Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile

Document number	DDI0608
Document version	B.a
Document confidentiality	Non-confidential
Document build information	Printed on: January 27, 2022.

Release information

Date	Version	Changes
2022/Feb/02	EAC	• Initial Armv9.3 EAC release. • Includes FEAT_BRBEv1p1. • Updates to ETE and TRBE.
2021/May/20	EAC	• Initial EAC release. • BRBE, ETE, TME, and TRBE specifications.

Non-Confidential Proprietary Notice

Contents

Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile

Part A   Preface

About this Supplement

Additional reading

Feedback

Part B   Armv9-A Architecture Introduction and Overview

Part C   The Transactional Memory Extension

Contents

Chapter D6   **Trace Unit**

Chapter D10 **Programming**

Chapter D11 **Trace Examples**

Chapter D12 **Examples of basic program trace when exceptions occur**

Chapter D13 **Examples of changes in Context**

Chapter D14 **An example of the use of the trace unit return stack**

Chapter D15   Pseudocode

Chapter D16   Functional Differences from ETMv4

Part E   The Trace Buffer Extension

Part F   The Branch Record Buffer Extension

Part G   Appendixes

Part H   Glossary

Chapter H1   **Glossary**

Part A
Preface

About this Supplement

This Supplement is the *Arm ® Architecture Reference Manual Supplement, Armv9-A, for Armv9-A architecture profile*. This book describes the changes and additions to the Armv8-A architecture that are introduced by the Armv9-A architecture extensions, and therefore must be read with the *Arm ® Architecture Reference Manual, for A-profile architecture*.

This manual is organized into parts:

- Part B

  Introduces the *Arm ® Architecture Reference Manual Supplement, Armv9-A, for Armv9-A architecture profile*.

- Part C

  Describes the Transactional Memory Extension (TME).

- Part D

  Describes the Embedded Trace Extensions (ETE).

- Part E

  Describes the Trace Buffer Extension (TRBE).

- Part F

  Describes the Branch Record Buffer Extension (BRBE).

- Part G

  Provides additional information.

  Chapter G1 provides system registers synchronization requirements.

  Chapter G2 provides stages of execution information for FEAT_TRBE.

  Chapter G3 provides software usage information for FEAT_TRBE.

  Chapter G4 provides *Transactional Memory Extension* (TME) litmus tests.

  Chapter G5 provides TME Transactional Lock Elision

  Chapter G6 provides TME implementation recommendations.

- Part H

  Glossary that defines terms used in this document that have a specialized meaning.

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>, #{, }"

> > 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 *content items*. A content item is classified as one of the following:

- Declaration
- Rule
- Goal
- Information
- Rationale
- Implementation note
- Software usage

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

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

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

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

Content item identifiers

A content item may have an associated identifier which is unique among content items in this specification.

After this specification reaches beta status, a given content item has the same identifier across subsequent versions of the specification.

Content item rendering

In this document, a content item is rendered with a token of the following format in the left margin: L_{iiiii}

- L is a label that indicates the content class of the content item.
- *iiiii* is the identifier of the content item.

Content item classes

Declaration

A Declaration is a statement that either

- introduces a concept, or
- introduces a term, or
- describes the structure of data, or
- describes the encoding of data.

A Declaration does not describe behaviour.

A Declaration is rendered with the label *D*.

Rule

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

A Rule explains what happens in a particular situation.

A Rule does not define concepts or terminology.

A Rule is rendered with the label *R*.

Goal

A Goal is a statement about the purpose of a set of rules.

A Goal explains why a particular feature has been included in the specification.

A Goal is comparable to a "business requirement" or an "emergent property."

A Goal is intended to be upheld by the logical conjunction of a set of rules.

A Goal is rendered with the label *G*.

Information

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

An Information statement is rendered with the label *I*.

Rationale

A Rationale statement explains why the specification was specified in the way it was.

A Rationale statement is rendered with the label *X*.

Implementation note

An Implementation note provides guidance on implementation of the specification.

An Implementation note is rendered with the label *U*.

Software usage

A Software usage statement provides guidance on how software can make use of the features defined by the specification.

A Software usage statement is rendered with the label *S*.

Additional reading

This section lists publications by Arm and by third parties.

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

[1] *Arm® Architecture Reference Manual, for A-profile architecture*. (ARM DDI 0487).

[2] *Arm® Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A*. (ARM DDI 615).

[3] *Arm® Embedded Trace Macrocell Architecture Specification ETMv4*. (ARM IHI 0064).

[4] *AMBA ATB Protocol Specification*. (ARM IHI 0032).

[5] *ARM CoreSight Architecture Specification*. (ARM IHI 0029).

[6] *Arm® Architecture Reference Manual Supplement; Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture*. (ARM DDI 0598).

This supplement should also be read with the following System register and ISA descriptions:

- *System Register XML for Armv9-A*.
- *A64 ISA XML for Armv9-A*.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this supplement

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

- The title, Arm® Architecture Reference Manual Supplement Armv9, for Armv9-A architecture profile.
- The number, DDI0608 B.a.
- 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.

Part B
Armv9-A Architecture Introduction and Overview

Chapter B1
Introduction to the Armv9-A Architecture

B1.1 Architectural extensions added by Armv9-A

An Armv9 compliant implementation must also be Armv8.5 compliant.

An Armv9.1 compliant implementation must also be Armv8.6 compliant.

An Armv9.2 compliant implementation must also be Armv8.7 compliant.

An Armv9.3 compliant implementation must also be Armv8.8 compliant.

The AArch32 Execution state might optionally be implemented at EL0. The AArch32 Execution state is not implemented at EL1, EL2, or EL3.

An implementation of the Armv9-A architecture must include all of the extensions that this section describes as mandatory. Such an implementation is also called an implementation of the Armv9-A architecture.

The implementation of FEAT_DoubleLock in an Armv9 implementation is prohibited.

An implementation of the Armv9-A architecture cannot include an Embedded Trace Macrocell (ETM).

The Armv9-A architecture extension adds the following architectural features, which are identified by the architectural feature name and a short description of the feature:

B1.1.1 FEAT_ETE, Embedded Trace Extension

FEAT_ETE provides a trace unit that records details about software control flow running on a *Processing Element* (PE), which can be used to aid debugging or optimizing. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.

This feature is OPTIONAL in an Armv9.0 implementation.

This feature requires FEAT_TRBE and FEAT_TRF.

This feature is supported in both AArch64 and AArch32 states.

The ID_AA64DFR0_EL1.TraceVer field identifies the presence of FEAT_ETE.

B1.1.2 FEAT_SVE2, Scalable Vector Extension version 2

The Scalable Vector Extension version 2 (SVE2) is a superset of SVE that incorporates functionality similar to Advanced SIMD, and other enhancements.

FEAT_SVE2 is OPTIONAL.

This feature is supported in AArch64 state only.

FEAT_SVE2 requires FEAT_SVE.

The following fields indicate the presence of FEAT_SVE2:

- ID_AA64PFR0_EL1.SVE.
- ID_AA64ZFR0_EL1.SVEver.

Although FEAT_SVE2 is OPTIONAL, standard Armv9-A software platforms support FEAT_SVE2.

For more information, see *The Scalable Vector Extension (SVE)* section in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

B1.1.3 FEAT_TME, Transactional Memory Extension

FEAT_TME adds the TCANCEL, TCOMMIT, TSTART, and TTEST instructions. These instructions support hardware transactional memory, which means a group of instructions can appear to be collectively executed as a single atomic operation.

FEAT_TME is OPTIONAL.

This feature is supported in AArch64 state only.

The ID_AA64ISAR0_EL1.TME field identifies the presence of FEAT_TME.

B1.1.4 FEAT_TRBE, Trace Buffer Extension

FEAT_TRBE enables support for a Trace Buffer Unit within a processing element (PE). When the Trace Buffer Unit is enabled, Program Flow Trace generated by a trace unit is written directly to memory by the Trace Buffer Unit, rather than routing trace data to a trace sink.

This feature is OPTIONAL in an Armv9.0 implementation.

This feature requires FEAT_ETE and FEAT_TRF.

This feature is supported in both AArch64 and AArch32 states.

The ID_AA64DFR0_EL1.TraceBuffer field identifies the presence of FEAT_TRBE.

B1.1.5 FEAT_ETEv1p1, Embedded Trace Extension

FEAT_ETEv1p1 extends FEAT_ETE to support FEAT_FGT.

This feature is OPTIONAL in an Armv9.1 implementation.

This feature requires FEAT_ETE, FEAT_TRBE, FEAT_FGT, and FEAT_TRF.

The TRCDEVARCH.REVISION field identifies the presence of FEAT_ETEv1p1.

For more information on FEAT_FGT, see *Arm® Architecture Reference Manual, for A-profile architecture* [1].

For more information on FEAT_ETEv1p1, see Chapter D1 *Embedded Trace Extension*.

B1.1.6 FEAT_BRBE, Branch Record Buffer Extension

FEAT_BRBE provides a Branch record buffer for capturing control path history.

This feature is OPTIONAL in an Armv9.2 implementation.

This feature is supported in AArch64 only.

The ID_AA64DFR0_EL1.BRBE field identifies the presence of FEAT_BRBE.

B1.1.7 FEAT_ETEv1p2, Embedded Trace Extension

FEAT_ETEv1p2 extends FEAT_ETE to support FEAT_RME.

This feature is OPTIONAL in an Armv9.2 implementation.

This feature requires FEAT_RME and FEAT_ETEv1p1.

The TRCDEVARCH.REVISION field identifies the presence of FEAT_ETEv1p2.

For more information on FEAT_RME and FEAT_ETEv1p2, see *Arm® Architecture Reference Manual Supplement, The Realm Management Extension (RME), for Armv9-A* [2].

B1.1.8 FEAT_BRBEv1p1, Branch Record Buffer Extension

FEAT_BRBEv1p1 FEAT_BRBEv1p1 extends FEAT_BRBE to enable branch recording at EL3.

This feature is OPTIONAL in an Armv9.3 implementation.

This feature requires FEAT_BRBE.

This feature is supported in AArch64 only.

The ID_AA64DFR0_EL1.BRBE field identifies the presence of FEAT_BRBEv1p1.

For more information on FEAT_BRBE, see Chapter F1 *Branch Record Buffer Extension*.

Part C
The Transactional Memory Extension

Chapter C1
Transactional Memory Extension

C1.1 Transactions

R_{TJXB} A *transaction* is a group of instructions executing in *Transactional state*.

R_{YQLB} Instructions outside a transaction execute in *Non-transactional state*.

C1.1.1 Transactional state

C1.1.1.1 Entering transactional state: starting a transaction (`TSTART`)

R_{ZYKL} When a `TSTART` instruction is committed for execution in Non-transactional state, it starts an *outer transaction*.

R_{BMPK} When starting an outer transaction, the *Processing Element* (PE) enters Transactional state.

R_{ZFWF} When a `TSTART` instruction is committed for execution in Transactional state it starts a transaction *nested* within the pre-existing transaction, or simply a *nested transaction*.

R_{DCDQ} The *transactional nesting depth* indicates the degree of nesting of a transaction.

R_{DNMF} The architecture requires the maximum transactional nesting depth to be 255.

R_{RRJX} In Non-transactional state, the transactional nesting depth is 0.

R_{XKHY} When starting a transaction, the transactional nesting depth is incremented by 1.

I_{WTLG} In the rest of the document, unless explicitly prefixed with *outer* or *nested*, the term *transaction* will refer to an outer transaction and all the nested transactions contained within.

C1.1.1.2 Exiting transactional state by committing a transaction (`TCOMMIT`)

R$_{HXMY}$ A transaction *commits* when a `TCOMMIT` instruction is committed for execution in Transactional state.

R$_{DDFV}$ Transactional state is exited when committing an outer transaction.

R$_{WYQK}$ When committing a transaction, the transactional nesting depth is decremented by 1.

C1.1.1.3 Exiting transactional state by cancelling (`TCANCEL`) or failing a transaction

R$_{LJYW}$ A transaction *is canceled* when a `TCANCEL` instruction is committed for execution in Transactional state.

R$_{CSXK}$ A transaction *fails* when the PE exits transactional state for any reason other than the execution of a `TCOMMIT` instruction or the execution of a `TCANCEL` instruction.

R$_{HVFD}$ When a transaction fails or is canceled, Transactional state is exited, and execution continues at the instruction that follows the `TSTART` instruction of the outer transaction.

R$_{MNVC}$ The result of the `TSTART` instruction of the outer transaction encodes the cause of the failure (see Section C1.2.1 *Failure causes*).

R$_{VRLY}$ When a transaction fails or is canceled, the transactional nesting depth is set to 0.

C1.1.2 Transactional reservation granule, read and write sets

R$_{XCXC}$ The *transactional reservation granule* is defined as a contiguous memory block of size 2^a bytes, formed by ignoring the least significant bits of a memory access.

R$_{DVWQ}$ The size of the memory block is IMPLEMENTATION DEFINED in the range 4 – 512 words.

R$_{NYST}$ The Exclusive Reservation Granule CTR_EL0.ERG identifies the transactional reservation granule.

Below the notions of Location and read or write memory effects are as described in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Basic definitions*.

C1.1.2.1 Transactional read set

R$_{RRZY}$ The *transactional read set* of a transaction is defined to be the set of transactional reservation granules containing all Locations accessed by memory reads inside the transaction.

R$_{KJCC}$ The reads in the transactional read set are referred to as *transactional reads*.

C1.1.2.2 Transactional write set

R$_{XJNK}$ The *transactional write set* of a transaction is defined to be the set of transactional reservation granules containing all Locations accessed by memory writes inside the transaction.

R$_{HWVK}$ The writes in the transactional write set are referred to as *transactional writes*.

R$_{BLGB}$ Limits to the transactional read set size and the transactional write set size are IMPLEMENTATION DEFINED.

C1.2 Transaction failure

C1.2.1 Failure causes

R_{BWYQ} When a transaction fails or is canceled, the destination register of the TSTART instruction of the outer transaction encodes the cause of the failure as follows.

R_{SJJT} For causes that are due to direct or attempted execution of an instruction, only the cause generated by the instruction that appears first in program order is reported.

R_{PTGZ} For causes that are not due to direct or attempted execution of an instruction, any number of causes may be reported.

R_{YXLQ} When more than one cause is reported, then RTRY is set to the logical AND of the prescribed or expected RTRY value of each identified failure cause.

R_{YSWY} **RTRY, bit [15]**

When this bit is set it signifies that the transaction may commit on retry.

When this bit is clear the software should assume that the transaction will not commit on retry.

RTRY is not a failure cause.

R_{YDHF} **REASON, bits [14:0]**

This field holds the 15 low order bits of the TCANCEL operand value when CNCL is 1 else this field is 0.

Bits [63:25]

Reserved, RES0.

R_{TYHM} **TRIVIAL, bit [24]**

When this bit is set it signifies that the system is currently running the trivial implementation enabled by the bits described in C3.1.5 *ISS encoding for an exception from a TSTART instruction*.

