions/float/fpmin/FPMin

```
// FPMin()
// =======

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPType_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FPType_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            result = FPRound(value, fpcr);
    return result;
```

5.530 shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// ==========

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    // Treat a single quiet-NaN as +Infinity
    if type1 == FPType_QNaN && type2 != FPType_QNaN then
        op1 = FPInfinity('0');
    elsif type1 != FPType_QNaN && type2 == FPType_QNaN then
        op2 = FPInfinity('0');

    return FPMin(op1, op2, fpcr);
```

5.531 shared/functions/float/fpmul/FPMul

```
// FPMul()
// =======

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

5.532 shared/functions/float/fpmuladd/FPMulAdd

```
// FPMulAdd()
// ==========
//
// Calculates addend + op1*op2 with a single rounding.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr);

    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType_Infinity);  zeroA = (typeA == FPType_Zero);

        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP  = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
```

```
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);

    return result;
```

5.533 shared/functions/float/fpmuladdh/FPMulAddH

```
// FPMulAddH()
// ===========

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRType fpcr)
    assert N IN {32,64};
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    inf1 = (type1 == FPType_Infinity); zero1 = (type1 == FPType_Zero);
    inf2 = (type2 == FPType_Infinity); zero2 = (type2 == FPType_Zero);
    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr);
    if typeA == FPType_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    if !done then
        infA = (typeA == FPType_Infinity); zeroA = (typeA == FPType_Zero);
        // Determine sign and type product will have if it does not cause an Invalid
        // Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;
        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        if (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP) then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');
        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);
        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr);
    return result;
```

5.534 shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =================

(boolean, bits(N)) FPProcessNaNs3H(FPType type1, FPType type2, FPType type3, bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3, FPCRType fpcr)
    assert N IN {32,64};
    bits(N) result;
    if type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr));
    else
        done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);
```

5.535 shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// ========

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);
    return result;
```

5.536 shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =======

bits(N) FPNeg(bits(N) op)
    assert N IN {16,32,64};
    return NOT(op<N-1>) : op<N-2:0>;
```

5.537 shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// ================

bits(N) FPOnePointFive(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

5.538 shared/functions/float/fpprocessexception/FPProcessException

```
// FPProcessException()
// ====================
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRType fpcr)
    // Determine the cumulative exception bit number
    case exception of
        when FPExc_InvalidOp     cumul = 0;
        when FPExc_DivideByZero  cumul = 1;
        when FPExc_Overflow      cumul = 2;
        when FPExc_Underflow     cumul = 3;
        when FPExc_Inexact       cumul = 4;
        when FPExc_InputDenorm   cumul = 7;
    enable = cumul + 8;
    if fpcr<enable> == '1' then
        // Trapping of the exception enabled.
        // It is IMPLEMENTATION DEFINED whether the enable bit may be set at all, and
        // if so then how exceptions may be accumulated before calling FPTrappedException()
        IMPLEMENTATION_DEFINED "floating-point trap handling";
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

5.539 shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// ==============

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};

    case N of
        when 16 topfrac =  9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then  // DefaultNaN requested
        result = FPDefaultNaN();
    return result;
```

5.540 shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// ===============
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2,
                                 bits(N) op1, bits(N) op2,
                                 FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
```

```
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

5.541 shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// ================
//
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                  bits(N) op1, bits(N) op2, bits(N) op3,
                                  FPCRType fpcr)
    assert N IN {16,32,64};
    if type1 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_SNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    elsif type1 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type1, op1, fpcr);
    elsif type2 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type2, op2, fpcr);
    elsif type3 == FPType_QNaN then
        done = TRUE;  result = FPProcessNaN(type3, op3, fpcr);
    else
        done = FALSE;  result = Zeros();  // 'Don't care' result
    return (done, result);
```

5.542 shared/functions/float/fprecipestimate/FPRecipEstimate

```
// FPRecipEstimate()
// =================

bits(N) FPRecipEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(operand, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero(sign);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif (
            (N == 16 && Abs(value) < 2.0^-16) ||
            (N == 32 && Abs(value) < 2.0^-128) ||
            (N == 64 && Abs(value) < 2.0^-1024)
          ) then
        case FPRoundingMode(fpcr) of
            when FPRounding_TIEEVEN
                overflow_to_inf = TRUE;
            when FPRounding_POSINF
                overflow_to_inf = (sign == '0');
            when FPRounding_NEGINF
                overflow_to_inf = (sign == '1');
            when FPRounding_ZERO
                overflow_to_inf = FALSE;
        result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
        FPProcessException(FPExc_Overflow, fpcr);
        FPProcessException(FPExc_Inexact, fpcr);
    elsif ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16))
          && (
               (N == 16 && Abs(value) >= 2.0^14) ||
               (N == 32 && Abs(value) >= 2.0^126) ||
               (N == 64 && Abs(value) >= 2.0^1022)
             ) then
        // Result flushed to zero of correct sign
        result = FPZero(sign);
        FPSR.UFC = '1';
    else
        // Scale to a fixed point value in the range 0.5 <= x < 1.0 in steps of 1/512, and
        // calculate result exponent. Scaled value has copied sign bit,
        // exponent = 1022 = double-precision biased version of -1,
        // fraction = original fraction
        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            if fraction<51> == '0' then
                exp = -1;
                fraction = fraction<49:0>:'00';
            else
                fraction = fraction<50:0>:'0';

        integer scaled = UInt('1':fraction<51:44>);

        case N of
            when 16 result_exp =   29 - exp; // In range 29-30 = -1 to 29+1 = 30
            when 32 result_exp =  253 - exp; // In range 253-254 = -1 to 253+1 = 254
            when 64 result_exp = 2045 - exp; // In range 2045-2046 = -1 to 2045+1 = 2046

        // scaled is in range 256..511 representing a fixed-point number in range [0.5..1.0)
        estimate = RecipEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit,
        // high-order bits from estimate, and exponent calculated above.

        fraction = estimate<7:0> : Zeros(44);
        if result_exp == 0 then
            fraction = '1' : fraction<51:1>;
        elsif result_exp == -1 then
            fraction = '01' : fraction<51:2>;
            result_exp = 0;

        case N of
            when 16 result = sign : result_exp<N-12:0> : fraction<51:42>;
            when 32 result = sign : result_exp<N-25:0> : fraction<51:29>;
            when 64 result = sign : result_exp<N-54:0> : fraction<51:0>;

    return result;
```

5.543 shared/functions/float/fprecipestimate/RecipEstimate

```
// Compute estimate of reciprocal of 9-bit fixed-point number
//
// a is in range 256 .. 511 representing a number in the range 0.5 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipEstimate(integer a)
    assert 256 <= a && a < 512;
    a = a*2+1; // round to nearest
    integer b = (2 ^ 19) DIV a;
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

5.544 shared/functions/float/fprecpx/FPRecpX

```
// FPRecpX()
// =========

bits(N) FPRecpX(bits(N) op, FPCRType fpcr)
```

```
    assert N IN {16,32,64};

    case N of
        when 16 esize =  5;
        when 32 esize =  8;
        when 64 esize = 11;

    bits(N)           result;
    bits(esize)       exp;
    bits(esize)       max_exp;
    bits(N-(esize+1)) frac = Zeros();

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    (fptype,sign,value) = FPUnpack(op, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    else
        if IsZero(exp) then // Zero and denormals
            result = sign:max_exp:frac;
        else // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

5.545 shared/functions/float/fpround/FPRound

```
// FPRound()
// =========
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.AHP = '0';
    boolean isbfloat = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat);

// Convert a real number OP into an N-bit floating-point value using the
// supplied rounding mode RMODE.

bits(N) FPRoundBase(real op, FPCRType fpcr, FPRounding rounding, boolean isbfloat)
    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    if N == 16 then
        minimum_exp = -14;  E = 5;  F = 10;
    elsif N == 32 && isbfloat then
        minimum_exp = -126;  E = 8;  F = 7;
    elsif N == 32 then
        minimum_exp = -126;  E = 8;  F = 23;
    else  // N == 64
        minimum_exp = -1022;  E = 11;  F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;
    exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;

    // Deal with flush-to-zero.
    if ((fpcr.FZ == '1' && N != 16) || (fpcr.FZ16 == '1' && N == 16)) && exponent < minimum_exp then
        // Flush-to-zero never generates a trapped exception
        FPSR.UFC = '1';
        return FPZero(sign);

    // Start creating the exponent value for the result. Start by biasing the actual exponent
    // so that the minimum exponent becomes 1, lower values 0 (indicating possible underflow).
    biased_exp = Max(exponent - minimum_exp + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Underflow occurs if exponent is too small before rounding, and result is inexact or
    // the Underflow exception is trapped.
    if biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') then
        FPProcessException(FPExc_Underflow, fpcr);

    // Round result according to rounding mode.
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        when FPRounding_POSINF
            round_up = (error != 0.0 && sign == '0');
            overflow_to_inf = (sign == '0');
        when FPRounding_NEGINF
            round_up = (error != 0.0 && sign == '1');
            overflow_to_inf = (sign == '1');
        when FPRounding_ZERO, FPRounding_ODD
            round_up = FALSE;
            overflow_to_inf = FALSE;

    if round_up then
        int_mant = int_mant + 1;
        if int_mant == 2^F then      // Rounded up from denormalized to normalized
            biased_exp = 1;
        if int_mant == 2^(F+1) then  // Rounded up to next exponent
            biased_exp = biased_exp + 1;  int_mant = int_mant DIV 2;

    // Handle rounding to odd aka Von Neumann rounding
    if error != 0.0 && rounding == FPRounding_ODD then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if N != 16 || fpcr.AHP == '0' then  // Single, double or IEEE half precision
        if biased_exp >= 2^E - 1 then
            result = if overflow_to_inf then FPInfinity(sign) else FPMaxNormal(sign);
            FPProcessException(FPExc_Overflow, fpcr);
            error = 1.0;  // Ensure that an Inexact exception occurs
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));
    else                                     // Alternative half precision
        if biased_exp >= 2^E then
            result = sign : Ones(N-1);
            FPProcessException(FPExc_InvalidOp, fpcr);
            error = 0.0;  // Ensure that an Inexact exception does not occur
        else
            result = sign : biased_exp<E-1:0> : int_mant<F-1:0> : Zeros(N-(E+F+1));

    // Deal with Inexact exception.
    if error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;

// FPRound()
// =========

bits(N) FPRound(real op, FPCRType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```

5.546 shared/functions/float/fpround/FPRoundCV

```
// FPRoundCV()
// ===========
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRType fpcr, FPRounding rounding)
    fpcr.FZ16 = '0';
    boolean isbfloat = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat);
```

5.547 shared/functions/float/fprounding/FPRounding

```
enumeration FPRounding  {FPRounding_TIEEVEN, FPRounding_POSINF,
                         FPRounding_NEGINF,  FPRounding_ZERO,
                         FPRounding_TIEAWAY, FPRounding_ODD};
```

5.548 shared/functions/float/fproundingmode/FPRoundingMode

```
// FPRoundingMode()
// ================

// Return the current floating-point rounding mode.