The prescribed RTRY value is 0.

R_{BXPB} **INT, bit [23]**

When IMP=1, this bit indicates whether or not an unmasked interrupt was delivered in transactional state but not subsequently taken in non-transactional state due to being masked by the PE. See Section C1.7 *Interrupt masking* for more information.

The prescribed RTRY value is 0.

R_{TTKV} **DBG, bit [22]**

When this bit is set it signifies that a debug-related exception was encountered but not raised.

The prescribed RTRY value is 0.

R_{STJB} **NEST, bit [21]**

When this bit is set it signifies that the maximum transactional nesting depth was exceeded.

The prescribed RTRY value is 0.

R_{RVBK} **SIZE, bit [20]**

When this bit is set it signifies that the transaction failed because the transactional read set limit or the transactional write set limit was exceeded.

The prescribed RTRY value is 0.

R_{SZFF}   **ERR, bit [19]**

When this bit is set it signifies that an operation was attempted which is not architecturally permitted in Transactional state. This includes but is not limited to attempting to raise a synchronous exception, attempting to execute an instruction not permitted in Transactional state, or attempting to change Exception level.

The prescribed RTRY value is 0.

R_{XZDB}   **IMP, bit [18]**

When this bit is set it signifies a failure cause that does not fall under any of the other cases.

The expected RTRY value is 1 if the transaction may commit on retrying and 0 otherwise.

RTRY must not systematically be set to 1 with IMP cause. This is because it could prevent the forward progress in finite time of at least one the threads that is accessing a location within the transactional read or write sets.

R_{RMJK}   **MEM, bit [17]**

When this bit is set it signifies that the transaction failed because a transactional memory conflict was detected.

The expected RTRY value is 1.

R_{KWHH}   **CNCL, bit [16]**

When this bit is set it signifies that the transaction was canceled by a `TCANCEL` instruction.

The RTRY value is the most significant bit of the `TCANCEL` immediate operand.

C1.2.2   Transaction checkpoint

R_{PJGV}   The *transaction checkpoint* defines the following subset of the AArch64 state:

- Registers in AArch64 execution state: R0-R30, SP, ICC_PMR_EL1.

- AArch64 process state: NZCV, DAIF.

- If both floating-point and SVE are enabled: Z0-Z31, P0-P15, FFR, FPCR, FPSR.

- If floating-point is enabled and SVE is disabled or trapped: V0-V31, FPCR, FPSR.

R_{GKRW}   It is IMPLEMENTATION DEFINED if any of the System registers encoded with op0==0b11 and CRn==0b1x11 are included in the transaction checkpoint.

R_{QJYL}   No other System registers are included in the transaction checkpoint.

R_{BFYL}   When a transaction fails or is canceled, the subset of the AArch64 state defined by the transaction checkpoint is reverted to a state that is consistent with the PE having executed all of the instructions up to but not including the point in the instruction stream where Transactional state was entered, and none afterwards, with the following exceptions:

- The destination register of the `TSTART` instruction of the outer transaction is updated to encode the transaction failure cause.

- When executing at an Exception level that is constrained to use a vector length that is less than the maximum implemented vector length, the bits beyond the constrained length of Z0-Z31, P0-P15, and FFR are restored to a value of either zero or the value they had when Transactional state was entered. The choice between these options is IMPLEMENTATION DEFINED and can vary dynamically.

R_{FFQR}   Writes by a failed or canceled transaction do not generate write Memory effects. For the definition of Memory effects, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Basic definitions*.

R_{KBBS} If SVE is disabled or trapped, the current vector length is considered to be constrained to 128 bits (see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *SVE Configurable vector length*).

R_{VQGT} SPSel cannot be modified in Transactional state. For more information, see Section C1.8 *A64 instruction behavior in Transactional state*.

C1.3 Memory model

Transactional Memory Extension (TME) proposes the following additions to the memory ordering and observability rules described in the *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Definition of the Armv8 memory model*.

C1.3.1 External visibility

Adding the following definitions:

R_{TCXC} **Locally-ordered-before**

A read or a write RW1 is *Locally-ordered-before* a read or a write RW2 from the same Observer if and only if any of the following cases apply:

- RW1 is Dependency-ordered-before RW2.

- RW1 is Atomic-ordered-before RW2.

- RW1 is Barrier-ordered-before RW2.

- RW1 is Locally-ordered-before a read or a write that is Locally-ordered-before RW2.

R_{KDFQ} **Transactionally-observed-by**

A read or a write RW1 from an Observer is *Transactionally-observed-by* a read or a write RW2 from a different Observer if and only if any of the following cases apply:

- There is a read or a write RW3 in the same transaction as RW1, and RW3 is Observed-by RW2.

- There is a read or a write RW3 in the same transaction as RW2, and RW1 is Observed-by RW3.

Changing the definition of Barrier-ordered-before to the following:

R_{RBRW} **Barrier-ordered-before**

Barrier instructions order prior Memory effects before subsequent Memory effects generated by the same Observer. A read or a write RW1 is *Barrier-ordered-before* a read or a write RW2 from the same Observer if and only if RW1 appears in program order before RW2 and any of the following cases apply:

- RW1 appears in program order before a `DMB FULL` that appears in program order before RW2.

- RW1 is a write W1 generated by an instruction with Release semantics and RW2 is a read R2 generated by an instruction with Acquire semantics.

- RW1 is generated by an instruction with Acquire semantics.

- RW2 is generated by an instruction with Release semantics.

- RW1 is a read R1 appearing in program order before a `DMB LD` that appears in program order before RW2.

- RW2 is a write W2 and either:

    - RW1 is a write W1 appearing in program order before a `DMB ST` that appears in program order before W2.

    - RW1 appears in program order before a write W3 generated by an instruction with Release semantics and W2 is Coherence-after W3.

- RW1 and RW2 are not in the same transaction, and at least one of RW1 or RW2 is in the read or write set of a committed transaction.

- RW1 appears in program order before a committed transaction that appears in program order before RW2.

Changing the definition of Ordered-before to the following:

R_{BXFQ} **Ordered-before**

An arbitrary pair of Memory effects is ordered if it can be linked by a chain of ordered accesses consistent with external observation. A read or a write RW1 is Ordered-before a read or a write RW2 if and only if any of the following cases apply:

* RW1 is Observed-by RW2.

* RW1 is Transactionally-observed-by RW2.

* RW1 is Locally-ordered-before RW2.

* RW1 is Ordered-before a read or a write that is Ordered-before RW2.

I_{FMHN} Conflicts are a natural consequence of the pre-existing External visibility requirement. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Ordering constraint*. A cycle in Ordered-before that involves a Transactionally-observed-by relation indicates a conflict.

R_{ZRBP} A transaction is said to be conflicting if and only if committing the transaction would violate the external visibility requirement, in which case the transaction fails with MEM cause.

S_{JCSK} In the event of repeated transactional conflicts the architecture does not guarantee forward progress for any transactions involved, and the software must take appropriate measures for example by setting a threshold after which the software takes a specific fallback path.

C1.3.2 Atomicity

I_{LWMY} This section documents the behavior of the A64 Load-Exclusive and Store-Exclusive instructions, and all A64 atomic instructions (CAS, CASP, LD<OP>, and SWP) in Transactional state.

R_{LJDF} Transactional writes generated as side-effects from the above instructions follow the ordering and observability rules described in the previous section.

R_{PNXP} A transactional store to an address marked for exclusive access in the global monitor for any other PE:

* Clears the marking if the transaction commits.

* May clear the marking if the transaction fails or is canceled.

R_{JFLH} When entering Transactional state or exiting Transactional state by committing, canceling or failing a transaction:

* The local monitor state of the executing PE transitions to the Open access state.

* The final state of the global monitor state machine for the executing PE is IMPLEMENTATION DEFINED.

* The global monitor state machine for any other PE is not affected.

R_{NBWK} If the global monitor state for a PE changes from Exclusive access to Open access because of entering or exiting Transactional state, an event is generated and held in the Event register for that PE.

S_{HBSY} Inserting any of the A64 atomic primitive instructions inside a transaction does not provide any extra functionality to software. Sharing code among the transaction and its fallback path may lead to such instructions being executed in transactional state.

C1.4 Transactions and memory attributes

I_{CBHX} Some system implementations might not support transactional accesses for all regions of the memory. This can apply to:

- Any type of memory in the system that does not support hardware cache coherency.

- Device memory, Non-cacheable memory, or memory that is treated as Non-cacheable, in an implementation that does support hardware cache coherency.

R_{FVGF} In such implementations, it is defined by the system which address ranges or memory types support transactional accesses.

R_{NZRZ} The memory types for which it is architecturally guaranteed that transactional accesses are supported are:

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

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

R_{LMVV} If transactional accesses are not supported for an address range or memory type, then performing a transactional load or a transactional store to such a location fails the transaction with IMP cause.

I_{JXXM} Memory accesses generated by different instructions inside a transaction can have different shareability attributes.

R_{LSXT} When accesses to any two Locations generated by the same instruction inside a transaction have different shareability attributes then the results are CONSTRAINED UNPREDICTABLE. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Memory access restrictions*.

R_{DCWP} Accesses, including transactional accesses, by multiple PEs to a Location with mismatched attributes leads to CONSTRAINED UNPREDICTABLE behavior. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Mismatched memory attributes*.

C1.5 Address translation

C1.5.1 Transactional translation table walks

R_{YCTD} Transactional memory accesses to a given address are permitted to perform translation table walks, except when the transactional memory access originates from EL0 and either of the following cases holds:

- The address is translated using TTBR0_EL1, and TCR_EL1.NFD0==1.

- The address is translated using TTBR1_EL1, and TCR_EL1.NFD1==1.

- The address is translated using TTBR0_EL2, and TCR_EL2.NFD0==1.

- The address is translated using TTBR1_EL2, and TCR_EL2.NFD1==1.

R_{MZBV} A transactional memory access that is not permitted to perform a translation table walk and would otherwise generate an exception in Non-transactional state fails the transaction with ERR cause without generating an exception.

X_{WRWD} This scheme addresses timing attacks on Kernel Address Space Layout Randomization. If TCR_EL1.NFD1 is set, an EL0 transaction that attempts to probe the kernel address space will always fail with the same timing and the same failure cause because either there is a TLB miss and the transaction fails with ERR cause, or there is a TLB hit and a suppressed MMU permission fault (assuming TTBR1_EL1 address range is protecting itself from EL0 accesses) fails the transaction with ERR cause. This way the malicious software should not be able to distinguish between the two cases.

C1.5.2 Hardware management of the Access flag and dirty state

R_{ZNNY} TME requires that the implementation supports hardware management of the Access flag and dirty state. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Hardware management of the Access flag and dirty state*.

R_{KGPQ} Transactional memory accesses follow the rules for updating the Access flag and dirty state as described in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Hardware management of the Access flag and dirty state* and *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Ordering of hardware updates to the translation tables*.

R_{NCHW} When hardware updating of the Access flag is enabled, updates to the stage 1 and stage 2 Access flag generated by memory accesses in Transactional state may become observable even if the transaction fails or is canceled.

R_{SSNZ} When hardware updating of the dirty state is enabled, updates to the stage 1 and stage 2 dirty state generated by memory accesses in Transactional state may become observable even if the transaction fails or is canceled.

I_{LZDF} Arm requires hardware management of the Access flag and dirty state for performance reasons.

S_{JKDY} Software management of the Access flag would mean that when a page is accessed for the first time inside a transaction, the transaction fails and is re-executed in Non-transactional state.

I_{TMBP} Arm requires allowing transactional dirty state updates to become observable even if the responsible transaction fails or is canceled for performance reasons. Otherwise, every time a page is written for the first time inside a transaction, then either the transaction fails which is bad for performance, or the hardware must manage the dirty state updates until the PE exits Transactional state which increases implementation complexity.

C1.5.3 TLB shoot-down

R_{XVMC} A TLBI by another PE that applies to a Location in the transactional read set or the transactional write set of the currently executing transaction causes that transaction to fail.

I_{YJQH} In order to provide this functionality, an implementation needs to either track the Virtual to Physical Address mappings for the Locations in the transactional read or write sets of the transaction that is currently executing, or fail the transaction on any invalidation by another PE. In the former case, if a transaction exceeds the IMPLEMENTATION DEFINED tracking limit of Virtual to Physical Address mappings, then the transaction fails.

I_{RFLJ} For performance reasons, Arm recommends that the implementation does not fail the transaction if the ASID and VMID of an invalidation by another PE, does not match the one of the currently executing transaction.

C1.5.4 Translation table modifications inside transactions

I_{KXRF} The required break-before-make sequence described in the *Arm® Architecture Reference Manual, for A-profile architecture* [1] *General TLB maintenance requirements* for updating translation table entries cannot be executed inside a transaction, since the required TLBI and DSB instructions lead to transaction failure (see Table C1.4 and Table C1.6).

C1.6 Modification of instructions in Transactional state

I_{ZMVM} The Arm architecture does not require the hardware to ensure coherency between instruction caches and memory, even for shared memory locations. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Implication of caches for the application programmer*.

R_{RLTS} TME follows the rules for concurrent modification and execution of instructions as explained in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Concurrent modification and execution of instructions*.

R_{CLBS} TME does not guarantee that a transactional thread of execution T is isolated from a non-transactional thread of execution making modifications to the instruction stream of T.

     See also Table C1.6 for the behavior of ISB and DSB instructions in Transactional state.

I_{CVKD} This implies that a transactional thread of execution cannot modify its own instruction stream, or other instruction streams using the mechanism suggested in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Concurrent modification and execution of instructions*, since transactional writes are not observable until a transaction commits and the DSB instruction required for synchronization of the modifications fails the transaction.

C1.7 Interrupt masking

R_{TTSQ} In Transactional state, interrupts are pended, and unmasked interrupts are taken when Transactional state is exited.

R_{CHZK} In the absence of a specific requirement to take an interrupt, it is IMPLEMENTATION DEFINED if the delivery of an unmasked interrupt fails the transaction, but the architecture requires that the interrupt is taken in finite time. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Prioritization and recognition of interrupts*.

R_{BFMS} If the delivery of an unmasked interrupt fails the transaction, the failure cause reported is IMP.

I_{LHSC} Transactional code with sufficient privileges can change the value of DAIF or ICC_PMR_EL1 to mask or unmask interrupts.

R_{NXXN} A transaction fails with IMP cause and INT set if both of the following happen:

- an unmasked interrupt delivered to a PE leads to the currently executing transaction on the PE to fail, and
- upon restoring DAIF and ICC_PMR_EL1 the interrupt becomes masked again and will not be taken.