FPRounding FPRoundingMode(FPCRType fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

5.549 shared/functions/float/fproundint/FPRoundInt

```
// FPRoundInt()
// ============

// Round OP to nearest integral floating point value using rounding mode ROUNDING.
// If EXACT is TRUE, set FPSR.IXC if result is not numerically equal to OP.

bits(N) FPRoundInt(bits(N) op, FPCRType fpcr, FPRounding rounding, boolean exact)
    assert rounding != FPRounding_ODD;
    assert N IN {16,32,64};

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        // extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
            when FPRounding_POSINF
                round_up = (error != 0.0);
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = (error != 0.0 && int_result < 0);
            when FPRounding_TIEAWAY
                round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact
        if real_result == 0.0 then
            result = FPZero(sign);
        else
            result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 && exact then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

5.550 shared/functions/float/fproundintn/FPRoundIntN

```
// FPRoundIntN()
// =============

bits(N) FPRoundIntN(bits(N) op, FPCRType fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // Unpack using FPCR to determine if subnormals are flushed-to-zero
    (fptype,sign,value) = FPUnpack(op, fpcr);

    if fptype IN {FPType_SNaN, FPType_QNaN, FPType_Infinity} then
        if N == 32 then
            exp = 126 + intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        else
            exp = 1022+intsize;
            result = '1':exp<(E-1):0>:Zeros(F);
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    else
        // Extract integer component
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment
        case rounding of
            when FPRounding_TIEEVEN
                round_up = error > 0.5 || (error == 0.5 && int_result<0> == '1');
            when FPRounding_POSINF
                round_up = error != 0.0;
            when FPRounding_NEGINF
                round_up = FALSE;
            when FPRounding_ZERO
                round_up = error != 0.0 && int_result < 0;
            when FPRounding_TIEAWAY
                round_up = error > 0.5 || (error == 0.5 && int_result >= 0);

        if round_up then int_result = int_result + 1;

        if int_result > 2^(intsize-1)-1 || int_result < -1*2^(intsize-1) then
            if N == 32 then
                exp = 126 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            else
                exp = 1022 + intsize;
                result = '1':exp<(E-1):0>:Zeros(F);
            FPProcessException(FPExc_InvalidOp, fpcr);
            // this case shouldn't set Inexact
            error = 0.0;

        else
            // Convert integer value into an equivalent real value
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact
            if real_result == 0.0 then
                result = FPZero(sign);
            else
                result = FPRound(real_result, fpcr, FPRounding_ZERO);

        // Generate inexact exceptions
        if error != 0.0 then
            FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

5.551 shared/functions/float/fprsqrtestimate/FPRSqrtEstimate

```
// FPRSqrtEstimate()
// =================

bits(N) FPRSqrtEstimate(bits(N) operand, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(operand, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr);
    elsif fptype == FPType_Zero then
        result = FPInfinity(sign);
        FPProcessException(FPExc_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif fptype == FPType_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        case N of
            when 16
                fraction = operand<9:0> : Zeros(42);
                exp = UInt(operand<14:10>);
            when 32
                fraction = operand<22:0> : Zeros(29);
                exp = UInt(operand<30:23>);
            when 64
                fraction = operand<51:0>;
                exp = UInt(operand<62:52>);

        if exp == 0 then
            while fraction<51> == '0' do
                fraction = fraction<50:0> : '0';
                exp = exp - 1;
            fraction = fraction<50:0> : '0';

        if exp<0> == '0' then
            scaled = UInt('1':fraction<51:44>);
        else
            scaled = UInt('01':fraction<51:45>);

        case N of
            when 16 result_exp = (  44 - exp) DIV 2;
            when 32 result_exp = ( 380 - exp) DIV 2;
            when 64 result_exp = (3068 - exp) DIV 2;

        estimate = RecipSqrtEstimate(scaled);

        // estimate is in the range 256..511 representing a fixed point result in the range [1.0..2.0)
        // Convert to scaled floating point result with copied sign bit and high-order
        // fraction bits, and exponent calculated above.
        case N of
            when 16 result = '0' : result_exp<N-12:0> : estimate<7:0>:Zeros( 2);
            when 32 result = '0' : result_exp<N-25:0> : estimate<7:0>:Zeros(15);
            when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);
    return result;
```

5.552 shared/functions/float/fprsqrtestimate/RecipSqrtEstimate

```
// Compute estimate of reciprocal square root of 9-bit fixed-point number
//
// a is in range 128 .. 511 representing a number in the range 0.25 <= x < 1.0.
// result is in the range 256 .. 511 representing a number in the range in the range 1.0 to 511/256.

integer RecipSqrtEstimate(integer a)
    assert 128 <= a && a < 512;
    if a < 256 then // 0.25 .. 0.5
        a = a*2+1;     // a in units of 1/512 rounded to nearest
    else // 0.5 .. 1.0
        a = (a >> 1) << 1;   // discard bottom bit
        a = (a+1)*2;  // a in units of 1/256 rounded to nearest
    integer b = 512;
    while a*(b+1)*(b+1) < 2^28 do
        b = b+1;
    // b = largest b such that b < 2^14 / sqrt(a) do
    r = (b+1) DIV 2; // round to nearest
    assert 256 <= r && r < 512;
    return r;
```

5.553 shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// ========

bits(N) FPSqrt(bits(N) op, FPCRType fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPType_Zero then
        result = FPZero(sign);
    elsif fptype == FPType_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN();
        FPProcessException(FPExc_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), fpcr);
    return result;
```

5.554 shared/functions/float/fpsub/FPSub

```
// FPSub()
// =======

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCRType fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType_Infinity);
        inf2 = (type2 == FPType_Infinity);
        zero1 = (type1 == FPType_Zero);
        zero2 = (type2 == FPType_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN();
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then  // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding_NEGINF then '1' else '0';
                result = FPZero(result_sign);
            else
                result = FPRound(result_value, fpcr, rounding);
    return result;
```

5.555 shared/functions/float/fpthree/FPThree

```
// FPThree()
// =========

bits(N) FPThree(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

5.556 shared/functions/float/fptofixed/FPToFixed

```
// FPToFixed()
// ===========

// Convert N-bit precision floating point OP to M-bit fixed point with
// FBITS fractional bits, controlled by UNSIGNED and ROUNDING.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCRType fpcr, FPRounding rounding)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    assert fbits >= 0;
    assert rounding != FPRounding_ODD;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero
    (fptype,sign,value) = FPUnpack(op, fpcr);

    // If NaN, set cumulative flag or take exception
    if fptype == FPType_SNaN || fptype == FPType_QNaN then
        FPProcessException(FPExc_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_result<0> == '1'));
        when FPRounding_POSINF
            round_up = (error != 0.0);
        when FPRounding_NEGINF
            round_up = FALSE;
        when FPRounding_ZERO
            round_up = (error != 0.0 && int_result < 0);
        when FPRounding_TIEAWAY
            round_up = (error > 0.5 || (error == 0.5 && int_result >= 0));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc_Inexact, fpcr);

    return result;
```

5.557 shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =======

bits(N) FPTwo(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

5.558 shared/functions/float/fptype/FPType

```
enumeration FPType      {FPType_Nonzero, FPType_Zero, FPType_Infinity,
                         FPType_QNaN, FPType_SNaN};
```

5.559 shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// ==========
//
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRType fpcr)
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

5.560 shared/functions/float/fpunpack/FPUnpackBase

```
// FPUnpackBase()
// ==============
//
// Unpack a floating-point number into its type, sign bit and the real number
// that it represents. The real number result has the correct sign for numbers
// and infinities, is very large in magnitude for infinities, and is 0.0 for
// NaNs. (These values are chosen to simplify the description of comparisons
// and conversions.)
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCRType fpcr)
    assert N IN {16,32,64};

    if N == 16 then
        sign   = fpval<15>;
        exp16  = fpval<14:10>;
        frac16 = fpval<9:0>;
        if IsZero(exp16) then
            // Produce zero if value is zero or flush-to-zero is selected
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType_Zero;  value = 0.0;
            else
                fptype = FPType_Nonzero;  value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then  // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 then

        sign   = fpval<31>;
        exp32  = fpval<30:23>;
        frac32 = fpval<22:0>;
        if IsZero(exp32) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac32) || fpcr.FZ == '1' then
                fptype = FPType_Zero;  value = 0.0;
                if !IsZero(frac32) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
                fptype = FPType_Nonzero;  value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
        elsif IsOnes(exp32) then
            if IsZero(frac32) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac32<22> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

    else // N == 64

        sign   = fpval<63>;
        exp64  = fpval<62:52>;
        frac64 = fpval<51:0>;
        if IsZero(exp64) then
            // Produce zero if value is zero or flush-to-zero is selected.
            if IsZero(frac64) || fpcr.FZ == '1' then
                fptype = FPType_Zero;  value = 0.0;
                if !IsZero(frac64) then  // Denormalized input flushed to zero
                    FPProcessException(FPExc_InputDenorm, fpcr);
            else
```

```
                fptype = FPType_Nonzero;  value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elsif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPType_Infinity;  value = 2.0^1000000;
            else
                fptype = if frac64<51> == '1' then FPType_QNaN else FPType_SNaN;
                value = 0.0;
        else
            fptype = FPType_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;
    return (fptype, sign, value);
```

5.561 shared/functions/float/fpunpack/FPUnpackCV

```
// FPUnpackCV()
// ============
//
// Used for FP <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCRType fpcr)
    fpcr.FZ16 = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr);
    return (fp_type, sign, value);
```

5.562 shared/functions/float/fpzero/FPZero

```
// FPZero()
// ========

bits(N) FPZero(bit sign)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
    frac = Zeros(F);
    return sign : exp : frac;
```

5.563 shared/functions/float/vfpexpandimm/VFPExpandImm

```
// VFPExpandImm()
// ==============

bits(N) VFPExpandImm(bits(8) imm8)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - E - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    return sign : exp : frac;
```

5.564 shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// ==============
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