X_{MLVZ} If the transaction fails or is canceled the DAIF and ICC_PMR_EL1 registers are restored to the values they held before entering Transactional state. This action will affect the masking or unmasking of interrupts before the first non-transactional instruction executes. If the implementation decides to fail the transaction when the interrupt is delivered, then after the values of DAIF and ICC_PMR_EL1 are restored to their pre-transactional state, the interrupt will be masked and will not be taken. But if the transaction restarts then, as soon as interrupts are transactionally re-enabled, the transaction will fail because there is a pending interrupt. To avoid a livelock this is reported as a non-restartable failure. For more information, see Section C1.2.2 *Transaction checkpoint*.

C1.8 A64 instruction behavior in Transactional state

R_{NHND} Transactional state changes the execution of some A64 instructions.

This section includes the affected instructions and their expected behavior in Transactional state.

R_{QHYS} Any instruction not included in this section behaves the same in Transactional state as in Non-transactional state.

R_{SHQC} • Exception level changes cannot occur. Executing an instruction that would otherwise generate an Exception level change fails the transaction with ERR cause as described in this document.

R_{LXPV} • Synchronous exceptions are suppressed and fail the transaction with ERR cause. See Sections C2.1.1 *Breakpoint Instruction exceptions*, C2.1.2 *Breakpoint exceptions*, and C2.1.3 *Watchpoint exceptions* for details.

Table C1.1: Exception generating instructions

Mnemonic	Instruction	Behavior
BRK	Breakpoint Instruction	See Section C2.1.1 *Breakpoint Instruction exceptions*
HLT	Halt Instruction	See Section C2.2.2 *Halting Instruction debug event*
HVC	Generate exception targeting EL2	Transaction fails with ERR cause
SMC	Generate exception targeting EL3	Transaction fails with ERR cause
SVC	Generate exception targeting EL1	Transaction fails with ERR cause

Table C1.2: Exception return instructions

Mnemonic	Instruction	Behavior
ERET	Exception return using current ELR and SPSR	Transaction fails with ERR cause
ERETAA, ERETAB	Exception return with pointer authentication	Transaction fails with ERR cause

Table C1.3: System register instructions

Mnemonic	Instruction	Behavior
MRS	Move System register to general-purpose register	See Section C1.8.1 *MRS*
MSR (register)	Move general-purpose register to System register	See Section C1.8.2 *MSR (register)*
MSR (immediate)	Move immediate to PSTATE field	See Section C1.8.3 *MSR (immediate)*

Table C1.4: System instructions

Mnemonic	Instruction	Behavior
SYS	System instruction	See Section C1.8.4 *SYS and SYSL*

Mnemonic	Instruction	Behavior
SYSL	System instruction with result	See Section C1.8.4 *SYS and SYSL*
IC	Instruction cache maintenance	Transaction fails with ERR cause
DC except DC ZVA	Data cache maintenance	Transaction fails with ERR cause
DC ZVA	Data cache zero	Same as in Non-transactional state
AT	Address translation	Transaction fails with ERR cause
TLBI	TLB Invalidate	Transaction fails with ERR cause

Table C1.5: Hint instructions

Mnemonic	Instruction	Behavior
NOP	No operation	Same as in Non-transactional state
YIELD	Yield hint	Same as in Non-transactional state
WFE	Wait for event	See Section C1.8.5 *Wait for Event*
WFI	Wait for interrupt	Transaction fails with ERR cause
SEV	Send event	Same as in Non-transactional state
SEVL	Send event local	Same as in Non-transactional state
HINT	Unallocated hint	Same as in Non-transactional state

Table C1.6: Barrier and CLREX instructions

Mnemonic	Instruction	Behavior
CLREX	Clear exclusive monitor	Same as in Non-transactional state
DSB	Data synchronization barrier	Transaction fails with ERR cause
DMB	Data memory barrier	See C1.8.6 *DMB*
ESB	Error synchronization barrier	Transaction fails with ERR cause
ISB	Instruction synchronization barrier	See C1.8.7 *ISB*
PSB CSYNC	Profiling synchronization barrier	Same as in Non-transactional state
TSB CSYNC	Trace synchronization barrier	Same as in Non-transactional state

C1.8.1 MRS

R_{KTMD} Registers encoded with op0==0b10 are not accessible at any Exception level.

R_{ZTYF} Registers encoded with op0==0b11 and CRn==12, except ICC_HPPIR0_EL1, ICC_HPPIR1_EL1, ICC_RPR_EL1, are not accessible at any Exception level.

R_{VJST} If enhanced nested virtualization is enabled and the read of a permitted System register is transformed to a read from memory, then the generated read is considered transactional.

R_{MCFP} Attempting to read a register that is not accessible at the current Exception level fails the transaction with ERR cause without trapping.

R_{KPDH} If a read from memory generates any exception, the exception is suppressed and the transaction fails with ERR cause without trapping.

C1.8.2 `MSR` (register)

R_{CKFL} Registers FPCR, FPSR, NZCV, DAIF, ICC_PMR_EL1, and PMSWINC_EL0 are accessible at the same Exception levels as in Non-transactional state.

R_{STQJ} All other registers are not accessible at any Exception level.

R_{XZGW} Attempting to write a register that is not accessible at the current Exception level fails the transaction with ERR cause without trapping.

C1.8.3 `MSR` (immediate)

R_{MNFL} Only the instruction forms that select the MSR DAIFSet and MSR DAIFClr instructions are defined.

R_{SGNZ} All other encodings are reserved, and the corresponding instructions are UNDEFINED.

R_{DNMG} Attempting to execute an UNDEFINED instruction fails the transaction with ERR cause without trapping.

C1.8.4 `SYS` and `SYSL`

R_{LKNM} The accessibility of the instructions encoded with op0=0b01 and CRn=0b1x11 is IMPLEMENTATION DEFINED.

R_{GKLF} Attempting to execute an undefined instruction fails the transaction with ERR cause without trapping.

C1.8.5 Wait for Event

R_{BXPX} If executing a WFE instruction in Non-transactional state would trap to a higher Exception level, then the transaction fails with ERR cause without trapping.

Otherwise, the WFE instruction behaves the same as in Non-transactional state.

R_{GPGL} A transaction that has entered low-power state due to the execution of a WFE instruction is called a *waiting transaction*.

R_{FCPL} A PE that enters a low-power state continues to track and respond to transactional conflicts with memory accesses from other PEs.

R_{GFJL} It is IMPLEMENTATION DEFINED whether a waiting transaction that receives a *WFE wake-up event* resumes execution without failing. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Wait for Event mechanism and Send event*.

R_{KYSZ} A waiting transaction is permitted to fail for any IMPLEMENTATION DEFINED reason before a wake-up event is received.

I_{PMZF} Arm recommends that a waiting transaction fails on a transactional conflict with another PE, for performance reasons.

I_{FFTN} Arm recommends that waiting transactions do not fail upon receiving a wake-up event that is not an interrupt that must be taken, for performance reasons.

The following is a non-exhaustive list of wake-up events that could safely resume a transaction:

R_{ZVBJ} • The execution of an SEV instruction on any other PE in the multiprocessor system.

R_{ZRNN} • An event sent by the timer event stream for the PE. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Event streams*.

R_{FDMK} • An event caused by the clearing of the global monitor for the PE.

R_{LPBD} • A masked interrupt.

C1.8.6 DMB

R_{SHWX} Transactional accesses to Device or Normal Non-cacheable memory that appear before the DMB in program order are merged with transactional accesses to Device or Normal Non-cacheable memory of the same type (read or write) to the same Location that appear in program order after the DMB, if they are executed in the same transaction.

R_{XMCB} If transactional accesses, executing in the same transaction containing the DMB, access the same memory-mapped peripheral of arbitrary system-defined size, then it is not guaranteed that accesses in program order before the DMB that are accessing Device or Normal Non-cacheable memory will arrive at the peripheral before accesses in program order after the DMB that are accessing Device or Normal Non-cacheable memory.

C1.8.7 ISB

R_{NLMZ} Executing an ISB instruction in Transactional state is a *Context synchronization event*, with the same effects of a Context synchronization event in Non-transactional state except that unmasked interrupts that are pending at the time of the Context synchronization event are not required to be taken.

R_{YHLW} If halting is allowed, any Halting debug event that is pending before the ISB instruction is executed fails the transaction with DBG cause.

R_{JVGJ} It is IMPLEMENTATION DEFINED whether or not the transaction fails if there are pending unmasked interrupts when the ISB instruction is executed.

I_{NMWC} If the first instruction after exiting Transactional state generates a synchronous exception, then the architecture does not define whether the PE takes the interrupt or the synchronous exception first.

See also Section C1.7 *Interrupt masking*.

C1.8.8 First-fault and Non-fault load instructions

I_{HMFW} SVE provides a First-fault option for some SVE vector load instructions. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Glossary*.

R_{NMXZ} In Transactional state, SVE's First-fault option causes memory access faults to be suppressed without causing the transaction to fail if they do not occur as a result of the First active element of the vector.

Instead, the FFR is updated to indicate which of the active vector elements were not successfully loaded (see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *FFR, First Fault Register*).

I_{GHCR} SVE provides a Non-fault option for some SVE vector load instructions. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Glossary*.

R_{ZVBJ}

R_{QVYD} In Transactional state, SVE's Non-fault option causes all memory access faults to be suppressed without causing the transaction to fail.

Instead, the FFR is updated to indicate which of the active Vector elements were not successfully loaded (see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *FFR, First Fault Register*).

C1.9 Reset

R_{KFBV} All the rules described in the *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Reset* section apply whether or not the PE is in Transactional state when a Cold or a Warm reset is asserted.

R_{ZGNK} If the PE resets to AArch64 state using either a Cold or a Warm reset, the PE resets to Non-transactional state.

C1.10 Identification mechanism

R_{XXMT} The implementation of TME is identified by ID_AA64ISAR0_EL1.TME.

I_{WFVR} Although TME defines no instruction enables and disables, or trap controls, Arm recommends the addition of an instruction disable control in ACTLR_ELx for the highest implemented Exception level which if set has the following effect:

- The bits in ID_AA64ISAR0_EL1.TME are RES0.

- The TME instructions are UNDEFINED at EL0 and above.

Chapter C2
Debug, PMU, Trace

C2.1 Self-hosted debug

C2.1.1 Breakpoint Instruction exceptions

R_{BTRM} In Transactional state, executing a *breakpoint instruction* fails the transaction with a DBG cause, without raising a Breakpoint Instruction exception. For more information on breakpoint instructions, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Breakpoint Instruction exceptions*.

I_{FDMB} A transaction with a breakpoint instruction cannot make forward progress; it will always fail. The software is responsible for reading the failure information returned by TSTART and acting accordingly.

C2.1.2 Breakpoint exceptions

R_{HLHZ} In Transactional state, Breakpoint exceptions are suppressed and fail the transaction with a DBG cause. For more information on breakpoint exceptions, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Breakpoint exceptions*.

I_{VMWG} A hardware breakpoint will continuously fail a restarting transaction until either the breakpoint conditions are not met (*e.g.*, the transactional code follows a different execution path), or the breakpoint is disabled. It is the responsibility of the software to detect this situation and act accordingly.

C2.1.3 Watchpoint exceptions

R_{WSMX} In Transactional state, Watchpoint exceptions are suppressed and fail the transaction with a DBG cause. For more information on watchpoint exceptions, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Watchpoint exceptions*.

I_{QHCS} A hardware watchpoint will continuously fail a restarting transaction until either the watchpoint conditions are not met (*e.g.*, the transactional code accesses different Locations), or the watchpoint is disabled. It is the responsibility of the software to detect this situation and act accordingly.

C2.1.4 Software Step exceptions

R_{TCSR} In Non-transactional state, executing a `TSTART` instruction when software step is active-not-pending fails the transaction with DBG cause. For more information on active-not-pending, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Software Step exceptions*.

I_{VYYF} Enabling or disabling software step is not possible in Transactional state because attempting to update MDSCR_EL1.SS fails the transaction. For more information, see Section C1.8.2 *MSR (register)*.

R_{JNVJ} If PSTATE.D is cleared inside a transaction and MDSCR_EL1.SS is 1 when entering Transactional state, the transaction fails with DBG cause.

C2.2 External debug

For the definitions of the various Halting debug events, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Halting Debug Events*.

C2.2.1 Breakpoint and Watchpoint debug events

R_{CLHP} In Transactional state, a Breakpoint debug event or a Watchpoint debug event that would otherwise cause entry to Debug state, fails the transaction with DBG cause without entering Debug state.

For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Breakpoint and Watchpoint debug events*.

C2.2.2 Halting Instruction debug event

R_{CVJX} If EDSCR.HDE == 0 or if halting is prohibited, then executing a HLT instruction in Transactional state fails the transaction with ERR cause.

R_{XHFX} If EDSCR.HDE == 1 and halting is allowed, then executing a HLT instruction in Transactional state fails the transaction with a DBG cause without entering Debug state.

For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Halt Instruction debug event*.

C2.2.3 Halting Step debug events

R_{BWLB} In Non-transactional state, executing a TSTART instruction when *Halting step* is active-not-pending fails the transaction with DBG cause. For more information on *Halting step*, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Halting Step debug events*.

I_{GTHQ} Enabling or disabling Halting step is not possible in Transactional state because attempting to update EDECR.SS fails the transaction as per Section C1.8.2 *MSR (register)*.

For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Halting Step debug events*.

C2.2.4 External Debug Request debug event

R_{BWYJ} If halting is allowed, all of the following applies:

- *External Debug Request* debug events asserted in Transactional state are pended.
- Unmasked External Debug Request debug events are taken when the *Processing Element* (PE) exits Transactional state.
- In the absence of a *Context synchronization event*, it is IMPLEMENTATION DEFINED if the delivery of an unmasked External Debug Request debug event fails the transaction, but the architecture requires that the External Debug Request debug event is taken in finite time as per *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Synchronization and External Debug Request debug events*.
- If the delivery of an unmasked External Debug Request debug event fails the transaction, the failure cause reported is DBG.

See also:

- *External Debug Request debug event* in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

- C1.8.7 *ISB*.

C2.2.5 Reset Catch debug event

R_{BJTP} If halting is allowed, all of the following applies:

- *Reset Catch* debug events asserted in Transactional state are pended and are taken when the PE exits Transactional state.
- In the absence of a *Context synchronization event*, it is IMPLEMENTATION DEFINED if the delivery of a Reset Catch debug event fails the transaction, but the architecture requires that the Reset Catch debug event is taken in finite time as per *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Synchronization and Halting debug events*.
- If the delivery of a Reset Catch debug event fails the transaction, the failure cause reported is DBG.

See also:

- *Reset Catch debug events* in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].
- C1.8.7 *ISB*.

C2.2.6 Other Halting debug events

I_{WTFX} *Exception Catch* debug events cannot occur inside a transaction because an exception entry or exception return cannot occur inside a transaction. For more information on Exception Catch, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Exception Catch debug event*.

I_{JYVR} *OS Unlock Catch* debug events, and *Software Access* debug events cannot occur inside a transaction because they are generated by accesses to System registers that cannot occur inside a transaction.

See also:

- *OS Unlock Catch debug event* in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].
- *Software Access debug event* in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

C2.2.7 Behavior in Debug state

R_{FKXF} The TCOMMIT instruction is unchanged in Debug state.

I_{FDRG} TCOMMIT follows the rules described in the *Any instruction that is* UNDEFINED *in Non-debug state* topic of the *Arm® Architecture Reference Manual, for A-profile architecture* [1] since the PE cannot enter Transactional state in Debug state and TCOMMIT is UNDEFINED in Non-transactional state.

R_{RVKB} TCANCEL and TTEST are CONSTRAINED UNPREDICTABLE in Debug state.

I_{ZKFG} TCANCEL and TTEST follow the rules described in the *Arm® Architecture Reference Manual, for A-profile architecture* [1] *All other instructions*.

R_{NQSW} TSTART is CONSTRAINED UNPREDICTABLE in Debug state.

R_{XYLB} TSTART behaves in one of the following ways:

- It is UNDEFINED.

- It executes as a NOP.

- It does not enter Transactional state and it returns an UNKNOWN value.

C2.2.8 The PC Sample-based Profiling Extension

R_{NPQB} All the rules described in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *The PC Sample-based Profiling Extension* chapter apply to a PE in Transactional state too.

R_{QCCR} Additionally, *Transactional Memory Extension* (TME) extends PPMPCSR to indicate if a sample references an instruction executed in Transactional state or Non-transactional state.

I_{SHHH} Like in Non-transaction state, only reference instructions that were committed for execution are sampled in Transactional state.

I_{VYCF} Samples can reference instructions from failed or canceled transactions.

C2.3 The Statistical Profiling Extension

C2.3.1 Memory accesses by profiling operations

R_{TYGM} The profiling operation executes independently of the instructions that are executed on the PE and acts as a separate memory observer from the PE in the system. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Synchronization and Statistical Profiling*.

R_{XXLF} If a profiling write operation overlaps with the read-set or write-set of a transaction, it is constraint UNPREDICTABLE whether:

- The write has the same effect on the transaction as a store by any other Observer to that address.
- The write has no effect on this transaction.

R_{ZVWK} A profiling operation executes independently of the instruction or instructions that are executed on the PE and acts as a separate memory observer from the PE in the system.

R_{TSLS} Writes to the Profiling Buffer generated by profiling operations in Transactional state are considered non-transactional and as such:

- They are not part of the transactional write set.

- They are observable even if the transaction fails or is canceled.

R_{GJXT} For a sampled operation, if the operation is executed in Transactional state then Events packet.E[16] (Transactional) is set to 1.

S_{PHDN} Software can use PMSEVFR_EL1[16] to filter recording of sampled operations based on the Transactional flag.

C2.3.2 Events packet payload

TME extends existing the *Statistical Profiling Extension* (SPE) protocol with the following events packet payload:

R_{QPZV} **E[16], byte 2 bit [0]**

If TME is not implemented, this bit reads-as-zero. The possible values of this bit are:

0	Operation executed in Non-transactional state.
1	Operation executed in Transactional state.

C2.3.3 Profile Buffer management interrupts

I_{NTMY} See Section C1.7 *Interrupt masking* for the treatment of interrupts in Transactional state.

C2.4 The Embedded Trace Extension

For all information, see Chapter D1 *Embedded Trace Extension*.

C2.5 The Performance Monitors Extension

C2.5.1 Event filtering

I_{WKDW} TME extends the filtering capabilities of the PMU to enable filtering by Transactional state.

R_{NDRV} For each *Attributable* event, if the value of PMEVTYPER<n>_EL0.T is 1, then the event is counted only if the PE is in Transactional state.

Otherwise, for each *Unattributable* event, it is IMPLEMENTATION DEFINED whether the filtering applies.

R_{LFKP} TME adds new events that count transitions between Transactional and Non-transactional states. It is IMPLEMENTATION DEFINED if these events are considered to occur in Transactional or Non-transactional state. See the description of the individual events in Table C2.2 for more details.

I_{LNQD} For the definition of *Attributable* and *Unattributable*, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Attributability*.

C2.5.2 Accuracy of event filtering

R_{QJQP} TME does not require filtering by Transactional state to be accurate. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Accuracy of event filtering*.

I_{WXMC} For many events, during a transition between Transactional and Non-transactional states, events generated by instructions executed in one state can be counted in the other state.

R_{VTYX} It is not permitted for the following events to be counted in the wrong state:

- Any event classified as *Instruction architecturally executed*.
- Any event classified as *Instruction architecturally executed, Condition code check pass*.
- EXC_TAKEN, Exception taken.

I_{VMRV} For the definition of *Instruction architecturally executed*, and *Instruction architecturally executed, Condition code check pass*, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *PMU events and event numbers*.

R_{FWHG} TME adds the following *required* events.

Table C2.2: TME related events

Number	Type	Mnemonic	Description
0x4030	Arch	TSTART_RETIRED	See C2.5.3 *TSTART_RETIRED*
0x4031	Arch	TCOMMIT_RETIRED	See C2.5.4 *TCOMMIT_RETIRED*
0x4032	Arch	TME_TRANSACTION_FAILED	See C2.5.5 *TME_TRANSACTION_FAILED*
0x4034	Arch	TME_INST_RETIRED_COMMITTED	See C2.5.6 *TME_INST_RETIRED_COMMITTED*
0x4035	Microarch	TME_CPU_CYCLES_COMMITTED	See C2.5.7 *TME_CPU_CYCLES_COMMITTED*
0x4038	Microarch	TME_FAILURE_CNCL	See C2.5.8 *TME_FAILURE_CNCL*
0x403A	Microarch	TME_FAILURE_ERR	See C2.5.9 *TME_FAILURE_ERR*

Number	Type	Mnemonic	Description
0x403B	Microarch	TME_FAILURE_IMP	See C2.5.10 *TME_FAILURE_IMP*
0x403C	Microarch	TME_FAILURE_MEM	See C2.5.11 *TME_FAILURE_MEM*
0x4039	Microarch	TME_FAILURE_NEST	See C2.5.12 *TME_FAILURE_NEST*
0x403D	Microarch	TME_FAILURE_SIZE	See C2.5.13 *TME_FAILURE_SIZE*
0x403E	Microarch	TME_FAILURE_TLBI	See C2.5.14 *TME_FAILURE_TLBI*
0x403F	Microarch	TME_FAILURE_WSET	See C2.5.15 *TME_FAILURE_WSET*

C2.5.3 TSTART_RETIRED

R_{NRPB} The counter increments for every architecturally executed TSTART instruction that starts an outer transaction.

R_{HKGP} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TSTART_RETIRED increments the counter.

C2.5.4 TCOMMIT_RETIRED

R_{CKYT} The counter increments for every architecturally executed TCOMMIT instruction that commits an outer transaction.

R_{XCGL} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TCOMMIT_RETIRED increments the counter.

C2.5.5 TME_TRANSACTION_FAILED

R_{YVWS} The counter increments for every transaction that fails or is canceled.

R_{JKPJ} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_TRANSACTION_FAILED increments the counter.

C2.5.6 TME_INST_RETIRED_COMMITTED

R_{LPZF} The counter increments for every architecturally executed instruction in Transactional state if the currently executing transaction commits.

I_{WBGV} It is permissible for an implementation to limit the increment that the execution of a transaction can generate to the counter to a maximum value of 2^{32}-1.

I_{CJMQ} Two possible implementations of this functionality are:

- The implementation accumulates events to the counter directly. If the transaction fails, the counter is restored to the value it had when the transaction started.

- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value.

C2.5.7 TME_CPU_CYCLES_COMMITTED

R_{RMSG} The counter increments on every cycle the PE is in Transactional state if the currently executing transaction commits.

R_{DSFD} All counters are subject to changes in clock frequency, including when a WFI or WFE instruction stops the clock. This means that it is CONSTRAINED UNPREDICTABLE whether or not TME_CPU_CYCLES_COMMITTED continues to increment when the clocks are stopped by WFI and WFE instructions.

R_{JBQT} In a multithreaded implementation, TME_CPU_CYCLES_COMMITTED counts each cycle for the processor for which this PE thread was active and could issue an instruction. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Cycle event counting on multithreaded implementations*.

I_{WMBD} It is permissible for an implementation to limit the increment that the execution of a transaction can generate to the counter to a maximum value of 2^{32}-1.

I_{CFNH} Two possible implementations of this functionality are:

- The implementation accumulates events to the counter directly. If the transaction fails, the counter is restored to the value it had when the transaction started.

- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value.

C2.5.8 TME_FAILURE_CNCL

R_{BTWV} The counter increments for every transaction that fails with CNCL cause.

R_{LVHX} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_CNCL increments the counter.

C2.5.9 TME_FAILURE_ERR

R_{NJXX} The counter increments for every transaction that fails with ERR cause.

R_{XDTJ} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_ERR increments the counter.

C2.5.10 TME_FAILURE_IMP

R_{TCHY} The counter increments for every transaction that fails with IMP cause.

R_{SFBT} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_IMP increments the counter.

C2.5.11 TME_FAILURE_MEM

R_{FFTX} The counter increments for every transaction that fails with MEM cause.

R_{ZTDY} If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_MEM increments the counter.

C2.5.12 TME_FAILURE_NEST

R_{LRJQ} The counter increments for every transaction that fails with NEST cause.

R_{QWVR}   If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_NEST increments the counter.

C2.5.13 TME_FAILURE_SIZE

R_{LDPC}   The counter increments for every transaction that fails with SIZE cause.

R_{BZTQ}   If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_SIZE increments the counter.

C2.5.14 TME_FAILURE_TLBI

R_{MXRY}   The counter increments for every transaction that fails with IMP cause due to the execution of a TLBI instruction by another PE.

R_{FFTW}   If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_TLBI increments the counter.

C2.5.15 TME_FAILURE_WSET

R_{BSJR}   The counter increments for every transaction that fails with SIZE cause due to a memory access that causes an eviction of an entry from the transactional write set.

R_{QZHV}   If PMEVTYPER<n>_EL0.T is 1, it is IMPLEMENTATION DEFINED whether or not TME_FAILURE_WSET increments the counter.

C2.5.16 Behavior on overflow

R_{RGWF}   A Performance Monitors counter overflow while in Transactional state behaves the same as in Non-transactional state. For more information, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Behavior on overflow*.

R_{DGMD}   A Performance Monitors counter that is configured to count the TME_INST_RETIRED_COMMITTED or the TME_CPU_CYCLES_COMMITTED events does not set the overflow status bit in PMOVSCLR if the currently executing transaction fails.

R_{PPFR}   A Performance Monitors counter that is configured to count the TME_INST_RETIRED_COMMITTED or the TME_CPU_CYCLES_COMMITTED events does not generate an overflow interrupt request in Transactional state.

I_{KHXN}   Two possible implementations of this functionality are:

- The implementation accumulates events to the counter directly and sets the overflow status bit when the counter overflows. If the system is programmed to generate an interrupt on overflow, the interrupt is not generated until the transaction commits. If the transaction fails, both the counter and the overflow status bit are restored to the value they had when the transaction started, and no interrupt is generated.

- The implementation accumulates events without updating the counter. If the transaction commits, the counter is updated with the accumulated value. If the counter update overflows the counter value, then the overflow status bit is set, and if the system is programmed to generate an interrupt on overflow, then an interrupt is generated.

Chapter C3
System registers

Transactional Memory Extension (TME) extends existing AArch64 registers with the following fields.

C3.1 General system control registers

C3.1.1 CTR_EL0

R$_{\text{LCCW}}$ **ERG, bits [23:20]**

Exclusives reservation granule, and, if TME is implemented, transactional reservation granule. Log$_2$ of the number of words of the maximum size of the reservation granule for the Load-Exclusive and Store-Exclusive instructions, and, if TME is implemented, for detecting transactional conflicts.

A value of 0b0000 indicates that this register does not provide granule information and the architectural maximum of 512 words (2KB) must be assumed.

Value 0b0001 and values greater than 0b1001 are reserved.

C3.1.2 ID_AA64ISAR0_EL1

R$_{\text{VJLM}}$ **TME, bits [27:24]**

Indicates whether TME instructions are implemented. Defined values are:

0000	No TME instructions are implemented.
0001	TCANCEL, TCOMMIT, TSTART, and TTEST instructions are implemented.

C3.1.3 TCR_EL1

R$_{\text{NJGX}}$ **NFD1, bit [54]**

Present only if SVE or TME is implemented.

Non-fault translation table walk disable for stage 1 translations using TTBR1_EL1.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR1_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.

- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.

- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0	Does not disable stage 1 translation table walks using TTBR1_EL1.
1	A TLB miss on a virtual address that is translated using TTBR1_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

If neither SVE nor TME is implemented, this field is RES0.

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

R_{XNRP} **NFD0, bit [53]**

Present only if SVE or TME is implemented.

Non-fault translation table walk disable for stage 1 translations using TTBR0_EL1.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR0_EL1.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.

- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.

- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0	Does not disable stage 1 translation table walks using TTBR0_EL1.
1	A TLB miss on a virtual address that is translated using TTBR0_EL1 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

If neither SVE nor TME is implemented, this field is RES0.

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

C3.1.4 TCR_EL2

R_{QWCK} **NFD1, bit [54]**

Present only if SVE or TME is implemented.

Non-fault translation table walk disable for stage 1 translations using TTBR1_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR1_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.

- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.

- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0	Does not disable stage 1 translation table walks using TTBR1_EL2.

1   A TLB miss on a virtual address that is translated using TTBR1_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

If neither SVE nor TME is implemented, this field is RES0.

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

R_{MWBR} **NFD0, bit [53]**

Present only if SVE or TME is implemented.

Non-fault translation table walk disable for stage 1 translations using TTBR0_EL2.

This bit controls whether to perform a stage 1 translation table walk in response to a non-fault access from EL0 for a virtual address that is translated using TTBR0_EL2.

If SVE is implemented, the affected access types include:

- All accesses due to an SVE non-fault contiguous load instruction.

- Accesses due to an SVE first-fault gather load that are not for the First active element. Accesses due to an SVE first-fault contiguous load instruction are not affected.

- Accesses due to prefetch instructions might be affected, but the effect is not architecturally visible.

If TME is implemented, the affected access types include all accesses generated by a load or store instruction in Transactional state.

Defined values are:

0   Does not disable stage 1 translation table walks using TTBR0_EL2.

1   A TLB miss on a virtual address that is translated using TTBR0_EL2 due to the specified access types causes the access to fail without taking an exception. No stage 1 translation table walk is performed.

If neither SVE nor TME is implemented, this field is RES0.

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

C3.1.5   ISS encoding for an exception from a `TSTART` instruction

R_{JKTZ} **Bits [24:10]** Reserved, RES0

Rd, Bits [9:5] The Rd value from the issued instruction, the general purpose register used for the destination.