5.565 shared/functions/memory/AArch64.BranchAddr

```
// AArch64.BranchAddr()
// ====================
// Return the virtual address with tag bits removed for storing to the program counter.

bits(64) AArch64.BranchAddr(bits(64) vaddress)
    assert !UsingAArch32();
    msbit = AddrTop(vaddress, PSTATE.EL);
    if msbit == 63 then
        return vaddress;
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>);
    else
        return ZeroExtend(vaddress<msbit:0>);
```

5.566 shared/functions/memory/AccType

```
enumeration AccType {AccType_NORMAL, AccType_VEC,        // Normal loads and stores
                     AccType_STREAM, AccType_VECSTREAM,  // Streaming loads and stores
                     AccType_ATOMIC, AccType_ATOMICRW,   // Atomic loads and stores
                     AccType_ORDERED, AccType_ORDEREDRW, // Load-Acquire and Store-Release
                     AccType_ORDEREDATOMIC,              // Load-Acquire and Store-Release with atomic access
                     AccType_ORDEREDATOMICRW,
                     AccType_LIMITEDORDERED,             // Load-LOAcquire and Store-LORelease
                     AccType_UNPRIV,                     // Load and store unprivileged
                     AccType_IFETCH,                     // Instruction fetch
                     AccType_PTW,                        // Page table walk
                     // Other operations
                     AccType_DC,                         // Data cache maintenance
                     AccType_DC_UNPRIV,                  // Data cache maintenance instruction used at EL0
                     AccType_IC,                         // Instruction cache maintenance
                     AccType_DCZVA,                      // DC ZVA instructions
                     AccType_AT};                        // Address translation
```

5.567 shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (
    AccType acctype,
    MPAMinfo mpam,
    boolean page_table_walk,
    boolean secondstage,
    boolean s2fs1walk,
    integer level
)
```

5.568 shared/functions/memory/AddrTop

```
// AddrTop()
// =========
// Return the MSB number of a virtual address in the stage 1 translation regime for "el".
// If EL1 is using AArch64 then addresses from EL0 using AArch32 are zero-extended to 64 bits.

integer AddrTop(bits(64) address, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        // AArch64 translation regime.
        case regime of
            when EL1
                tbi = (if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0);
            when EL2
                if HaveVirtHostExt() && ELIsInHost(el) then
                    tbi = (if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0);
                else
                    tbi = TCR_EL2.TBI;
            when EL3
                tbi = TCR_EL3.TBI;

    return (if tbi == '1' then 55 else 63);
```

5.569 shared/functions/memory/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord      fault,    // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    bits(64)         vaddress
)
```

5.570 shared/functions/memory/Allocation

```
constant bits(2) MemHint_No = '00';     // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01';     // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10';     // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11';    // Read-Allocate, Write-Allocate
```

5.571 shared/functions/memory/BigEndian

```
// BigEndian()
// ===========

boolean BigEndian()
    boolean bigend;
    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[].E0E != '0');
    else
        bigend = (SCTLR[].EE != '0');
    return bigend;
```

5.572 shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// ==================

bits(width) BigEndianReverse (bits(width) value)
    assert width IN {8, 16, 32, 64, 128};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

5.573 shared/functions/memory/BranchAddr

```
// BranchAddr()
// ============
// Return the virtual address with tag bits removed for storing to the program counter.

Capability BranchAddr(Capability c, bits(2) el)
    assert !UsingAArch32();
    bits(64) cap_value = CapGetValue(c);
    msbit = AddrTop(cap_value, el);

    if CapIsSealed(c) then
        c = CapWithTagClear(c);

    if msbit == 63 then
        return c;
    elsif (el IN {EL0, EL1} || IsInHost()) && cap_value<msbit> == '1' then
        assert msbit == 55;
        return CapSetFlags(c, SignExtend(cap_value<msbit:0>));
    else
        assert msbit == 55;
        return CapSetFlags(c, ZeroExtend(cap_value<msbit:0>));
```

5.574 shared/functions/memory/Cacheability

```
constant bits(2) MemAttr_NC = '00';     // Non-cacheable
constant bits(2) MemAttr_WT = '10';     // Write-through
constant bits(2) MemAttr_WB = '11';     // Write-back
```

5.575 shared/functions/memory/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// ========================

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype IN {AccType_IFETCH, AccType_IC});
    accdesc.page_table_walk = FALSE;
    return accdesc;
```

5.576 shared/functions/memory/CreateAccessDescriptorPTW

```
// CreateAccessDescriptorPTW()
// ===========================

AccessDescriptor CreateAccessDescriptorPTW(AccType acctype, boolean secondstage,
                                           boolean s2fs1walk, integer level)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype IN {AccType_IFETCH, AccType_IC});
    accdesc.page_table_walk = TRUE;
    accdesc.s2fs1walk = s2fs1walk;
    accdesc.secondstage = secondstage;
    accdesc.level = level;
    return accdesc;
```

5.577 shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

5.578 shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types);
```

5.579 shared/functions/memory/DescriptorUpdate

```
type DescriptorUpdate is (
    boolean AF,                  // AF needs to be set
    boolean AP,                  // AP[2] / S2AP[2] will be modified
    boolean SC,                  // SC needs to be set
    AddressDescriptor descaddr   // Descriptor to be updated
)
```

5.580 shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

5.581 shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// ==============
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tbi = if address<55> == '1' then TCR_EL1.TBI1 else TCR_EL1.TBI0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
            else
                tbi = TCR_EL2.TBI;
        when EL3
            tbi = TCR_EL3.TBI;

    return tbi;
```

5.582 shared/functions/memory/EffectiveTGEN

```
// EffectiveTGEN()
// ===============
// Returns the effective TGEN of a virtual address in the stage 1 translation regime
// for "el".

bit EffectiveTGEN(bits(64) address, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    case regime of
        when EL1
            tgen = if address<55> == '1' then CCTLR_EL1.TGEN1 else CCTLR_EL1.TGEN0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tgen = if address<55> == '1' then CCTLR_EL2.TGEN1 else CCTLR_EL2.TGEN0;
            else
                tgen = CCTLR_EL2.TGEN0;
        when EL3
            tgen = CCTLR_EL3.TGEN0;

    return tgen;
```

5.583 shared/functions/memory/Fault

```
enumeration Fault {Fault_None,
                   Fault_AccessFlag,
                   Fault_Alignment,
                   Fault_Background,
                   Fault_Domain,
                   Fault_Permission,
                   Fault_Translation,
                   Fault_AddressSize,
                   Fault_SyncExternal,
                   Fault_SyncExternalOnWalk,
                   Fault_SyncParity,
                   Fault_SyncParityOnWalk,
                   Fault_AsyncParity,
                   Fault_AsyncExternal,
                   Fault_Debug,
                   Fault_TLBConflict,
                   Fault_HWUpdateAccessFlag,
                   Fault_CapTag,
                   Fault_CapSeal,
                   Fault_CapBounds,
                   Fault_CapPerm,
                   Fault_CapPagePerm,
                   Fault_Lockdown,
                   Fault_Exclusive,
                   Fault_ICacheMaint};
```

5.584 shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    statuscode,  // Fault Status
                     AccType  acctype,      // Type of access that faulted
                     bits(48) ipaddress,    // Intermediate physical address
                     boolean  s2fs1walk,    // Is on a Stage 1 page table walk
                     boolean  write,        // TRUE for a write, FALSE for a read
                     integer  level,        // For translation, access flag and permission faults
                     bit      extflag,      // IMPLEMENTATION DEFINED syndrome for external aborts
                     boolean  secondstage,  // Is a Stage 2 abort
                     bits(4)  domain,       // Domain number, AArch32 only
                     bits(2)  errortype,    // [Armv8.2 RAS] AArch32 AET or AArch64 SET
                     bits(4)  debugmoe)     // Debug method of entry, from AArch32 only

type PARTIDtype = bits(16);
type PMGtype = bits(8);

type MPAMinfo is (
     bit mpam_ns,
     PARTIDtype partid,
     PMGtype pmg
)
```

5.585 shared/functions/memory/FullAddress

```
type FullAddress is (
    bits(48) address,
    bit      NS                    // '0' = Secure, '1' = Non-secure
)
```

5.586 shared/functions/memory/Hint_Prefetch

```
// Signals the memory system that memory accesses of type HINT to or from the specified address are
// likely in the near future. The memory system may take some action to speed up the memory
// accesses when they do occur, such as pre-loading the the specified address into one or more
// caches as indicated by the innermost cache level target (0=L1, 1=L2, etc) and non-temporal hint
// stream. Any or all prefetch hints may be treated as a NOP. A prefetch hint must not cause a
// synchronous abort due to Alignment or Translation faults and the like. Its only effect on
// software-visible state should be on caches and TLBs associated with address, which must be
// accessible by reads, writes or execution, as defined in the translation regime of the current
// Exception level. It is guaranteed not to access Device memory.
// A Prefetch_EXEC hint must not result in an access that could not be performed by a speculative
// instruction fetch, therefore if all associated MMUs are disabled, then it cannot access any
// memory location that cannot be accessed by instruction fetches.
Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

5.587 shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

5.588 shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

5.589 shared/functions/memory/MemAttrHints

```
type MemAttrHints is (
    bits(2) attrs,  // See MemAttr_*, Cacheability attributes
    bits(2) hints,  // See MemHint_*, Allocation hints
    boolean transient
)
```

5.590 shared/functions/memory/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

5.591 shared/functions/memory/MemoryAttributes

```
type MemoryAttributes is (
    MemType      memtype,

    DeviceType   device,     // For Device memory types
    MemAttrHints inner,      // Inner hints and attributes
    MemAttrHints outer,      // Outer hints and attributes
    boolean      readtagzero,      // Tag is read as zero
    boolean      readtagfault,     // Fault if reading valid tag
    bit          readtagfaulttgen, // Value of TGENy leading to fault
    boolean      writetagfault,    // Fault if writing valid tag
    boolean      iss2writetagfault,// Fault if writing valid tag is due to stage 2
    boolean      shareable,
    boolean      outershareable
)
```

5.592 shared/functions/memory/Permissions

```
type Permissions is (
    bits(3) ap,   // Access permission bits
    bit     xn,   // Execute-never bit
    bit     xxn,  // [Armv8.2] Extended execute-never bit for stage 2
    bit     pxn   // Privileged execute-never bit
)
```

5.593 shared/functions/memory/PrefetchHint

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

5.594 shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
SpeculativeStoreBypassBarrierToPA();
```

5.595 shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
SpeculativeStoreBypassBarrierToVA();
```

5.596 shared/functions/memory/TLBRecord

```
type TLBRecord is (
    Permissions       perms,
    bit               nG,             // '0' = Global, '1' = not Global
    bits(4)           domain,         // AArch32 only
    boolean           contiguous,     // Contiguous bit from page table
    integer           level,          // AArch32 Short-descriptor format: Indicates Section/Page
    integer           blocksize,      // Describes size of memory translated in KBytes
    DescriptorUpdate  descupdate,     // [Armv8.1] Context for h/w update of table descriptor
    bit               CnP,            // [Armv8.2] TLB entry can be shared between different PEs
    AddressDescriptor addrdesc
)
```

5.597 shared/functions/memory/_Mem

```
// These two _Mem[] accessors are the hardware operations which perform single-copy atomic,
// aligned, little-endian memory accesses of size bytes from/to the underlying physical
// memory array of bytes.
//
// The functions address the array using desc.paddress which supplies:
// * A 48-bit physical address
// * A single NS bit to select between Secure and Non-secure parts of the array.
//
// The accdesc descriptor describes the access type: normal, exclusive, ordered, streaming,
// etc and other parameters required to access the physical memory or for setting syndrome
// register in the event of an external abort.
bits(8*size) _Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc];

_Mem[AddressDescriptor desc, integer size, AccessDescriptor accdesc] = bits(8*size) value;
```

5.598 shared/functions/mpam/DefaultMPAMinfo

```
// DefaultMPAMinfo
// ===============
// Returns default MPAM info.  If secure is TRUE return default Secure
// MPAMinfo, otherwise return default Non-secure MPAMinfo.