Bits [4:0] Reserved, RES0

C3.1.6   SCTLR_EL1

R_{NBFV} **TMT0, bit [50]**

Forces a trivial implementation of TME at EL0.

The defined values are:

0b0   This control does not cause TSTART instructions to fail.

> 0b1 When the AArch64 TSTART instruction is executed at EL0,
> the transaction fails with TRIVIAL cause.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the *Processing Element* (PE) resets into EL1, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{QHHT}   **TMT, bit [51]**

Forces a trivial implementation of TME at EL1.

The defined values are:

> 0b0 This control does not cause TSTART instructions to fail.
>
> 0b1 When the AArch64 TSTART instruction is executed at EL1,
> the transaction fails with TRIVIAL cause.

In a system where the PE resets into EL1, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{YCNC}   **TME0, bit [52]**

Enables the AArch64 TSTART instruction at EL0, otherwise traps to EL1.

The defined values are:

0b0	Any attempt at EL0 to execute the AArch64 TSTART instruction is trapped to EL1, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.
0b1	This control does not cause TSTART instructions to be trapped.

When ARMv8.1-VHE is implemented, and the value of HCR_EL2.{E2H, TGE} is {1,1}, this bit has no effect on execution at EL0.

In a system where the PE resets into EL1, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{NQPY}   **TME, bit [53]**

Enables the AArch64 TSTART instruction at EL1.

The defined values are:

> 0b0 Any attempt at EL1 to execute the AArch64 TSTART instruction is trapped
> to EL1, (reported with ESR_ELx.EC value 0b011011), subject to the exception

prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes ᴛꜱᴛᴀʀᴛ to

be ᴜɴᴅᴇғɪɴᴇᴅ at EL1.

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

In a system where the PE resets into EL1, this field resets to a value that is architecturally ᴜɴᴋɴᴏᴡɴ.

Otherwise:

Reserved, ʀᴇꜱ0.

C3.1.7 SCTLR_EL2

R_{HYJD}   **TMT0, bit [50]**

When HCR_EL2.{E2H,TGE} is {1,1}, forces a trivial implementation of TME at EL0.

The defined values are:

0b0	This control does not cause ᴛꜱᴛᴀʀᴛ instructions to fail.
0b1	When the AArch64 ᴛꜱᴛᴀʀᴛ instruction is executed at EL0, the transaction fails with TRIVIAL cause.

In a system where the PE resets into EL2, this field resets to a value that is architecturally ᴜɴᴋɴᴏᴡɴ.

Otherwise:

Reserved, ʀᴇꜱ0.

R_{TXJP}   **TMT, bit [51]**

Forces a trivial implementation of TME at EL2.

The defined values are:

0	This control does not cause ᴛꜱᴛᴀʀᴛ instructions to fail.
1	When the AArch64 ᴛꜱᴛᴀʀᴛ instruction is executed at EL2, the transaction fails with TRIVIAL cause.

In a system where the PE resets into EL2, this field resets to a value that is architecturally ᴜɴᴋɴᴏᴡɴ.

Otherwise:

Reserved, ʀᴇꜱ0.

R_{QZST}   **TME0, bit [52]**

When HCR_EL2.{E2H,TGE} is {1,1}, enables the AArch64 ᴛꜱᴛᴀʀᴛ instruction at EL0, otherwise traps to EL2.

The defined values are:

0b0	Any attempt at EL0 to execute the AArch64 ᴛꜱᴛᴀʀᴛ instruction is trapped to EL2, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules,

unless HCR_EL2.TME or SCR_EL3.TME causes TSTART instructions to be UNDEFINED at EL0.

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

In a system where the PE resets into EL2, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{HHMY}   **TME, bit [53]**

Enables the AArch64 TSTART instruction at EL2.

The defined values are:

0b0	Any attempt at EL2 to execute the AArch64 TSTART instruction is trapped to EL2, (reported with ESR_ELx.EC value 0b011011), subject to the exception prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART to be UNDEFINED at EL2.
0b1	This control does not cause TSTART instructions to be trapped.

In a system where the PE resets into EL2, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

C3.1.8 SCTLR_EL3

R_{YDGG}   **TMT, bit [51]**

Forces a trivial implementation of TME at EL3.

The defined values are:

0	This control does not cause TSTART instructions to fail.
1	When the AArch64 TSTART instruction is executed at EL3, the transaction fails with TRIVIAL cause.

In a system where the PE resets into EL3, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

R_{ZRLR}   **TME, bit [53]**

Enables the AArch64 TSTART instruction at EL3.

The defined values are:

0b0	Any attempt at EL3 to execute the AArch64 TSTART instruction is trapped to EL3, (reported with ESR_ELx.EC value 0b011011), subject to the exception

> prioritization rules, unless HCR_EL2.TME or SCR_EL3.TME causes TSTART to
> be UNDEFINED at EL0, EL1 and EL2.
>
> 0b1   This control does not cause TSTART instructions to be trapped.

In a system where the PE resets into EL3, this field resets to a value that is architecturally UNKNOWN.

Otherwise:

Reserved, RES0.

C3.1.9 HCR_EL2

R_{WBJM}   **TME, bit [39]**

Enables the AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions at EL{0,1}.

The defined values are:

> 0b0   The AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions are UNDEFINED
> at EL{0,1}, and EL1 reads from ID_AA64ISAR0_EL1.TME return 0, when EL2 is
> enabled in the current Security state.
>
> 0b1   This control does not cause these instructions to be UNDEFINED.

In a system where the PE resets into EL2, this field resets to a value that is architecturally UNKNOWN.

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

Otherwise:

Reserved, RES0.

C3.1.10 SCR_EL3

R_{XYXB}   **TME, bit [34]**

Enables the AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions at EL{0,1,2}.

The defined values are:

> 0b0   The AArch64 TSTART, TCOMMIT, TTEST and TCANCEL instructions are
> UNDEFINED at EL{0,1,2}, and EL{1,2} reads from ID_AA64ISAR0_EL1.TME return 0.
>
> 0b1   This control does not cause these instructions to be UNDEFINED.

In a system where the PE resets into EL3, this field resets to an architecturally unknown value.

Otherwise:

Reserved, RES0.

C3.2 Performance Monitors registers

C3.2.1 PMEVTYPER<n>_EL0

R_{DQWK} **T, bit [23]**

Transactional state filtering bit. Controls counting in Transactional state. If TME is not implemented, this bit is RES0. The possible values of this bit are:

0	Count events in Non-transactional state and in Transactional state.
1	Count events in Transactional state only.

C3.2.2 PMCCFILTR_EL0

R_{KCZZ} **T, bit [23]**

Non-transactional state filtering bit. Controls counting in Non-transactional state. If TME or PMUv3 are not implemented, this bit is RES0. The possible values of this bit are:

0	Count cycles in Non-transactional state and in Transactional state.
1	Count cycles in Transactional state only.

This bit resets to an architecturally UNKNOWN value on a reset.

C3.2.3 PMSEVFR_EL1

R_{RHWX} **E, bit [16]**

Transactional. The possible values of this bit are:

0	Transactional event is ignored.
1	Do not record samples that have event 16 (Transactional) == 0.

This bit is ignored by the PE when PMSFCR_EL1.FE == 0.

This bit resets to an architecturally UNKNOWN value on a reset.

C3.3 Performance Monitors external registers

C3.3.1 PMPCSR

R_{RCJT}   **T, bit [60]**

Transactional state of the sample. Indicates the Transactional state that is associated with the most recent PMPCSR sample or, when it is read as a single atomic 64-bit read, the current PMPCSR sample.

0	Sample is from Non-transactional state.
1	Sample is from Transactional state.

This field resets to a value that is architecturally UNKNOWN.

Chapter C4
Instructions

Transactional Memory Extension (TME) adds the following instructions.

C4.1 `TCANCEL`

R_{VPTY} The `TCANCEL` instruction exits Transactional state and discards all state modifications that are due to instructions that were executed transactionally.

R_{DSCF} Execution continues at the instruction that follows the `TSTART` instruction of the outer transaction.

R_{YFDC} The destination register of the `TSTART` instruction of the outer transaction is written with the immediate operand of `TCANCEL`.

```
TCANCEL #<imm>
```

C4.2 **TCOMMIT**

R_{VVNT} The TCOMMIT instruction commits the current transaction.

R_{YHKK} If the current transaction is an outer transaction, then Transactional state is exited, and all state modifications due to instructions that were executed transactionally are committed to the architectural state.

R_{XJVQ} TCOMMIT takes no inputs and returns no value.

R_{SJJW} Execution of TCOMMIT is UNDEFINED in Non-transactional state.

```
TCOMMIT
```

C4.3 **TSTART**

R_{WBJV} This instruction starts a new transaction.

R_{NXXP} If the transaction started successfully, the destination register is set to zero.

R_{VNLM} If the transaction failed or was canceled, then all state modifications that are due to instructions that were executed transactionally are discarded and the destination registers is written with a non-zero value that encodes the cause of the failure.

```
TSTART <Xd>
```

C4.4 `TTEST`

R_{FZLN} The TTEST instruction takes no inputs.

R_{VLYG} The TTEST instruction writes the depth of the transaction to the destination register, or the value 0 otherwise.

```
TTEST <Xd>
```

Chapter C5
Interaction with Memory Tagging Extension

This section describes the interaction of *Transactional Memory Extension* (TME) with the Memory Tagging Extension introduced in v8.5.

R_{SKRL} The MTE instructions for Tag generation, Tag setting and getting, are allowed within a transaction. This means in particular that the accesses to GCR_EL1 and RGSR_EL1 stemming from the MTE instructions are allowed within a transaction, but it is IMPLEMENTATION DEFINED whether they are checkpointed.

R_{MYGP} In the case of an asynchronous Tag Check Failure within a Transaction:

- Tag check failures configured to asynchronously accumulate failure status should not expect transaction failure with ERR cause.
- If the transaction succeeds then reading TFSR_ELx.TFy status determines if there are any errors.

Chapter C6
Transactional Memory Extension additional reading

For more information about the Transactional Memory Extension and litmus tests, elision locks and implementation recommendations, see the appendixes of this document.

- Chapter G4 Transactional Memory Extension *(TME) Litmus tests*
- Chapter G5 Transactional Memory Extension *(TME) Transactional Lock Elision*
- Chapter G6 Transactional Memory Extension *(TME) Implementation recommendations*

Part D
The Embedded Trace Extension

Chapter D1
Embedded Trace Extension

D1.1 Introduction

I_{JTPNL} The FEAT_ETE describes the operation of a *trace unit*. The trace unit provides details about software control flow running on a *Processing Element* (PE) which can be used to aid debugging or optimizing. The trace unit provides filtering functionality to allow the targeting of the information to specific code regions or periods of operation.

I_{LVKQS} The FEAT_ETE overlaps with the ETMv4 architecture *Arm® Embedded Trace Macrocell Architecture Specification ETMv4* [3]. The FEAT_ETE has additions to support new architecture features, and does not support all the features of ETMv4. Readers familiar with ETMv4 should refer to Chapter D16 *Functional Differences from ETMv4*.

D1.1.1 Mathematical notation

To aid the understanding of some of the functions defined by the ETE architecture are described in mathematical notation. This table provides a description and examples of the symbols used within this document.

Symbol	Function	Example
A $\land$ B	AND	$0 \land 0 = 0$
		$1 \land 0 = 0$
		$0 \land 1 = 0$
		$1 \land 1 = 1$

Symbol	Function	Example
A $\lor$ B	OR	$0 \lor 0 = 0$
		$1 \lor 0 = 1$
		$0 \lor 1 = 1$
		$1 \lor 1 = 1$
$\neg$A	NOT(A)	$\neg 0 = 1$
		$\neg 1 = 0$
$\displaystyle\prod_n f(n)$	Product	$f(0) \times f(1) \times \cdots \times f(N) \equiv f(0) \land f(1) \land \cdots \land f(N)$
$\displaystyle\sum_n f(n)$	Sum	$f(0) + f(1) + \cdots + f(N) \equiv f(0) \lor f(1) \lor \cdots \lor f(N)$
$x \mod q$	modular	$5 \mod 4 = 1$

D1.2 Attributes of tracing

X_{RGLSR} The attributes of PE tracing are:

Trace is generated in real time   Trace provides a means of observing the PE operation while the PE is running. For diagnostic purposes, this is useful as some types of erroneous behavior are only solvable by observing the system during runtime. In addition, because the PE trace can include cycle counts, trace can be used for PE profiling purposes.

Trace has a minimal effect on functional performance of the PE   Usually, trace has no effect on the functional performance of the PE. This attribute does depend on the market use of the PE being debugged, however, and on the trace requirements for the PE and the trace solution that is adopted to meet those requirements. For some markets, some impact on PE performance is acceptable but for others, most notably in real-time systems, an impact on PE performance might be unacceptable.

Trace is available for self-hosted analysis   The trace from a PE or process is available for analysis by software running on the target. See D1.3 *Self-hosted Trace*.

Trace is deeply embedded in an SoC   Trace provides a method of debugging software executing on PEs that are deeply embedded within an SoC.

Trace is available for external analysis   The trace from a PE or process can be exported off chip for analysis by external tools.

D1.3 Self-hosted Trace

X_{FNMRL} Self-hosted trace is used for various purposes, including:

Non-invasive single stepping The trace provides a history of execution similar to that obtained by single-stepping through code.

Failure logging This is similar to a stack trace dump when a failure occurs.

Performance analysis The trace might be used with other trace sources or performance analysis units to analyze program performance.

Capturing the trace on-chip involves either:

Use of system memory The trace output from the trace unit is directed to a buffer in main system memory via the Trace Buffer Unit defined by FEAT_TRBE.

Use of existing shared system memory, where some main system memory is reserved for trace capture
The trace output from the trace unit is directed to the reserved memory over the main system bus, typically using CoreSight *Embedded Trace Router* (ETR).

Use of a dedicated on-chip buffer The trace output from the trace unit is directed to the dedicated memory, typically using CoreSight technology such as a CoreSight *Embedded Trace Buffer* (ETB). A dedicated bus such as AMBA ATB is also usually implemented between the trace unit and the dedicated memory. Use of dedicated memory means that PE tracing can be performed with zero or minimal effect on system behavior.

AMBA ATB is defined by the *AMBA ATB Protocol Specification* [4].

See also:

- Chapter E1 *Trace Buffer Extension*

D1.4 External Debug

X_{TQGBH} External debug is commonly used in trace applications that require long-term logging of behavior. In addition, external debug is more likely to be used when the impact of PE tracing on system performance must be minimized.

For example, external debug might be used:

- For debugging real-time systems.
- When analyzing programs that do not frequently vary their behavior.
- For debugging software, where a history of execution is required up to the point of failure.

Exporting the trace off-chip usually involves one of the following methodologies:

Real-time continuous export

X_{VJBMM} This can be done using either:

- A dedicated trace port capable of sustaining the bandwidth of the trace.
- An existing interface on the SoC, such as a USB or other high-speed port.

Use of a dedicated trace port means that the trace can be exported off-chip with zero or minimum effect on system behavior. An existing interface is usually used when system constraints, such as cost or package size, mean that a dedicated trace port is not possible. However, use of an existing interface might affect system behavior, because both trace and normal interface traffic use the same port.

Short-term on-chip capture with subsequent low speed export

X_{PXSSB} This option is used when a low-cost method of exporting the trace is required, or when system constraints prevent real-time continuous export. The trace output from the trace unit is stored temporarily on-chip, and then exported using either:

- An existing debug port on the SoC, such as a JTAG-DP or SW-DP.
- Another existing interface on the SoC, such as USB.

Typically, the temporary storage is a circular buffer. If the buffer is full, newer trace overwrites older trace, which means that the buffer always contains the most recent trace. In SoCs that employ Arm CoreSight technology, a dedicated Embedded Trace Buffer (ETB) is provided for the on-chip capture of trace.

D1.5 Trace output

R_{SVDBD} The trace unit outputs the trace byte stream to one or more of the following:

- The Trace Buffer Extension.
- A CoreSight subsystem, via an AMBA ATB interface.
- One or more IMPLEMENTATION DEFINED interfaces.

R_{LGVCX} If the Trace Buffer Unit is enabled, the trace byte stream is only output to the Trace Buffer Unit.

R_{FJFNS} If the Trace Buffer Unit is disabled, the trace byte stream is output to one or more of the other interfaces.

I_{TVDMT} If the trace output is captured by a single capture point dedicated to a PE, and does not mix with any other trace streams, then the value programmed in to TRCTRACEIDR.TRACEID does not need to be unique among all values programmed in to all trace units in the system. For example, the value $0x01$ could be programmed in to all trace units that have their own dedicated capture point.

- An example of a dedicated trace capture point is the Trace Buffer Extension, and when the Trace Buffer Extension is enabled is the value of TRCTRACEIDR.TRACEID can be the same for all PEs that are using the Trace Buffer Extension.
- Another example of a dedicated trace capture point is a CoreSight ETR connected via an AMBA ATB interface and dedicated to a PE. If the AMBA ATB interface connects to the ETR without mixing with any other trace streams, then when using the ETR and when ensuring the trace does not go to any other trace capture points, TRCTRACEIDR.TRACEID does not need to be unique.

R_{RWPFG} If an AMBA ATB interface is implemented, the trace unit must support all of the following:

- ATB triggers, as defined in TRCIDR5.ATBTRIG.
- A 7-bit trace ID, as defined in TRCIDR5.TRACEIDSIZE.

I_{QJKVW} If the trace unit implements an AMBA ATB interface, or an IMPLEMENTATION DEFINED interface for trace output, Arm recommends that this path is not affected by a Warm reset of the PE. This ensures tracing is possible through a Warm reset of the PE, which is useful for low level debugging scenarios.

R_{NLSSL} While all trace outputs are disabled, the trace unit considers any generated trace data as having been output.

See also:

- Chapter E1 *Trace Buffer Extension*

D1.6 Trace Sessions

R_{GLTKQ} At any one time, the trace unit is either enabled or disabled. See D8.3 *Trace unit programming states* for more details on the states of the trace unit.

R_{XBPSQ} A *trace session* is the period between the trace unit becoming enabled, and when the trace unit next becomes disabled.

R_{MTLFH} While the trace unit is enabled, the ViewInst function is either *active* or *inactive*. While ViewInst is *active*, the trace unit generates trace for instructions that are executed, unless trace generation is inoperative.

R_{ZVNKV} Trace generation is *operative* while neither of the following conditions exist:

* The trace unit is disabled.
* The trace unit is recovering from a trace unit buffer overflow.

R_{MFNWB} Whether ViewInst is active or inactive is independent of whether trace generation is operative or inoperative.

R_{RDPSW} Trace generation *becomes operative* when trace generation transitions from being *inoperative* to *operative*, and occurs:

* When the trace unit transitions from being disabled to being enabled.
* When the trace unit recovers from a trace unit buffer overflow.

R_{BDRKW} Trace generation *becomes inoperative* when trace generation transitions from being *operative* to *inoperative*, and occurs:

* When the trace unit transitions from being enabled to being disabled.
* When the trace unit encounters a trace unit buffer overflow.

R_{LDDLP} When the trace unit is unable to generate at least one trace packet which is required by the architecture, a *trace unit buffer overflow* occurs.

I_{HDJWW} A *trace unit buffer overflow* is usually caused when any buffering in the trace unit is unable to receive any more trace packets. Such inability to receive more trace packets is often caused by being unable to sustain output of trace packets to any trace capture infrastructure.

Note

A *trace unit buffer overflow* is independent of the Trace Buffer Unit filling or wrapping a trace buffer in memory. However a *trace unit buffer overflow* might be caused by the Trace Buffer Unit rejecting trace data due to insufficient capacity, and the limit of any trace unit internal buffers is subsequently reached.

D1.7 Elements

I_{FNFWZ} The elements form an *Abstract Syntax Tree* (AST) which is used to describe the control flow of program execution. Different sequences of the elements can be used to imply the same operation. In this way the FEAT_ETE can be used by different micro-architectures. This is similar to the approach used in previous trace protocols, see *Arm® Embedded Trace Macrocell Architecture Specification ETMv4* [3].

I_{XXBMZ} A trace unit compresses the information on the execution of the PE and outputs a trace byte stream that comprises multiple packets of encoded data. Compression techniques that are used include:

The instruction trace element stream does not contain an element for every executed instruction Instead, the trace unit generates *P0 elements* in the trace element stream when certain types of instruction are executed. These certain types of instructions are known as *P0 instructions*. A *P0 element* acts as a signpost in the program flow, indicating that execution is proceeding along a given path.

Consequently, the stream of *P0 elements* implies the execution of a greater number of instructions, and a trace analyzer can reconstruct the stream of instructions that are executed between *P0 elements* by using the *P0 element* stream and the program image.

Multiple elements can be encoded into a single packet Common sequences of elements are encoded into single packets.

The trace unit can remove program addresses from the trace element stream The trace analyzer can infer the addresses from the program image and previous history. This includes the targets of direct branch instructions, where the target address is encoded in the instruction itself.

Removal of predictable elements Some elements can be removed from the AST representation if the contents of the element can be predicted by previous control flow choices in the software flow. For example the *Target Address element* for returning from a subroutine might not be required if the branch to the subroutine has been traced.

D1.8 Layer Model

I_{XMFJT} The FEAT_ETE is based on a layer model. Each layer deals with a unique aspect of tracing the PE.
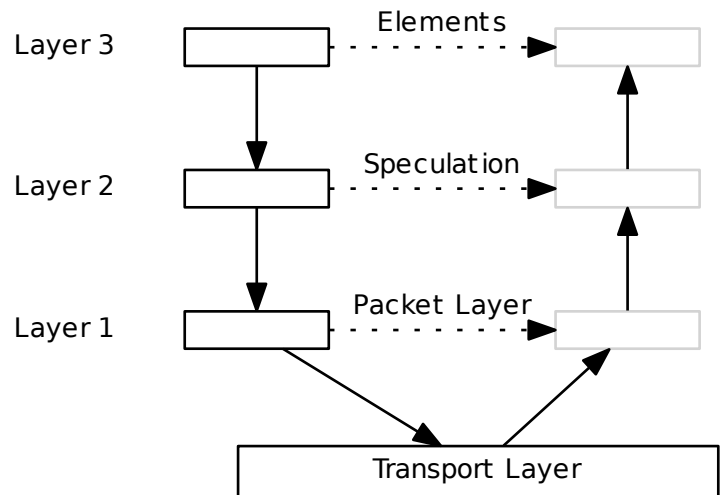
Figure D1.1: Layer model for compression and decompression

Transport Layer

I_{GLPQZ} The transport layer either provides a path off chip or a path to a memory buffer for trace to be stored.

Layer 1

Layer 1 provides compression by:

- Grouping elements together to form packets.
- Removing elements that can be implied.
- Compression against previous values.
- Leading zero compression.
- Reordering of elements.

Layer 2

Layer 2 provides:

- Speculation resolution.
- Transactional Memory resolution.

Layer 3

At layer 3:

- PE behavior is converted into elements.
- Compression is achieved by removing elements which can be predicted using the program image:
  - Direct branch target addresses.
  - Return stack optimization.
- Requires knowledge of the application to decompress:
  - Processes that modify the instruction opcodes require additional information to allow debugging.

D1.9 Trace protocol synchronization

I_{CLTCM} The trace byte stream of a trace unit is typically stored in a circular buffer where, if the buffer is full, newer trace overwrites older trace. To ensure that a trace stream can be analyzed when the trace has been stored in a circular buffer, a trace unit must periodically generate trace protocol synchronization points in the trace byte stream.

I_{BPNSY} To understand the different levels the following elements or packets are used to provide synchronization information in the different layers.

Table D1.2: Control of each layer.

Layer	Control
Layer 3	*Context element* and *Target Address element*
Layer 2	*Trace Info element*
Layer 1	Trace Info packet
Transport Layer	Alignment Synchronization packet

I_{SFXXD} Whenever a trace analyzer receives a Trace Info packet, the trace analyzer receives information about the current state of the trace. However, the trace analyzer cannot begin analysis of program execution until it knows the context in which instructions are being executed and it has an instruction address to start analysis from.

R_{PGHPW} When a *Trace Info element* is generated, the trace unit generates a *Context element* and a *Target Address element* soon after the *Trace Info element*.

> **Note**
>
> There are common use cases where the ratio between the number of bytes associated with trace protocol synchronization and other trace bytes increases significantly, resulting in a degradation of the usability of the trace. Therefore Arm recommends that trace protocol synchronization only occurs when required.

D1.9.1 Non-periodic trace protocol synchronization

R_{QZRMQ} When the trace unit becomes operative, non-periodic trace protocol synchronization occurs.

R_{TTLJC} When non-periodic trace protocol synchronization occurs, the trace unit generates an Alignment Synchronization packet in the trace byte stream before any other trace packets are generated.

R_{HMDGL} When non-periodic trace protocol synchronization occurs, the trace unit generates a *Trace Info element* in the trace element stream before any other trace elements are generated, except *Event elements*.

I_{MQNBT} Arm recommends that if a trace protocol synchronization request occurs while ViewInst is inactive, the Alignment Synchronization packet is not output in the trace byte stream until just before either:

- ViewInst becomes active.
- An Event packet is output.

D1.9.2 Periodic trace protocol synchronization

I_{YPRYM} The trace unit can be programmed to generate trace protocol synchronization requests on a periodic basis, so that the trace element streams and the trace byte streams can be analyzed when stored in a circular trace buffer. TRCSYNCPR.PERIOD controls periodic trace protocol synchronization requests.

I_{NTFYC} Periodic trace protocol synchronization can also be requested by the trace capture infrastructure, for example if a trace protocol synchronization request is received on an Arm AMBA ATB interface *AMBA ATB Protocol Specification* [4].

R_{QHHSY} When periodic trace protocol synchronization is requested, either by TRCSYNCPR.PERIOD or by other sources, the trace unit performs periodic trace protocol synchronization.

R_{VMPYW} When periodic trace protocol synchronization occurs, the trace unit generates an Alignment Synchronization packet and then generates a *Trace Info element*.

I_{QYQRY} Arm recommends that an Alignment Synchronization packet is only output in the trace byte stream if other trace packets have been output since the previous Alignment Synchronization packet. This strategy reduces the risk of a circular buffer filling and overwriting trace.

I_{NQYXW} If two or more periodic trace protocol synchronization requests occur, and no trace is generated between these two requests, then Arm recommends that a non-periodic trace protocol synchronization occurs before any further trace is generated. This ensures that when tracing has been inactive for a long period of time, the trace stream is fully synchronized when tracing is re-activated.

D1.9.3 Synchronization of instruction trace

R_{KKQGK} When non-periodic trace protocol synchronization occurs, the trace unit generates a *Context element* and a *Target Address element* before any *P0 elements* are generated, to provide the trace analyzer with *Context information* and *Address information*.

R_{SVGNN} When periodic trace protocol synchronization occurs, and ViewInst is active when the corresponding *Trace Info element* is generated, the trace unit generates a *Context element* and a *Target Address element* which provide the *Context information* and *Address information* for the target of the most recent non-canceled *P0 element*.

Note

If the trace unit generates the *Context element* and *Target Address element* immediately after the *Trace Info element*, then the most recent non-canceled *P0 element* might have occurred before the *Trace Info element*.

```
                                                    P0

                                                    P0
                                                        ↖
                           Program      Trace          ⋮ target of
                           flow         Info           ⋮
                            │                          ⋮
                            │           Target
                            │           Address
                            │
                            ▼
                                                    P0
```
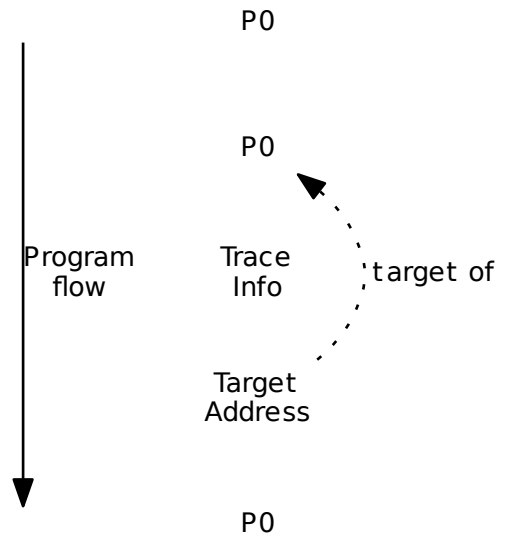
Figure D1.2: Example of Target Address element after Trace Info element.

R_{DLPYX} When periodic trace protocol synchronization occurs, and ViewInst is inactive when the corresponding *Trace Info element* is generated, when ViewInst becomes active and a *Trace On element* is generated, the trace unit generates a *Context element* and a *Target Address element* before any *Atom elements*, *Q elements*, or *Exception elements* are generated, to provide the trace analyzer with *Context information* and *Address information*.

```
                                                    Trace
                                                    Info

                                                    Trace
                                                    On
                            │
                           Program
                           flow
                            │           Target
                            │           Address
                            │
                            ▼
                                                    P0
```
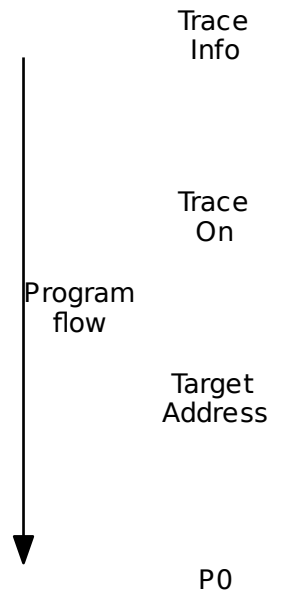
Figure D1.3: Example of Target Address element after Trace Info element in a filtered region.