MPAMinfo DefaultMPAMinfo(boolean secure)
    MPAMinfo DefaultInfo;
    DefaultInfo.mpam_ns = if secure then '0' else '1';
    DefaultInfo.partid  = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;
```

5.599 shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

5.600 shared/functions/mpam/DefaultPMG

```
constant PMGtype    DefaultPMG = 0<7:0>;
```

5.601 shared/functions/mpam/GenMPAMcurEL

```
// GenMPAMcurEL
// ============
// Returns MPAMinfo for the current EL and security state.
// InD is TRUE instruction access and FALSE otherwise.
// May be called if MPAM is not implemented (but in an version that supports
// MPAM), MPAM is disabled, or in AArch32.  In AArch32, convert the mode to
// EL if can and use that to drive MPAM information generation.  If mode
// cannot be converted, MPAM is not implemented, or MPAM is disabled return
// default MPAM information for the current security state.

MPAMinfo GenMPAMcurEL(boolean InD)
    bits(2) mpamel;
    boolean validEL;
    boolean securempam;
    securempam = IsSecure();
    if HaveMPAMExt() && MPAMisEnabled() then
        mpamel = PSTATE.EL;
        return genMPAM(UInt(mpamel), InD, securempam);
    return DefaultMPAMinfo(securempam);
```

5.602 shared/functions/mpam/MAP_vPARTID

```
// MAP_vPARTID
// ===========
// Performs conversion of virtual PARTID into physical PARTID
// Contains all of the error checking and implementation
// choices for the conversion.

(PARTIDtype, boolean) MAP_vPARTID(PARTIDtype vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDtype ret;
    boolean err;
    integer virt    = UInt( vpartid );
    integer vpmrmax = UInt( MPAMIDR_EL1.VPMR_MAX );

    // vpartid_max is largest vpartid supported
    integer vpartid_max = (4 * vpmrmax) + 3;

    // One of many ways to reduce vpartid to value less than vpartid_max.
    if virt > vpartid_max then
        virt = virt MOD (vpartid_max+1);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elsif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DefaultPARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt( MPAMIDR_EL1.PARTID_MAX );
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;
    return (ret, err);
```

5.603 shared/functions/mpam/MPAMisEnabled

```
// MPAMisEnabled
// =============
// Returns TRUE if MPAMisEnabled.

boolean MPAMisEnabled()
    el = HighestEL();
    case el of
        when EL3 return MPAM3_EL3.MPAMEN == '1';
        when EL2 return MPAM2_EL2.MPAMEN == '1';
        when EL1 return MPAM1_EL1.MPAMEN == '1';
```

5.604 shared/functions/mpam/MPAMisVirtual

```
// MPAMisVirtual
// =============
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMisVirtual(integer el)
    return ( MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
            ( HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0' ) &&
            (( el == 0 && MPAMHCR_EL2.EL0_VPMEN == '1' ) ||
             ( el == 1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```

5.605 shared/functions/mpam/genMPAM

```
// genMPAM
// =======
// Returns MPAMinfo for exception level el.
// If InD is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// Produces a Secure PARTID if Secure is TRUE and a Non-secure PARTID otherwise.

MPAMinfo genMPAM(integer el, boolean InD, boolean secure)
    MPAMinfo returnInfo;
    PARTIDtype partidel;
    boolean perr;
    boolean gstplk = (el == 0 && EL2Enabled() &&
                      MPAMHCR_EL2.GSTAPP_PLK == '1' && HCR_EL2.TGE == '0');
    integer eff_el = if gstplk then 1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel  = genPMG(eff_el, InD, perr);
    returnInfo.mpam_ns = if secure then '0' else '1';
    returnInfo.partid  = partidel;
    returnInfo.pmg     = groupel;
    return returnInfo;
```

5.606 shared/functions/mpam/genMPAMel

```
// genMPAMel
// =========
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMel(bits(2) el, boolean InD)
    boolean secure = IsSecure();
    boolean securempam = secure;
    if HaveMPAMExt() && MPAMisEnabled() then
        return genMPAM(UInt(el), InD, securempam);
    return DefaultMPAMinfo(securempam);
```

5.607 shared/functions/mpam/genPARTID

```
// genPARTID
// =========
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(integer el, boolean InD)
    PARTIDtype partidel = getMPAM_PARTID(el, InD);

    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(partidel) > partid_max then
        return (DefaultPARTID, TRUE);

    if MPAMisVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

5.608 shared/functions/mpam/genPMG

```
// genPMG
// ======
// Returns PMG for exception level el and I- or D-side (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(integer el, boolean InD, boolean partid_err)
    integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);

    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

5.609 shared/functions/mpam/getMPAM_PARTID

```
// getMPAM_PARTID
// ==============
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register.  Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(integer MPAMn, boolean InD)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_I;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_I;
            when 0 partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDtype UNKNOWN;
    else
        case MPAMn of
            when 3 partid = MPAM3_EL3.PARTID_D;
            when 2 partid = if el2avail then MPAM2_EL2.PARTID_D else Zeros();
            when 1 partid = MPAM1_EL1.PARTID_D;
            when 0 partid = MPAM0_EL1.PARTID_D;
            otherwise partid = PARTIDtype UNKNOWN;
    return partid;
```

5.610 shared/functions/mpam/getMPAM_PMG

```
// getMPAM_PMG
// ===========
// Returns a PMG from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register.  Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(integer MPAMn, boolean InD)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when 3 pmg = MPAM3_EL3.PMG_I;
            when 2 pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            when 1 pmg = MPAM1_EL1.PMG_I;
            when 0 pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGtype UNKNOWN;
    else
        case MPAMn of
            when 3 pmg = MPAM3_EL3.PMG_D;
            when 2 pmg = if el2avail then MPAM2_EL2.PMG_D else Zeros();
            when 1 pmg = MPAM1_EL1.PMG_D;
            when 0 pmg = MPAM0_EL1.PMG_D;
            otherwise pmg = PMGtype UNKNOWN;
    return pmg;
```

5.611 shared/functions/mpam/mapvpmw

```
// mapvpmw
// =======
// Map a virtual PARTID into a physical PARTID using
// the MPAMVPMn_EL2 registers.
// vpartid is now assumed in-range and valid (checked by caller)
// returns physical PARTID from mapping entry.

PARTIDtype mapvpmw(integer vpartid)
    bits(64) vpmw;
    integer wd = vpartid DIV 4;
    case wd of
        when 0 vpmw = MPAMVPM0_EL2;
        when 1 vpmw = MPAMVPM1_EL2;
        when 2 vpmw = MPAMVPM2_EL2;
        when 3 vpmw = MPAMVPM3_EL2;
        when 4 vpmw = MPAMVPM4_EL2;
        when 5 vpmw = MPAMVPM5_EL2;
        when 6 vpmw = MPAMVPM6_EL2;
        when 7 vpmw = MPAMVPM7_EL2;
        otherwise vpmw = Zeros(64);
    // vpme_lsb selects LSB of field within register
    integer vpme_lsb = (vpartid REM 4) * 16;
    return vpmw<vpme_lsb +: 16>;
```

5.612 shared/functions/registers/BranchTo

```
// BranchTo()
// ==========

// Set program counter to a new address, with a branch type
// In AArch64 state the address might include a tag in the top eight bits.

BranchTo(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
        PCC = CapSetValue(PCC, ZeroExtend(target));
    else
        assert N == 64 && !UsingAArch32();
        _PC = AArch64.BranchAddr(target<63:0>);
        PCC = CapSetValue(PCC, AArch64.BranchAddr(target<63:0>));
    return;
```

5.613 shared/functions/registers/BranchToAddr

```
// BranchToAddr()
// ==============

// Set program counter to a new address, with a branch type
// In AArch64 state the address does not include a tag in the top eight bits.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target);
        PCC = CapSetValue(PCC, ZeroExtend(target));
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
        PCC = CapSetValue(PCC, target<63:0>);
    return;
```

5.614 shared/functions/registers/BranchToOffset

```
// BranchToOffset()
// ================
// Branch to an offset from the PC

BranchToOffset(bits(64) offset, BranchType branch_type)
    Hint_Branch(branch_type);
    assert  !UsingAArch32();
    Capability new_pcc = CapAdd(PCC, offset);
    PCC    = BranchAddr(new_pcc, PSTATE.EL);
    _PC    = CapGetValue(PCC);
    return;
```

5.615 shared/functions/registers/BranchType

```
enumeration BranchType {
    BranchType_DIRCALL,     // Direct Branch with link
    BranchType_INDCALL,     // Indirect Branch with link
    BranchType_ERET,        // Exception return (indirect)
    BranchType_DBGEXIT,     // Exit from Debug state
    BranchType_RET,         // Indirect branch with function return hint
    BranchType_DIR,         // Direct branch
    BranchType_INDIR,       // Indirect branch
    BranchType_EXCEPTION,   // Exception entry
    BranchType_RESET,       // Reset
    BranchType_UNKNOWN};    // Other
```

5.616 shared/functions/registers/Hint_Branch

```
BranchToCapability(Capability target, BranchType branch_type)
    Hint_Branch(branch_type);
    assert !UsingAArch32();

    _PC   = AArch64.BranchAddr(CapGetValue(target));
    PCC = BranchAddr(target, PSTATE.EL);
    return;

BranchXToCapability(Capability target, BranchType branch_type)
    PSTATE.C64 = target<0>;
    target<0> = '0';
    BranchToCapability(target, branch_type);

// Report the hint passed to BranchTo() and BranchToAddr(), for consideration when processing
// the next instruction.
Hint_Branch(BranchType hint);
```

5.617 shared/functions/registers/NextInstrAddr

```
// Return address of the sequentially next instruction.
bits(N) NextInstrAddr();
```

5.618 shared/functions/registers/ResetExternalDebugRegisters

```
// Reset the External Debug registers in the Core power domain.
ResetExternalDebugRegisters(boolean cold_reset);
```

5.619 shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// ===============
// Return address of the current instruction.

bits(N) ThisInstrAddr()
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

5.620 shared/functions/registers/_PC

```
bits(64) _PC;
```

5.621 shared/functions/registers/_R

```
array Capability _R[0..30];
```

5.622 shared/functions/registers/_V

```
array bits(128) _V[0..31];
```

5.623 shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// ============================

bits(32) SPSR[]
    bits(32) result;
    case PSTATE.EL of
        when EL1          result = SPSR_EL1;
        when EL2          result = SPSR_EL2;
        when EL3          result = SPSR_EL3;
        otherwise         Unreachable();
    return result;

// SPSR[] - assignment form
// ========================

SPSR[] = bits(32) value
    case PSTATE.EL of
        when EL1          SPSR_EL1 = value;
        when EL2          SPSR_EL2 = value;
        when EL3          SPSR_EL3 = value;
        otherwise         Unreachable();
    return;
```

5.624 shared/functions/system/ArchVersion

```
enumeration ArchVersion {
    ARMv8p0
    , ARMv8p1
    , ARMv8p2
};
```

5.625 shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// ====================
// Clear the Event Register of this PE

ClearEventRegister()
    EventRegister = '0';
    return;
```

5.626 shared/functions/system/ClearPendingPhysicalSError

```
// Clear a pending physical SError interrupt
ClearPendingPhysicalSError();
```

5.627 shared/functions/system/ClearPendingVirtualSError

```
// Clear a pending virtual SError interrupt
ClearPendingVirtualSError();
```

5.628 shared/functions/system/ConditionHolds

```
// ConditionHolds()
// ===============
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
    // Evaluate base condition.
    case cond<3:1> of
        when '000' result = (PSTATE.Z == '1');                          // EQ or NE
        when '001' result = (PSTATE.C == '1');                          // CS or CC
        when '010' result = (PSTATE.N == '1');                          // MI or PL
        when '011' result = (PSTATE.V == '1');                          // VS or VC
        when '100' result = (PSTATE.C == '1' && PSTATE.Z == '0');       // HI or LS
        when '101' result = (PSTATE.N == PSTATE.V);                     // GE or LT
        when '110' result = (PSTATE.N == PSTATE.V && PSTATE.Z == '0');  // GT or LE
        when '111' result = TRUE;                                       // AL

    // Condition flag values in the set '111x' indicate always true
    // Otherwise, invert condition if necessary.
    if cond<0> == '1' && cond != '1111' then
        result = !result;

    return result;
```

5.629 shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
ConsumptionOfSpeculativeDataBarrier();
```

5.630 shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// ================

InstrSet CurrentInstrSet()

    if UsingAArch32() then
        result = if PSTATE.T == '0' then InstrSet_A32 else InstrSet_T32;
        // PSTATE.J is RES0. Implementation of T32EE or Jazelle state not permitted.
    else
        result = InstrSet_A64;
    return result;
```

5.631 shared/functions/system/EL0

```
constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '00';
```

5.632 shared/functions/system/EL2Enabled

```
// EL2Enabled()
// ============
// Returns TRUE if EL2 is present and access is Non-secure, FALSE otherwise.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_EL3.NS == '1');
```

5.633 shared/functions/system/ELFromSPSR

```
// ELFromSPSR()
// ============

// Convert an SPSR value encoding to an Exception level.
// Returns (valid,EL):
//   'valid' is TRUE if 'spsr<4:0>' encodes a valid mode for the current state.
//   'EL'    is the Exception level decoded from 'spsr'.

(boolean,bits(2)) ELFromSPSR(bits(32) spsr)
    if spsr<4> == '0' then                      // AArch64 state
        el = spsr<3:2>;
        if HighestELUsingAArch32() then         // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then                  // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then               // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then  // for EL0, M[0] must be 0
            valid = FALSE;
        elsif el == EL2 && HaveEL(EL3) && SCR_EL3.NS == '0' then
            valid = FALSE;                      // EL2 only valid in Non-secure state
        else
            valid = TRUE;
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid,el);
```

5.634 shared/functions/system/ELIsInHost

```
// ELIsInHost()
// ============

boolean ELIsInHost(bits(2) el)
    return (!IsSecureBelowEL3() && HaveVirtHostExt() && !ELUsingAArch32(EL2) &&
            HCR_EL2.E2H == '1' && (el == EL2 || (el == EL0 && HCR_EL2.TGE == '1')));
```

5.635 shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====================

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

5.636 shared/functions/system/ELStateUsingAArch32K

```
// ELStateUsingAArch32K()
// ======================

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    // Returns (known, aarch32):
    //   'known'   is FALSE for EL0 if the current Exception level is not EL0 and EL1 is
    //             using AArch64, since it cannot determine the state of EL0; TRUE otherwise.
    //   'aarch32' is TRUE if the specified Exception level is using AArch32; FALSE otherwise.
    if !HaveAArch32EL(el) then
        return (TRUE, FALSE);                   // Exception level is using AArch64
    elsif HighestELUsingAArch32() then
        return (TRUE, TRUE);                    // Highest Exception level, and therefore all levels
           ↪are using AArch32
    elsif el == HighestEL() then
        return (TRUE, FALSE);                   // This is highest Exception level, so is using AArch64

    // Remainder of function deals with the interprocessing cases when highest Exception level is using
        ↪AArch64

    boolean aarch32 = boolean UNKNOWN;
    boolean known = TRUE;

    aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0';
    aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) && !secure && HCR_EL2.RW == '0' &&
                                            !(HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' &&
                                                 ↪HaveVirtHostExt())));
    if el == EL0 && !aarch32_at_el1 then       // Only know if EL0 using AArch32 from PSTATE
        if PSTATE.EL == EL0 then
            aarch32 = PSTATE.nRW == '1';       // EL0 controlled by PSTATE
        else
            known = FALSE;                     // EL0 state is UNKNOWN
    else
        aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1,EL0});

    if !known then aarch32 = boolean UNKNOWN;
    return (known, aarch32);
```

5.637 shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// ================

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

5.638 shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =================

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

5.639 shared/functions/system/EndOfInstruction

```
// Terminate processing of the current instruction.
EndOfInstruction();
```

5.640 shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state
EnterLowPowerState();
```

5.641 shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

5.642 shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// ==================
// Return a PSR value which represents the current PSTATE

bits(32) GetPSRFromPSTATE()
    bits(32) spsr = Zeros();
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if HavePANExt() then spsr<22> = PSTATE.PAN;
    spsr<20>    = PSTATE.IL;
    if HaveCapabilitiesExt() then spsr<26> = PSTATE.C64;
    if HaveUAOExt() then spsr<23> = PSTATE.UAO;
    spsr<21>    = PSTATE.SS;
    if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
    spsr<9:6>   = PSTATE.<D,A,I,F>;
    spsr<4>     = PSTATE.nRW;
    spsr<3:2>   = PSTATE.EL;
    spsr<0>     = PSTATE.SP;
    return spsr;
```

5.643 shared/functions/system/HasArchVersion

```
// HasArchVersion()
// ================
// Return TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
    return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

5.644 shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// ===============

boolean HaveAArch32EL(bits(2) el)
    // Return TRUE if Exception level 'el' supports AArch32 in this implementation
    if !HaveEL(el) then
        return FALSE;                    // The Exception level is not implemented
    elsif !HaveAnyAArch32() then
        return FALSE;                    // No Exception level can use AArch32
    elsif HighestELUsingAArch32() then
        return TRUE;                     // All Exception levels are using AArch32
    elsif el == HighestEL() then
        return FALSE;                    // The highest Exception level is using AArch64
    elsif el == EL0 then
        return TRUE;                     // EL0 must support using AArch32 if any AArch32
    return boolean IMPLEMENTATION_DEFINED;
```

5.645 shared/functions/system/HaveAnyAArch32

```
// HaveAnyAArch32()
// ================
// Return TRUE if AArch32 state is supported at any Exception level

boolean HaveAnyAArch32()
    return boolean IMPLEMENTATION_DEFINED;
```

5.646 shared/functions/system/HaveAnyAArch64

```
// HaveAnyAArch64()
// ================
// Return TRUE if AArch64 state is supported at any Exception level

boolean HaveAnyAArch64()
    return !HighestELUsingAArch32();
```

5.647 shared/functions/system/HaveEL

```
// HaveEL()
// ========
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1,EL0} then
        return TRUE;                             // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

5.648 shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// ==========================
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    case el of
        when EL3
            assert secure;
            return HaveEL(EL3);
        when EL2
            return !secure && HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                    (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

5.649 shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =============
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

5.650 shared/functions/system/HighestEL

```
// HighestEL()
// ===========
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

5.651 shared/functions/system/HighestELUsingAArch32

```
// HighestELUsingAArch32()
// =======================
// Return TRUE if configured to boot into AArch32 operation

boolean HighestELUsingAArch32()
    if !HaveAnyAArch32() then return FALSE;
    return boolean IMPLEMENTATION_DEFINED;       // e.g. CFG32SIGNAL == HIGH
```

5.652 shared/functions/system/Hint_Yield

```
// Provides a hint that the task performed by a thread is of low
// importance so that it could yield to improve overall performance.
Hint_Yield();
```

5.653 shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// ========================

boolean IllegalExceptionReturn(bits(32) spsr)

    // Check for illegal return:
    //   * To an unimplemented Exception level.
    //   * To EL2 in Secure state.
    //   * To EL0 using AArch64 state, with SPSR.M[0]==1.
    //   * To AArch64 state with SPSR.M[1]==1.
    //   * To AArch32 state with an illegal value of SPSR.M.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (spsr<4> == '1');

    // Check for illegal return:
    //   * To EL1, EL2 or EL3 with register width specified in the SPSR different from the
    //     Execution state used in the Exception level being returned to, as determined by
    //     the SCR_EL3.RW or HCR_EL2.RW bits, or as configured from reset.
    //   * To EL0 using AArch64 state when EL1 is using AArch32 state as determined by the
    //     SCR_EL3.RW or HCR_EL2.RW bits or as configured from reset.
    //   * To AArch64 state from AArch32 state (should be caught by above)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 in Non-secure state when HCR.TGE is set
    if HaveEL(EL2) && target == EL1 && !IsSecureBelowEL3() && HCR_EL2.TGE == '1' then return TRUE;
    return FALSE;
```

5.654 shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

5.655 shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

5.656 shared/functions/system/InterruptPending