I_{YZPCB} If a *Cancel element* cancels any *P0 elements* before a *Trace Info element*, then the trace analyzer discards all of the following:

- The canceled *P0 elements*.
- The *Trace Info element*.

- All elements after the *Trace Info element*, up to and including the *Cancel element*. This includes any *Context elements* or *Target Address elements*.

Note

In this scenario, information from the canceled *Trace Info element* can still be used.
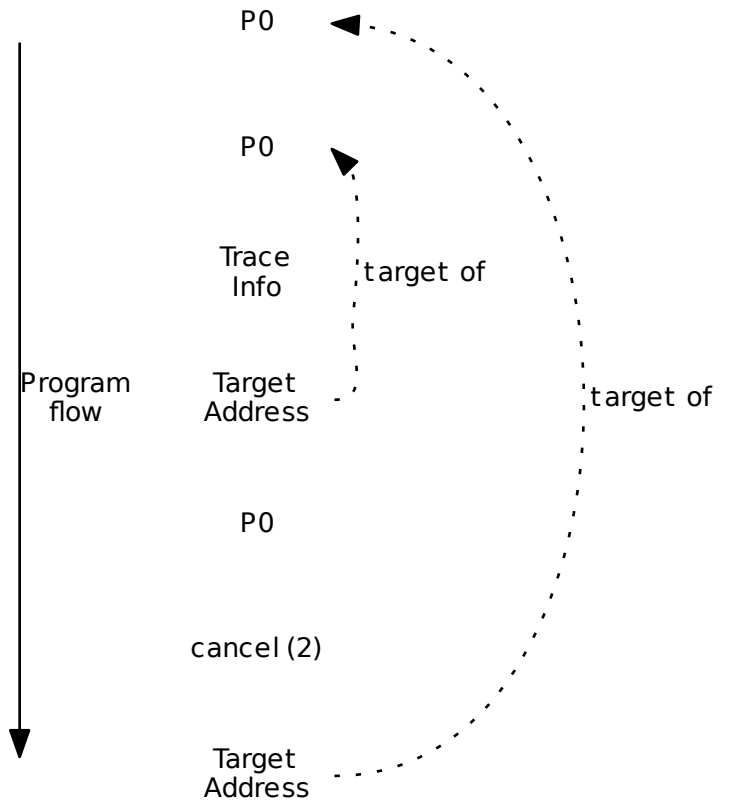
Figure D1.4: Example of Target Address element after Trace Info element in a mispredicted region.

R_{KGPTB}   When a *Cancel element* is generated which cancels any *P0 elements* before a *Trace Info element*, the trace unit generates a new *Context element* and a new *Target Address element*, which indicate the target of the most recent *P0 element* that has not been canceled.

I_{CHTFM}   The *Target Address element* and *Context element* might indicate the target of a *P0 element* from before the *Trace Info element*, or might be delayed until after the next *P0 element*, and therefore indicate the target of that *P0 element*.

Note

If the trace unit generates the new *Context element* and *Target Address element* prior to the next new *P0 element*, then this might prevent the indication of execution of some instructions before the *Trace Info element*.

I_{NSWTK}   If the *Cancel element* cancels all *P0 elements* after a *Trace Info element* but no *P0 elements* prior to the *Trace Info element*, then it might be necessary for the trace unit to immediately generate a *Context element* and *Target Address*

element. This is because a *Context element* and *Target Address element* might have been present in the element stream after the *Trace Info element*, and those *Context elements* and *Target Address elements* are now discarded.

D1.10 Speculation in the trace element stream

I_{NVBWS} The FEAT_ETE architecture supports the correction of trace. This might be because of:

- Tracing of speculative execution of instructions by a PE.
- For some implementations, the tracing of the Transactional Memory Extension.

I_{RTJNK} An FEAT_ETE trace unit traces speculatively-executed instructions in the same way as all other instructions, so that both speculatively-executed instructions and architecturally-executed instructions appear in the instruction trace element stream. This means that some of the program execution information that is shown in the trace element stream might be incorrect, because some of the speculatively executed instructions might be mis-speculated.

Note

The level of speculation that is revealed in the trace is IMPLEMENTATION SPECIFIC.

I_{XLLKT} The trace unit resolves this speculation by generating elements to confirm the status of each instruction in the instruction trace element stream. That is, the trace unit generates elements to show whether each instruction has been committed for execution, or canceled because of mis-speculation. This means that a trace analyzer does not know the status of a traced instruction until the trace analyzer receives an element that indicates whether the instruction has been committed for execution, or canceled because the instruction was mis-speculated.

R_{ZJJKY} When speculatively-executed instructions are traced, the trace unit subsequently generates elements that indicate whether the instructions have been committed for execution, or have been canceled.

I_{KYXKZ} A trace analyzer takes the appropriate action, which might involve canceling some trace elements, to determine the actual program execution.

I_{GGFML} Elements that resolve the status of a traced instruction are called speculation resolution elements. See D2.5 *Speculation Resolution Elements*.

R_{KYGRF} When trace is generated for speculative execution, for mis-speculated execution, the trace unit does not trace any information that cannot be accessed by software executing at the same or at a lower level of privilege than the mis-speculated execution.

R_{QHQLY} When a Context synchronization event is speculated as being taken or executed, the trace unit does not generate trace for any speculative execution after the Context synchronization event until the Context synchronization event is resolved.

R_{LWJCQ} When a speculated Context synchronization event is resolved as being not taken or not executed, the trace unit does not generate trace for mis-speculated execution that occurred after the Context synchronization event.

R_{YGSGJ} When an exit from a Trace Prohibited region is speculated as being taken, the trace unit does not generate trace for any speculative execution after the exit from the Trace Prohibited region, until the exit from the Trace Prohibited region is resolved.

R_{SRLCG} When a speculated exit from a Trace Prohibited region is resolved as being not taken, the trace unit does not generate trace for mis-speculated execution that occurred after the exit from a Trace Prohibited region.

D1.10.1 Tracing Transactions

I_{KBTHL} The Transactional Memory Extension defines the *Transactional state*. For instructions executed in Transactional state, the trace stream indicates which instructions are executed in Transactional state, and provides indicators for a trace analyzer to determine whether the transaction was successful or failed.

I_{FWGBM} If the instruction is executed in Transactional state then the result of the instruction is not known until the transaction succeeds or fails. Transactions can be of an arbitrary length and can be nested, so the ETE architecture does not guarantee an entire transaction is traced, if any of the transaction is traced.

I_{VJTLG} The execution of transactions is represented in the trace element stream by 3 elements:

- *Transaction Start element*.
- *Transaction Commit element*.
- *Transaction Failure element*.

These provide markers in the trace element stream to indicate the sections which represent transactions. The *Transaction Start element* indicates that any following instructions are executed in Transactional state. When the PE leaves Transactional state, either the *Transaction Commit element* or *Transaction Failure element* are traced to indicate the resolution of the transaction.

I_{QZNBZ} An entry to Transactional state might be traced using a *Transaction Start element* and the subsequent exit from Transactional state might be traced, without tracing any execution in Transactional state. There might have been no execution in Transactional state, or the trace unit might have been programmed to not trace such execution.

See also:

- Chapter C1 *Transactional Memory Extension*

D1.10.1.1 Implementation flexibility

R_{VVFQZ} If no speculation in the trace element stream is implemented, TRCIDR8.MAXSPEC == 0x0 and TRCIDR0.COMMTRANS indicates that the *Transaction Start element* is a *P0 element*.

D1.10.1.2 Filtering of trace

I_{ZYNHF} The ETE architecture supports filtering of the trace within a transaction.

I_{BRWSS} Filtering of a transaction can be due to any of the following:

- The ViewInst function.
- Prohibited regions.
- Asynchronous events.

I_{ZNYSY} Due to filtering the start of the transaction might not necessarily be traced. See the *Transaction Start element* for details.

I_{VXTQS} Due to filtering the end of a transaction might not necessarily be traced. See the *Transaction Commit element* and *Transaction Failure element* for details.

X_{PCSKD} If an instruction is traced which was executed in Transactional state, then the trace analyzer must be aware, so that the effect of the instructions executed in the Transactional state can be determined.

R_{NMWFJ} When an instruction is traced and the PE is in Transactional state, the trace unit traces the result of the transaction unless any of the following occur:

- The trace unit becomes disabled.
- A trace unit buffer overflow occurs.
- The PE enters a Trace Prohibited region.

In the above scenarios, the trace unit generates a *Transaction Failure element*, and the resolution of the transaction is UNKNOWN.

Chapter D2
Trace Element Model

This chapter provides details on the different elements used to create an *Abstract Syntax Tree* (AST) for describing the software control sequence.

D2.1 Trace Info element

I_{LBSZF} A *Trace Info element* provides a point in the trace element stream where analysis of the trace element stream can begin.

Trace Info elements include setup information about:

- The static trace programming that does not change during a trace session, including:

    – Whether cycle counting is enabled, and if enabled, the cycle count threshold.

- Dynamic information that might change during a trace session, such as:

    – The speculation depth. This indicates how many unresolved *P0 elements* were traced before the *Trace Info element*.
    – Whether the *Processing Element* (PE) trace unit has traced that the PE is executing in Transactional state.

D2.2 P0 element

I_{LLDBJ} *P0 elements* imply the execution of instructions.

I_{XPZXL} *P0 elements* are generated speculatively and must be either committed or canceled (see D2.5 *Speculation Resolution Elements*).

R_{XVHWG} *P0 elements* must be generated in simple sequential execution order.

D2.2.1 Atom Element

I_{XPFJG} An *Atom element* implies that one or more instructions have been executed, up to and including the next *P0 instruction*. Only certain instructions generate an *Atom element*. See Chapter D3 *Instruction and Exception classifications* for details of these instructions.

R_{PRNZH} The *Atom element* is one of the following types:

- E Atom.
- N Atom.

I_{CYMYM} The meaning of the type of an *Atom element* is dependent on the instruction it is encoding. For example, branch instructions are represented as an E *Atom element* if the branch was taken and an N *Atom element* if not taken.

D2.2.2 Exception Element

I_{YVMSC} An *Exception element* indicates a change in program flow which cannot be calculated by the analysis of the program image, or which is caused by an instruction which is not a *P0 instruction*. Such a change in program flow is described as an *Exceptional occurrence*.

R_{MKPFJ} An *Exceptional occurrence* consists of the following:

- PE Architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

Note

Transaction failure is not classified as an *Exceptional occurrence*, although it is traced using an Exception packet.

I_{JLZPY} An *Exception element* indicates:

- That an *Exceptional occurrence* has occurred.
- The type of *Exceptional occurrence*.
- The virtual address where the *Exceptional occurrence* was taken from, also known as the preferred exception return address.

R_{DXJBQ} The instruction set for the preferred exception return address for a *Exception element* is one of the following:

- AArch64 A64.
- AArch32 A32.
- AArch32 T32.

R_{YPPRH} An *Exception element* is a *P0 element*.

D2.2.2.1 PE Architectural exceptions

R_{PZRFL} The following exception types are used to indicate PE Architectural exceptions:

- IRQ.
- FIQ.
- Trap.
- Call.
- Inst fault.
- Data fault.
- Inst debug.
- Data debug.
- Alignment.
- System Error.
- Debug halt.

See Chapter D3 *Instruction and Exception classifications* for details of the mapping between the PE Architectural exceptions and these exception types.

R_{SFYMW} Table D2.1 defines the preferred exception return address for each exception type for PE Architectural exceptions.

Table D2.1: Preferred exception return address for PE Architectural exceptions

Exception type	Preferred exception return address
IRQ	Instruction after the last executed instruction
FIQ	Instruction after the last executed instruction
Trap	For a trapped instruction or UNDEFINED instruction, the preferred exception return address is the address of the instruction. For a trapped exception, the preferred exception return address is the address of the instruction that caused the exception.
Call	Instruction after the call instruction
Inst fault	Instruction that caused the exception
Data fault	Instruction that caused the exception
Inst debug	Instruction that caused the exception
Data debug	Instruction that caused the exception
Alignment	Instruction that caused the alignment exception
System Error	Instruction after the last executed instruction
Debug halt	The instruction after the last executed instruction, that is, the value loaded into the DLR register.

I_{GZKGC} The nature of System Error means that execution might not complete up to the preferred exception return address, or it might perform some operations after the preferred exception return address. This behavior is IMPLEMENTATION DEFINED and might vary depending on the cause of the exception.

R_{GFJZF} When an imprecise System Error exception occurs, the preferred exception return address is the address stored in the relevant ELR when the exception is taken.

S_{GKMTH} When a System Error exception occurs, the trace analyzer must be aware that the preferred exception return address might not indicate the exact point at which program execution was interrupted. The trace analyzer should not rely on the preferred exception return address for inferring exactly which instructions were executed. This behavior only occurs for imprecise System Error exceptions.

R_{LBLWT} When an imprecise Debug halt exception occurs, the preferred exception return address is the address stored in DLR or DLR_EL0 when the exception is taken.

S_{RDJXM} When an imprecise Debug halt exception occurs, the trace analyzer must be aware that the preferred exception return address might not indicate the exact point at which program execution was interrupted. The trace analyzer should not rely on the preferred exception return address for inferring exactly which instructions were executed. An imprecise Debug halt exception can only occur under direct control of a debugger, usually by controlling EDRCR.CBRRQ.

D2.2.2.2 ETE defined exceptions

R_{MZJTJ} In addition to the Arm Architectural exceptions, the ETE specifies the following *Exceptional occurrences* that are traced using *Exception elements*:

- PE Reset, which indicates that a PE Warm reset has occurred.

R_{NRJGC} Table D2.2 defines the preferred exception return address for each exception type for ETE defined exceptions.

Table D2.2: Preferred exception return address for ETE defined exceptions

Exception type	Preferred exception return address
PE Reset	UNKNOWN

R_{JRNYF} When a PE Reset occurs, the preferred exception return address and context are UNKNOWN. Therefore for an *Exception element* indicating a PE Reset the preferred exception return address and context are UNKNOWN. No instruction execution is indicated between the previous *P0 element* and the *Exception element*.

I_{QJYYZ} When an *Exception element* indicating a PE Reset occurs:

- The target address and target context of the previous *P0 element* might be UNKNOWN.
- If there are no *P0 elements* between a *Trace On element* and the *Exception element*, then the initial address and context after the previous *Trace On element* might be UNKNOWN.

D2.2.2.3 IMPLEMENTATION DEFINED exceptions

R_{ZVYQW} ETE defines some exception types which are IMPLEMENTATION DEFINED, including but not limited to:

- Error Correction Code (ECC) error correction.
- Generic replay of program execution.

I_{XHFLL} The use of the IMPLEMENTATION DEFINED exceptions is optional and IMPLEMENTATION DEFINED. IMPLEMENTATION DEFINED exceptions are not required to be traced but are intended to be used to simplify tracing of certain micro-architectural situations.