```
// InterruptPending()
// ==================
// Return TRUE if there are any pending physical or virtual interrupts, and FALSE otherwise

boolean InterruptPending()
    return IsPhysicalSErrorPending() || IsVirtualSErrorPending();
```

5.657 shared/functions/system/IsEventRegisterSet

```
// IsEventRegisterSet()
// ====================
// Return TRUE if the Event Register of this PE is set, and FALSE otherwise

boolean IsEventRegisterSet()
    return EventRegister == '1';
```

5.658 shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =============
// Returns TRUE if given exception level is the highest exception level implemented
```

```

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

5.659 shared/functions/system/IsInHost

```
// IsInHost()
// ==========

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

5.660 shared/functions/system/IsPhysicalSErrorPending

```
// Return TRUE if a physical SError interrupt is pending
boolean IsPhysicalSErrorPending();
```

5.661 shared/functions/system/IsSecure

```
// IsSecure()
// ==========
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

5.662 shared/functions/system/IsSecureBelowEL3

```
// IsSecureBelowEL3()
// ==================
// Return TRUE if an Exception level below EL3 is in Secure state
// or would be following an exception return to that level.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// That is, if at AArch64 EL3 or in AArch32 Monitor mode, whether an
// exception return would pass to Secure or Non-secure state.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR_GEN[].NS == '0';
    elsif HaveEL(EL2) then
        return FALSE;
    else
        // TRUE if processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

5.663 shared/functions/system/IsVirtualSErrorPending

```
// Return TRUE if a virtual SError interrupt is pending
boolean IsVirtualSErrorPending();
```

5.664 shared/functions/system/Mode_Bits

```
constant bits(5) M32_User    = '10000';
constant bits(5) M32_FIQ     = '10001';
constant bits(5) M32_IRQ     = '10010';
constant bits(5) M32_Svc     = '10011';
constant bits(5) M32_Monitor = '10110';
constant bits(5) M32_Abort   = '10111';
constant bits(5) M32_Hyp     = '11010';
constant bits(5) M32_Undef   = '11011';
constant bits(5) M32_System  = '11111';
```

5.665 shared/functions/system/PSTATE

```
ProcState PSTATE;
```

5.666 shared/functions/system/PrivilegeLevel

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

5.667 shared/functions/system/ProcState

```
type ProcState is (
    bits (1) N,        // Negative condition flag
    bits (1) Z,        // Zero condition flag
    bits (1) C,        // Carry condition flag
    bits (1) V,        // oVerflow condition flag
    bits (1) D,        // Debug mask bit                     [AArch64 only]
    bits (1) A,        // SError interrupt mask bit
    bits (1) I,        // IRQ mask bit
    bits (1) F,        // FIQ mask bit
    bits (1) PAN,      // Privileged Access Never Bit        [v8.1]
    bits (1) UAO,      // User Access Override               [v8.2]
    bits (1) C64,      // Current instruction set state      [Morello only]
    bits (1) SS,       // Software step bit
    bits (1) IL,       // Illegal Execution state bit
    bits (2) EL,       // Exception Level
    bits (1) nRW,      // not Register Width: 0=64, 1=32
    bits (1) SP,       // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,        // Cumulative saturation flag         [AArch32 only]
    bits (4) GE,       // Greater than or Equal flags        [AArch32 only]
    bits (1) SSBS,     // Speculative Store Bypass Safe
    bits (8) IT,       // If-then bits, RES0 in CPSR         [AArch32 only]
    bits (1) J,        // J bit, RES0                        [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,        // T32 bit, RES0 in CPSR              [AArch32 only]
    bits (1) E,        // Endianness bit                     [AArch32 only]
    bits (5) M         // Mode field                         [AArch32 only]
)
```

5.668 shared/functions/system/SCRType

```
type SCRType;
```

5.669 shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =========

SCRType SCR_GEN[]
    assert HaveEL(EL3);
    return ZeroExtend(SCR_EL3);
```

5.670 shared/functions/system/SendEvent

```
// Signal an event to all PEs in a multiprocessor system to set their Event Registers.
// When a PE executes the SEV instruction, it causes this function to be executed
SendEvent();
```

5.671 shared/functions/system/SendEventLocal

```
// SendEventLocal()
// ================
// Set the local Event Register of this PE.
// When a PE executes the SEVL instruction, it causes this function to be executed

SendEventLocal()
    EventRegister = '1';
    return;
```

5.672 shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// ==================
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(32) spsr)
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IllegalExceptionReturn(spsr) then
        PSTATE.IL = '1';
        if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
        // PSTATE.C64 is unchanged if access to Morello is trapped at the target EL.
        if HaveCapabilitiesExt() && !IsAccessToCapabilitiesEnabledAtEL(PSTATE.EL) then
            PSTATE.C64 = '0';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        PSTATE.nRW = '0';
        PSTATE.EL  = spsr<3:2>;
        PSTATE.SP  = spsr<0>;
        if HaveSSBSExt() then PSTATE.SSBS = spsr<12>;
        if HaveCapabilitiesExt() then
            if IsAccessToCapabilitiesEnabledAtEL(PSTATE.EL) then
                PSTATE.C64 = spsr<26>;
            else
                PSTATE.C64 = '0';

    // If PSTATE.IL is set, it is CONSTRAINED UNPREDICTABLE whether the T bit is set to zero or
    // copied from SPSR.
    if PSTATE.IL == '1' && PSTATE.nRW == '1' then
        if ConstrainUnpredictableBool(Unpredictable_ILZEROT) then spsr<5> = '0';

    // State that is reinstated regardless of illegal exception return
    PSTATE.<N,Z,C,V> = spsr<31:28>;
    if HavePANExt() then PSTATE.PAN = spsr<22>;
    if HaveUAOExt() then PSTATE.UAO = spsr<23>;
    PSTATE.<D,A,I,F> = spsr<9:6>;
    return;
```

5.673 shared/functions/system/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

5.674 shared/functions/system/SpeculationBarrier

```
SpeculationBarrier();
```

5.675 shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

5.676 shared/functions/system/SynchronizeErrors

```
// Implements the error synchronization event.
SynchronizeErrors();
```

5.677 shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

5.678 shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.
TakeUnmaskedSErrorInterrupts();
```

5.679 shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

5.680 shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

5.681 shared/functions/system/Unreachable

```
Unreachable()
    assert FALSE;
```

5.682 shared/functions/system/UsingAArch32

```
// UsingAArch32()
// ==============
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAnyAArch32() then assert !aarch32;
    if HighestELUsingAArch32() then assert aarch32;
    return aarch32;
```

5.683 shared/functions/system/WaitForEvent

```
// WaitForEvent()
// ==============
// PE suspends its operation and enters a low-power state
// if the Event Register is clear when the WFE is executed

WaitForEvent()
    if EventRegister == '0' then
        EnterLowPowerState();
    return;
```

5.684 shared/functions/system/WaitForInterrupt

```
// WaitForInterrupt()
// ==================
// PE suspends its operation to enter a low-power state
// until a WFI wake-up event occurs or the PE is reset

WaitForInterrupt()
    EnterLowPowerState();
    return;
```

5.685 shared/functions/unpredictable/ConstrainUnpredictable

```
// ConstrainUnpredictable()
// ========================
// Return the appropriate Constraint result to control the caller's behavior. The return value
// is IMPLEMENTATION DEFINED within a permitted list for each UNPREDICTABLE case.
// (The permitted list is determined by an assert or case statement at the call site.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.

Constraint ConstrainUnpredictable(Unpredictable which)
    case which of
        when Unpredictable_WBOVERLAPLD
            return Constraint_WBSUPPRESS; // return loaded value
        when Unpredictable_WBOVERLAPST
            return Constraint_NONE;     // store pre-writeback value
        when Unpredictable_LDPOVERLAP
            return Constraint_UNDEF;    // instruction is UNDEFINED
        when Unpredictable_BASEOVERLAP
            return Constraint_NONE;     // use original address
        when Unpredictable_DATAOVERLAP
            return Constraint_NONE;     // store original value
        when Unpredictable_DEVPAGE2
            return Constraint_FAULT;    // take an alignment fault
        when Unpredictable_INSTRDEVICE
            return Constraint_NONE;     // Do not take a fault
        when Unpredictable_RESCPACR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESMAIR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESTEXCB
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESDACR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESPRRR
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESVTCRS
            return Constraint_UNKNOWN;  // Map to UNKNOWN value
        when Unpredictable_RESTnSZ
            return Constraint_FORCE;    // Map to the limit value
        when Unpredictable_LARGEIPA
            return Constraint_FORCE;    // Restrict the inputsize to the PAMax value
        when Unpredictable_ESRCONDPASS
            return Constraint_FALSE;    // Report as "AL"
        when Unpredictable_ILZEROIT
            return Constraint_FALSE;    // Do not zero PSTATE.IT
        when Unpredictable_ILZEROT
            return Constraint_FALSE;    // Do not zero PSTATE.T
        when Unpredictable_BPVECTORCATCHPRI
            return Constraint_TRUE;     // Debug Vector Catch: match on 2nd halfword
        when Unpredictable_VCMATCHHALF
            return Constraint_FALSE;    // No match
        when Unpredictable_VCMATCHDAPA
            return Constraint_FALSE;    // No match on Data Abort or Prefetch abort
        when Unpredictable_WPMASKANDBAS
            return Constraint_FALSE;    // Watchpoint disabled
        when Unpredictable_WPBASCONTIGUOUS
            return Constraint_FALSE;    // Watchpoint disabled
        when Unpredictable_RESWPMASK
            return Constraint_DISABLED; // Watchpoint disabled
        when Unpredictable_WPMASKEDBITS
            return Constraint_FALSE;    // Watchpoint disabled
        when Unpredictable_RESBPWPCTRL
            return Constraint_DISABLED; // Breakpoint/watchpoint disabled
        when Unpredictable_BPNOTIMPL
            return Constraint_DISABLED; // Breakpoint disabled
        when Unpredictable_RESBPTYPE
            return Constraint_DISABLED; // Breakpoint disabled
        when Unpredictable_BPNOTCTXCMP
            return Constraint_DISABLED; // Breakpoint disabled
        when Unpredictable_BPMATCHHALF
            return Constraint_FALSE;    // No match
        when Unpredictable_BPMISMATCHHALF
            return Constraint_FALSE;    // No match
        when Unpredictable_RESTARTALIGNPC
            return Constraint_FALSE;    // Do not force alignment
        when Unpredictable_RESTARTZEROUPPERPC
            return Constraint_TRUE;     // Force zero extension
        when Unpredictable_ZEROUPPER
            return Constraint_TRUE;     // zero top halves of X registers
        when Unpredictable_ERETZEROUPPERPC
            return Constraint_TRUE;     // zero top half of PC
        when Unpredictable_A32FORCEALIGNPC
            return Constraint_FALSE;    // Do not force alignment
        when Unpredictable_SMD
            return Constraint_UNDEF;    // disabled SMC is Unallocated
        when Unpredictable_AFUPDATE     // AF update for alignment or permission fault
            return Constraint_TRUE;
        when Unpredictable_IESBinDebug  // Use SCTLR[].IESB in Debug state
            return Constraint_TRUE;
        when Unpredictable_BADPMSFCR    // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
            return Constraint_TRUE;
        when Unpredictable_CLEARERRITEZERO // Clearing sticky errors when instruction in flight
            return Constraint_FALSE;
        when Unpredictable_LINKTRANSFEROVERLAPLD // Link/transfer register overlap (load)
            return Constraint_UNKNOWN;
        when Unpredictable_LINKBASEOVERLAPLD // Link/base register overlap (load)
            return Constraint_UNKNOWN;
```

5.686 shared/functions/unpredictable/ConstrainUnpredictableBits

```
// ConstrainUnpredictableBits()
// ============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN.
// If the result is Constraint_UNKNOWN then the function also returns UNKNOWN value, but that
// value is always an allocated value; that is, one for which the behavior is not itself
// CONSTRAINED.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, Zeros(width));           // See notes; this is an example implementation only
    else
        return (c, bits(width) UNKNOWN);    // bits result not used
```

5.687 shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// ============================

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)

    c = ConstrainUnpredictable(which);
    assert c IN {Constraint_TRUE, Constraint_FALSE};
    return (c == Constraint_TRUE);
```

5.688 shared/functions/unpredictable/ConstrainUnpredictableInteger

```
// ConstrainUnpredictableInteger()
// ===============================

// This is a variant of ConstrainUnpredictable for when the result can be Constraint_UNKNOWN. If
// the result is Constraint_UNKNOWN then the function also returns an UNKNOWN value in the range
// low to high, inclusive.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, low);                // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN);    // integer result not used
```

5.689 shared/functions/unpredictable/Constraint

```
enumeration Constraint    {// General
                           Constraint_NONE,              // Instruction executes with
                                                         //   no change or side-effect to its described
                                                              ↪behavior
                           Constraint_UNKNOWN,           // Destination register has UNKNOWN value
                           Constraint_UNDEF,             // Instruction is UNDEFINED
                           Constraint_UNDEFEL0,          // Instruction is UNDEFINED at EL0 only
                           Constraint_NOP,               // Instruction executes as NOP
                           Constraint_TRUE,
                           Constraint_FALSE,
                           Constraint_DISABLED,
                           Constraint_UNCOND,            // Instruction executes unconditionally
                           Constraint_COND,              // Instruction executes conditionally
                           Constraint_ADDITIONAL_DECODE, // Instruction executes with additional decode
                           // Load-store
                           Constraint_WBSUPPRESS, Constraint_FAULT,
                           // IPA too large
                           Constraint_FORCE, Constraint_FORCENOSLCHECK};
```

5.690 shared/functions/unpredictable/Unpredictable

```
enumeration Unpredictable {// Writeback/transfer register overlap (load)
                           Unpredictable_WBOVERLAPLD,
                           // Writeback/transfer register overlap (store)
                           Unpredictable_WBOVERLAPST,
                           // Load Pair transfer register overlap
                           Unpredictable_LDPOVERLAP,
                           // Store-exclusive base/status register overlap
                           Unpredictable_BASEOVERLAP,
                           // Store-exclusive data/status register overlap
                           Unpredictable_DATAOVERLAP,
                           // Load-store alignment checks
                           Unpredictable_DEVPAGE2,
                           // Instruction fetch from Device memory
                           Unpredictable_INSTRDEVICE,
                           // Reserved CPACR value
                           Unpredictable_RESCPACR,
                           // Reserved MAIR value
                           Unpredictable_RESMAIR,
                           // Reserved TEX:C:B value
                           Unpredictable_RESTEXCB,
                           // Reserved PRRR value
                           Unpredictable_RESPRRR,
                           // Reserved DACR field
                           Unpredictable_RESDACR,
                           // Reserved VTCR.S value
                           Unpredictable_RESVTCRS,
                           // Reserved TCR.TnSZ value
                           Unpredictable_RESTnSZ,
                           // IPA size exceeds PA size
                           Unpredictable_LARGEIPA,
                           // Syndrome for a known-passing conditional A32 instruction
                           Unpredictable_ESRCONDPASS,
                           // Illegal State exception: zero PSTATE.IT
                           Unpredictable_ILZEROIT,
                           // Illegal State exception: zero PSTATE.T
                           Unpredictable_ILZEROT,
                           // Debug: prioritization of Vector Catch
                           Unpredictable_BPVECTORCATCHPRI,
                           // Debug Vector Catch: match on 2nd halfword
                           Unpredictable_VCMATCHHALF,
                           // Debug Vector Catch: match on Data Abort or Prefetch abort
                           Unpredictable_VCMATCHDAPA,
                           // Debug watchpoints: non-zero MASK and non-ones BAS
                           Unpredictable_WPMASKANDBAS,
                           // Debug watchpoints: non-contiguous BAS
                           Unpredictable_WPBASCONTIGUOUS,
                           // Debug watchpoints: reserved MASK
                           Unpredictable_RESWPMASK,
                           // Debug watchpoints: non-zero MASKed bits of address
                           Unpredictable_WPMASKEDBITS,
                           // Debug breakpoints and watchpoints: reserved control bits
                           Unpredictable_RESBPWPCTRL,
                           // Debug breakpoints: not implemented
                           Unpredictable_BPNOTIMPL,
                           // Debug breakpoints: reserved type
                           Unpredictable_RESBPTYPE,
                           // Debug breakpoints: not-context-aware breakpoint
                           Unpredictable_BPNOTCTXCMP,
                           // Debug breakpoints: match on 2nd halfword of instruction
                           Unpredictable_BPMATCHHALF,
                           // Debug breakpoints: mismatch on 2nd halfword of instruction
                           Unpredictable_BPMISMATCHHALF,
                           // Debug: restart to a misaligned AArch32 PC value
                           Unpredictable_RESTARTALIGNPC,
                           // Debug: restart to a not-zero-extended AArch32 PC value
                           Unpredictable_RESTARTZEROUPPERPC,
                           // Zero top 32 bits of X registers in AArch32 state
                           Unpredictable_ZEROUPPER,
                           // Zero top 32 bits of PC on illegal return to AArch32 state
                           Unpredictable_ERETZEROUPPERPC,
                           // Force address to be aligned when interworking branch to A32 state
                           Unpredictable_A32FORCEALIGNPC,
                           // SMC disabled
                           Unpredictable_SMD,
                           // Access Flag Update by HW
                           Unpredictable_AFUPDATE,
                           // Consider SCTLR[].IESB in Debug state
                           Unpredictable_IESBinDebug,
                           // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
                           Unpredictable_BADPMSFCR,
                           // Link/transfer register overlap (load)
                           Unpredictable_LINKTRANSFEROVERLAPLD,
                           // Link/base register overlap (load)
                           Unpredictable_LINKBASEOVERLAPLD,
                           // Clearing DCC/ITR sticky flags when instruction is in flight
                           Unpredictable_CLEARERRITEZERO};
```

5.691 shared/functions/vector/AdvSIMDExpandImm

```
// AdvSIMDExpandImm()
// ==================

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
    case cmode<3:1> of
        when '000'
            imm64 = Replicate(Zeros(24):imm8, 2);
        when '001'
            imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
        when '010'
            imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
        when '011'
            imm64 = Replicate(imm8:Zeros(24), 2);
        when '100'
            imm64 = Replicate(Zeros(8):imm8, 4);
        when '101'
            imm64 = Replicate(imm8:Zeros(8), 4);
        when '110'
            if cmode<0> == '0' then
                imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
            else
                imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
        when '111'
            if cmode<0> == '0' && op == '0' then
                imm64 = Replicate(imm8, 8);
            if cmode<0> == '0' && op == '1' then
                imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
                imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
                imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
                imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
                imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
            if cmode<0> == '1' && op == '0' then
                imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,5):imm8<5:0>:Zeros(19);
                imm64 = Replicate(imm32, 2);
            if cmode<0> == '1' && op == '1' then
                if UsingAArch32() then ReservedEncoding();
                imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>,8):imm8<5:0>:Zeros(48);

    return imm64;
```

5.692 shared/functions/vector/MatMulAdd

```
// MatMulAdd()
// ===========
//
// Signed or unsigned 8-bit integer matrix multiply and add to 32-bit integer matrix
// result[2, 2] = addend[2, 2] + (op1[2, 8] * op2[8, 2])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned, boolean op2_unsigned)
    assert N == 128;

    bits(N)  result;
    bits(32) sum;
    integer  prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 7
                prod = Int(Elem[op1, 8*i + k, 8], op1_unsigned) * Int(Elem[op2, 8*j + k, 8], op2_unsigned);
                sum  = sum + prod;
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

5.693 shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// ================

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1<i> == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

5.694 shared/functions/vector/SatQ

```
// SatQ()
// ======

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

5.695 shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// ============

(bits(N), boolean) SignedSatQ(integer i, integer N)
    if i > 2^(N-1) - 1 then
        result = 2^(N-1) - 1;  saturated = TRUE;
    elsif i < -(2^(N-1)) then
        result = -(2^(N-1));  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

5.696 shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =======================

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1:N-2> == '00' then  // Operands <= 0x3FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x40000000 .. 0xffffffff representing [0.25 .. 1.0)

        // estimate is in the range 256 .. 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipSqrtEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipSqrtEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

5.697 shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =======================

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N IN {16,32};
    if operand<N-1> == '0' then  // Operands <= 0x7FFFFFFF produce 0xFFFFFFFF
        result = Ones(N);
    else
        // input is in the range 0x80000000 .. 0xffffffff representing [0.5 .. 1.0)

        // estimate is in the range 256 to 511 representing [1.0 .. 2.0)
        case N of
            when 16 estimate = RecipEstimate(UInt(operand<15:7>));
            when 32 estimate = RecipEstimate(UInt(operand<31:23>));

        // result is in the range 0x80000000 .. 0xff800000 representing [1.0 .. 2.0)
        result = estimate<8:0> : Zeros(N-9);

    return result;
```

5.698 shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// ==============

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    if i > 2^N - 1 then
        result = 2^N - 1;  saturated = TRUE;
    elsif i < 0 then
        result = 0;  saturated = TRUE;
    else
        result = i;  saturated = FALSE;
    return (result<N-1:0>, saturated);
```

5.699 shared/translation/attrs/CanonicalizeMemoryAttributes

```
// CanonicalizeMemoryAttributes()
// ==============================
// Canoninicalize the memory attributes for Device and Non-cacheable memory types.

MemoryAttributes CanonicalizeMemoryAttributes(MemoryAttributes memattrs)

    if memattrs.memtype == MemType_Device then
        memattrs.inner = MemAttrHints UNKNOWN;
        memattrs.outer = MemAttrHints UNKNOWN;
        memattrs.shareable = TRUE;
        memattrs.outershareable = TRUE;
    else
        memattrs.device = DeviceType UNKNOWN;
        if memattrs.inner.attrs == MemAttr_NC && memattrs.outer.attrs == MemAttr_NC then
            memattrs.shareable = TRUE;
            memattrs.outershareable = TRUE;

    return memattrs;
```

5.700 shared/translation/attrs/CombineS1S2AttrHints

```
// CombineS1S2AttrHints()
// ======================
// Combines cacheability attributes and allocation hints from stage 1 and stage 2

MemAttrHints CombineS1S2AttrHints(MemAttrHints s1desc, MemAttrHints s2desc)

    MemAttrHints result;

    if s2desc.attrs == '01' || s1desc.attrs == '01' then
        result.attrs = bits(2) UNKNOWN;   // Reserved
    elsif s2desc.attrs == MemAttr_NC || s1desc.attrs == MemAttr_NC then
        result.attrs = MemAttr_NC;        // Non-cacheable
    elsif s2desc.attrs == MemAttr_WT || s1desc.attrs == MemAttr_WT then
        result.attrs = MemAttr_WT;        // Write-through
    else
        result.attrs = MemAttr_WB;        // Write-back

    result.hints = s1desc.hints;
    result.transient = s1desc.transient;

    return result;
```

5.701 shared/translation/attrs/CombineS1S2Device

```
// CombineS1S2Device()
// ===================
// Combines device types from stage 1 and stage 2

DeviceType CombineS1S2Device(DeviceType s1device, DeviceType s2device)

    if s2device == DeviceType_nGnRnE || s1device == DeviceType_nGnRnE then
        result = DeviceType_nGnRnE;
    elsif s2device == DeviceType_nGnRE || s1device == DeviceType_nGnRE then
        result = DeviceType_nGnRE;
    elsif s2device == DeviceType_nGRE || s1device == DeviceType_nGRE then
        result = DeviceType_nGRE;
    else
        result = DeviceType_GRE;

    return result;
```

5.702 shared/translation/attrs/CombineS1S2LCSC

```
// CombineS1S2LCSC()
// =================
// Combine attributes protecting capability tag access

```

```
MemoryAttributes CombineS1S2LCSC(MemoryAttributes new_attr, MemoryAttributes s1_attr, MemoryAttributes
    ↪s2_attr)

    new_attr.readtagzero = s1_attr.readtagzero || s2_attr.readtagzero;
    new_attr.readtagfault = s1_attr.readtagfault && !s2_attr.readtagzero;
    new_attr.readtagfaulttgen = s1_attr.readtagfaulttgen;

    new_attr.writetagfault = s1_attr.writetagfault || s2_attr.writetagfault;
    assert !s1_attr.iss2writetagfault;
    new_attr.iss2writetagfault = !s1_attr.writetagfault && s2_attr.iss2writetagfault;

    return new_attr;
```

5.703 shared/translation/attrs/LongConvertAttrsHints

```
// LongConvertAttrsHints()
// =======================
// Convert the long attribute fields for Normal memory as used in the MAIR fields
// to orthogonal attributes and hints

MemAttrHints LongConvertAttrsHints(bits(4) attrfield, AccType acctype)
    assert !IsZero(attrfield);
    MemAttrHints result;
    if S1CacheDisabled(acctype) then             // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        if attrfield<3:2> == '00' then          // Write-through transient
            result.attrs = MemAttr_WT;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        elsif attrfield<3:0> == '0100' then     // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
            result.transient = FALSE;
        elsif attrfield<3:2> == '01' then       // Write-back transient
            result.attrs = MemAttr_WB;
            result.hints = attrfield<1:0>;
            result.transient = TRUE;
        else                                    // Write-through/Write-back non-transient
            result.attrs = attrfield<3:2>;
            result.hints = attrfield<1:0>;
            result.transient = FALSE;

    return result;
```

5.704 shared/translation/attrs/S1CacheDisabled

```
// S1CacheDisabled()
// =================

boolean S1CacheDisabled(AccType acctype)
    enable = if acctype == AccType_IFETCH then SCTLR[].I else SCTLR[].C;
    return enable == '0';
```

5.705 shared/translation/attrs/S2AttrDecode

```
// S2AttrDecode()
// ==============
// Converts the Stage 2 attribute fields into orthogonal attributes and hints

MemoryAttributes S2AttrDecode(bits(2) SH, bits(4) attr, AccType acctype)

    MemoryAttributes memattrs;

    // Device memory
    if attr<3:2> == '00' then
        memattrs.memtype = MemType_Device;
        case attr<1:0> of
            when '00'  memattrs.device = DeviceType_nGnRnE;
            when '01'  memattrs.device = DeviceType_nGnRE;
            when '10'  memattrs.device = DeviceType_nGRE;
            when '11'  memattrs.device = DeviceType_GRE;

    // Normal memory
    elsif attr<1:0> != '00' then
        memattrs.memtype = MemType_Normal;
        memattrs.outer = S2ConvertAttrsHints(attr<3:2>, acctype);
        memattrs.inner = S2ConvertAttrsHints(attr<1:0>, acctype);
        memattrs.shareable = SH<1> == '1';
        memattrs.outershareable = SH == '10';
    else
        memattrs = MemoryAttributes UNKNOWN;    // Reserved

    return CanonicalizeMemoryAttributes(memattrs);
```

5.706 shared/translation/attrs/S2CacheDisabled

```
// S2CacheDisabled()
// =================

boolean S2CacheDisabled(AccType acctype)
    disable = if acctype == AccType_IFETCH then HCR_EL2.ID else HCR_EL2.CD;
    return disable == '1';
```

5.707 shared/translation/attrs/S2ConvertAttrsHints

```
// S2ConvertAttrsHints()
// =====================
// Converts the attribute fields for Normal memory as used in stage 2
// descriptors to orthogonal attributes and hints

MemAttrHints S2ConvertAttrsHints(bits(2) attr, AccType acctype)
    assert !IsZero(attr);

    MemAttrHints result;

    case attr of
        when '01'                               // Non-cacheable (no allocate)
            result.attrs = MemAttr_NC;
            result.hints = MemHint_No;
        when '10'                               // Write-through
            result.attrs = MemAttr_WT;
            result.hints = MemHint_RWA;
        when '11'                               // Write-back
            result.attrs = MemAttr_WB;
            result.hints = MemHint_RWA;

    result.transient = FALSE;

    return result;
```

5.708 shared/translation/attrs/ShortConvertAttrsHints

```
// ShortConvertAttrsHints()
// ========================
// Converts the short attribute fields for Normal memory as used in the TTBR and
// TEX fields to orthogonal attributes and hints

MemAttrHints ShortConvertAttrsHints(bits(2) RGN, AccType acctype, boolean secondstage)

    MemAttrHints result;

    if (!secondstage && S1CacheDisabled(acctype)) || (secondstage && S2CacheDisabled(acctype)) then
       // Force Non-cacheable
        result.attrs = MemAttr_NC;
        result.hints = MemHint_No;
    else
        case RGN of
            when '00'                   // Non-cacheable (no allocate)
                result.attrs = MemAttr_NC;
                result.hints = MemHint_No;
            when '01'                   // Write-back, Read and Write allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RWA;
            when '10'                    // Write-through, Read allocate
                result.attrs = MemAttr_WT;
                result.hints = MemHint_RA;
            when '11'                    // Write-back, Read allocate
                result.attrs = MemAttr_WB;
                result.hints = MemHint_RA;

    result.transient = FALSE;

    return result;
```

5.709 shared/translation/attrs/WalkAttrDecode

```
// WalkAttrDecode()
// ================

MemoryAttributes WalkAttrDecode(bits(2) SH, bits(2) ORGN, bits(2) IRGN, boolean secondstage)

    MemoryAttributes memattrs;

    AccType acctype = AccType_NORMAL;

    memattrs.memtype = MemType_Normal;
    memattrs.inner = ShortConvertAttrsHints(IRGN, acctype, secondstage);
    memattrs.outer = ShortConvertAttrsHints(ORGN, acctype, secondstage);
    memattrs.shareable = SH<1> == '1';
    memattrs.outershareable = SH == '10';

    return CanonicalizeMemoryAttributes(memattrs);
```

5.710 shared/translation/translation/HasS2Translation

```
// HasS2Translation()
// ==================
// Returns TRUE if stage 2 translation is present for the current translation regime

boolean HasS2Translation()
    return (EL2Enabled() && !IsInHost() && PSTATE.EL IN {EL0,EL1});
```

5.711 shared/translation/translation/Have16bitVMID

```
// Have16bitVMID()
// ===============
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return HaveEL(EL2) && boolean IMPLEMENTATION_DEFINED;
```

5.712 shared/translation/translation/PAMax

```
// PAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the physical address
// size for this processor, as log2().

integer PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

5.713 shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====================
// Stage 1 translation regime for the given Exception level

```

```
bits(2) S1TranslationRegime(bits(2) el)
    if el != EL0 then
        return el;
    elsif HaveVirtHostExt() && ELIsInHost(el) then
        return EL2;
    else
        return EL1;

// S1TranslationRegime()
// =====================
// Returns the Exception level controlling the current Stage 1 translation regime. For the most
// part this is unused in code because the system register accessors (SCTLR[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

5.714 shared/translation/translation/VAMax

```
// VAMax()
// =======
// Returns the IMPLEMENTATION DEFINED upper limit on the virtual address
// size for this processor, as log2().

integer VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

Chapter 6
Glossary

Manipulating a capability

An operation manipulates a capability if it changes the rights of that capability by copying the rights to a new capability.

Using a capability

An operation uses a capability if it relies on the permissions granted by that capability.