I_{DFLDJ} In general, the preferred exception return address is the address of the instruction after the last executed instruction, before the exception occurs.

D2.2.3 Source Address Element

I_{DJTGL} The *Source Address element* indicates execution up to and including a provided *P0 instruction* address, and indicates the *P0 instruction* is taken. All *P0 instructions* except the final *P0 instruction* are not taken, which means that explicit N *Atom elements* are not required to be traced for those *P0 instructions*. A *Source Address element* indicates both of the following for the final *P0 instruction*:

- The instruction set.

• The virtual address of the instruction.

R_{HVVRK} The instruction set for a *Source Address element* is one of the following:

• AArch64 A64.
• AArch32 A32.
• AArch32 T32.

R_{WTRBB} A *Source Address element* is a *P0 element*.

D2.2.4 Q Element

R_{JRFYT} A *Q element* belongs to the *P0 element* group in the instruction trace element stream, and must be explicitly resolved or canceled.

I_{XPNWS} A *Q element* can optionally include a number, M. The number is a count of the instructions that are executed since the most recent *P0 element*, which might be a *Q element*. If it does not include a count of instructions, then the number of instructions that are executed since the most recent *P0 element* is UNKNOWN.

R_{XWBMW} The trace unit generates *Q elements* in the program order in which they occur, and the trace protocol encode and decode process maintains this order.

R_{JBYXC} A *Q element* does not imply *Exceptional occurrences*.

R_{KPNGG} When a *Q element* implies an Exception Return instruction which is taken, that instruction is the last instruction that is implied by the *Q element*.

R_{YRLJR} When a *Q element* implies an executed ISB instruction, this is the last instruction implied by the *Q element* if execution continues from a new context after the ISB.

R_{LZLDH} When execution continues from a new context after a *Q element* is generated, the trace unit generates a *Context element* after the *Q element*.

I_{BTNZC} The *Context element* might be generated before or after the *Target Address element* that is also required after the *Q element*.

If a context change occurs at a point that is not a Context synchronization event, then the last instruction that is implied by a *Q element* must be the last instruction that is executed with the old context. The trace unit can then generate a *Context element* after the *Q element* to indicate the new context.

D2.2.5 Transaction Start Element

R_{CTLXL} TRCIDR0.COMMTRANS indicates whether the *Transaction Start element* is a *P0 element*. See D2.7.1 *Transaction Start element* for more details about the *Transaction Start element*.

D2.3 Virtual Address Space Elements

D2.3.1 Trace On Element

R_{NHDCF} A *Trace On element* indicates a discontinuity in the trace element stream. The trace unit inserts a *Trace On element* after a gap in the generation of the trace element stream:

- When the trace generation becomes operative and before any *P0 elements*.
- If some instructions are filtered out of the trace.
- The first traced instructions after:
  - A Trace Prohibited region.
  - The PE leaves Debug state.
- When instruction trace is lost because a trace unit buffer overflow occurs.

R_{KMFKP} When a *Trace On element* is generated, the trace unit generates a *Target Address element* before the next *P0 element*.

R_{TJLYH} When a *Trace On element* is generated, the trace unit generates a *Context element* before the next *Atom element*, *Exception element* or *Q element*, to indicate where tracing starts, unless the context has not changed since the previous *Context element* was output.

R_{JKFBS} When the first *Trace On element* is generated, the trace unit outputs the corresponding *Context element* before the first *P0 element*.

D2.3.2 Target Address Element

R_{QWBLT} A *Target Address element* indicates both of the following for the next instruction to be executed:

- The instruction set.
- The virtual address of the instruction.

R_{JYKHH} The instruction set for a *Target Address element* is one of the following:

- AArch64 A64.
- AArch32 A32.
- AArch32 T32.

R_{HMWHY} The trace unit generates *Target Address elements* in program order relative to other *P0 elements*.

I_{XCKNM} *Target Address element* values can be corrected by another *Target Address element* if both *Target Address elements* are generated before the next *P0 element* or *Trace On element*.

D2.3.3 Context Element

I_{KQKFF} The *Context element* indicates the execution context for the next instruction to be executed.

R_{VHQYV} The *Context element* provides the following *Context information*:

- The Security state, either Secure or Non-secure.
- The Exception level, EL0 to EL3.
- Whether the PE is executing in AArch64 state or AArch32 state.

R_{WSVRL} The *Context element* can optionally provide the following *Context information*:

- The Context identifier.
- The Virtual context identifier.

R_{WJDWF} The trace unit generates *Context elements* in program order relative to *P0 elements*.

D2.4 Temporal Elements

I$_{\text{HHXND}}$ Temporal elements provide information about the passage of time within the trace element stream. The following temporal elements are supported by ETE:

The *Cycle Count element*. Indicates the passage of PE clock cycles within the trace element stream.

The *Timestamp element*. Indicates the passage of time within the trace element stream.

The *Timestamp Marker element*. Indicates the most recent *P0 element* or *Event element* has been timestamped, and that a *Timestamp element* will follow containing the timestamp value.

D2.4.1 Cycle Count Element

I$_{\text{NVGJP}}$ Each *Cycle Count element* is associated with a *Commit element*, and when a *Commit element* is generated, a *Cycle Count element* might also be generated.

R$_{\text{BZQWX}}$ Each *Cycle Count element* is associated with the most recent *Commit element*.

R$_{\text{VZXNN}}$ A *Cycle Count element* indicates the number of PE clock cycles between the two most recent *Commit elements* that both have an associated *Cycle Count element*.

I$_{\text{FHGKM}}$ Not every *Commit element* is required to have an associated *Cycle Count element*.

R$_{\text{VNYMN}}$ *Cycle Count elements* are generated in order relative to *Commit elements*.

D2.4.2 Timestamp Element

I$_{\text{LKDJM}}$ The *Timestamp element* inserts a global timestamp value into the trace element stream.

I$_{\text{BLBJX}}$ The source for timestamp reported in the timestamp element is controlled by:

- TRFCR_EL1.TS
- TRFCR_EL2.TS

R$_{\text{BRJJF}}$ A timestamp value of zero indicates that the timestamp value is UNKNOWN.

I$_{\text{VTLTF}}$ An UNKNOWN timestamp value might occur if the system does not support timestamping or if the timestamp is temporarily unavailable.

I$_{\text{YQJDR}}$ The source for the payload of *Timestamp elements* is controlled by the TRFCR registers and the virtual timers. It is expected that these registers will be changed by context switch software. As a result it is possible that payloads of *Timestamp elements* might appear to have discontinuities, and even go backwards, if the source of the timestamp changes, or any context switching changes the system registers which control the timestamp value.

R$_{\text{MCSGX}}$ If FEAT_ETEv1p1 is implemented, when there has been a *Timestamp Marker element* before the *Timestamp element*, the *Timestamp element* contains the timestamp value of the most recent *P0 element* or *Event element* before the *Timestamp Marker element*.

R$_{\text{DGTJZ}}$ If FEAT_ETEv1p1 is not implemented or if there has not been a *Timestamp Marker element* before the *Timestamp element*, the *Timestamp element* contains the timestamp value of the most recent *P0 element* or *Event element* before the *Timestamp element*.

I$_{\text{PXZVX}}$ If TRCIDR0.TSMARK is 0b1 and there is no previous *Timestamp Marker element*, the *Timestamp element* is for a *P0 element* or *Event element* which is before the start of the trace. This scenario might occur when trace analysis starts at a *Trace Info element* which is not the first *Trace Info element*, and the *Timestamp Marker element* was generated before the *Trace Info element*.

X_{CSZYW} The requirement for a *Timestamp Marker element* for every *Timestamp element* is to avoid needing to indicate if there's been a *Timestamp Marker element* at a Trace Info point. This allows a trace analyzer to assume there's one (or not) before the Trace Info, based on a static piece of information.

D2.4.3 Timestamp Marker element

R_{RFYPT} The *Timestamp Marker element* indicates the most recent *P0 element* or *Event element* has been timestamped, and that a *Timestamp element* will follow containing the timestamp value.

R_{SZRHP} *Timestamp Marker element*s are generated in order with respect to *P0 element*s and *Event element*s.

R_{DCRVK} *Timestamp Marker element*s are not canceled by *Cancel elements*.

I_{DLCLX} A *Cancel element* might cause a *P0 element* to be canceled and if there is a *Timestamp Marker element* that is associated with that *P0 element* then the *Timestamp Marker element* is not associated with any *P0 element*. The *Timestamp element* which is associated with the *Timestamp Marker element* is unaffected, and is still useable for timestamping the approximate position in the trace stream.

R_{VWJVC} If 2 *Timestamp Marker element*s occur without a *Timestamp element* between them, the oldest *Timestamp Marker element* is ignored.

R_{JNWJY} If an *Overflow element* or *Discard element* occurs after a *Timestamp Marker element* and before a *Timestamp element*, the *Timestamp Marker element* is ignored.

R_{LWZXK} If *Timestamp Marker element*s are generated by the trace unit, every *Timestamp element* must have a corresponding *Timestamp Marker element* generated before the *Timestamp element*.

I_{JGKZJ} The generation of *Timestamp Marker element*s is indicated in TRCIDR0.TSMARK.

D2.5 Speculation Resolution Elements

I_{YYMXT} The ETE architecture allows trace to be generated speculatively and then later committed or removed by the decompression process. Each *P0 element* is traced and is is considered speculative until either committed by a *Commit element* or canceled by a *Cancel element*. This method of generating speculative trace allows for the tracing of speculative execution, including the tracing of transactions when the Transactional Memory Extension is implemented in the PE.

I_{SRRZZ} Speculation resolution elements provide a trace analyzer with information about which trace elements were correctly or incorrectly generated, and ensure the trace analyzer can reconstruct the program execution. The following speculation resolution elements are supported by ETE:

The *Mispredict element*. Corrects the most recent *Atom element*.

The *Cancel element*. Indicates that one or more *P0 elements* are canceled.

The *Commit element*. Indicates that one or more *P0 elements* are resolved for execution.

The *Discard element*. Removes all speculative *P0 elements*.

I_{XLHWT} TRCIDR8.MAXSPEC specifies the maximum number of uncommitted *P0 elements* which can be discarded at a later stage using *Cancel elements*.

D2.5.1 Commit Element

I_{KQQML} A *Commit element* indicates that a number of unresolved *P0 elements* have been resolved for execution. The resolved *P0 elements* are the oldest *P0 elements*.

R_{PNBGQ} The *Commit element* resolves all types of *P0 element*.

I_{KHYLN} *Commit elements* might be merged if the total number of *P0 elements* resolved is less than TRCIDR8.MAXSPEC. *Commit elements* are merged by adding their respective commit count values together.
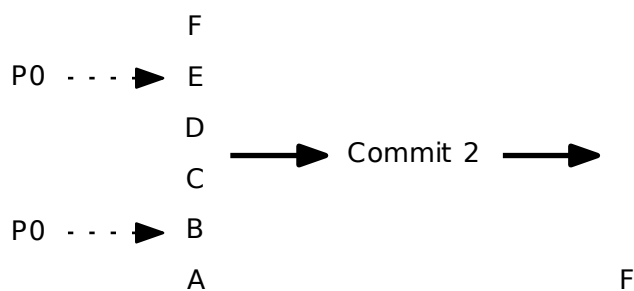
Figure D2.1: Commit Operation Example

D2.5.2 Cancel Element

I_{MRGLC}   The *Cancel element* indicates the number of youngest unresolved and un-canceled *P0 elements* that are canceled from execution. A trace unit might cancel elements because of many reasons, including but not limited to:

- A *P0 instruction* is mis-speculated.

- An exception occurs.

R_{WLTNX}   The *Cancel element* cancels all types of *P0 element*.

I_{NDQKN}   *Cancel elements* might be merged if no *P0 elements* have been generated in between. *Cancel elements* are merged by adding their respective cancel numbers together.
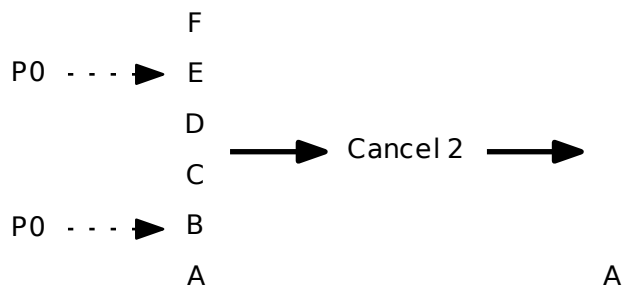
Figure D2.2: Cancel Operation Example

D2.5.3 Discard Element

I_{TCWCN}   A *Discard element* is generated if uncommitted *P0 elements* remain when trace generation becomes inoperative or if the resolution of uncommitted *P0 elements* cannot be output by the trace unit.

I_{STXQZ}   If trace generation remains inoperative, the outcomes of instructions that are traced by *P0 elements*, such as conditional *P0 instructions*, cannot be resolved, and therefore a *Discard element* indicates that all uncommitted *P0 elements* must be discarded.

D2.5.4 Mispredict Element

I_{GBKKQ}   The *Mispredict element* indicates that the most recent non-canceled *Atom element* has the incorrect E or N status.

I_{RGVGL}   For example, if a branch instruction is predicted as taken, it is traced with an E *Atom element*. If the prediction becomes incorrect then a *Mispredict element* is traced to indicate to a trace analyzer that the E *Atom element* changes to an N *Atom element*.

D2.6 Others

D2.6.1 Event Element

I_{RBKYZ} The *Event element* indicates when a programmed ETEEvent occurs and its payload contains a number to identify the ETEEvent number. See TRCEVENTCTL0R, and TRCEVENTCTL1R, for information about the programming of arbitrary ETEEvents.

R_{SMLVB} *Event elements* maintain order relative to other *Event elements*.

D2.6.2 Overflow Element

I_{RFQKZ} The *Overflow element* indicates that the trace unit buffer has overflowed, and at least one trace element might have been lost.

D2.7 Transactional Memory

R_{LLCQG}   The TSTART instruction is a *P0 instruction*.

D2.7.1 Transaction Start element

I_{QNFVH}   The *Transaction Start element* indicates that subsequent elements are within a transaction, until any of the following are traced:

- A *Transaction Failure element*.
- A *Transaction Commit element*.
- A *Cancel element* which cancels the *Transaction Start element*.

R_{KMNXW}   When the PE enters Transactional state, a *Transaction Start element* is generated before any instructions are traced. This indicates to the trace analyzer that subsequent elements have been executed in Transactional state.

R_{MMCQD}   Only a single *Transaction Start element* is generated for each outer transaction, unless the trace unit indicated the transaction had finished by generating a *Transaction Failure element*.

I_{MQZZY}   An example of when the trace unit generates a *Transaction Failure element* without the PE leaving Transactional state is when a trace unit buffer overflow occurs. In this example, tracing might resume after the trace unit buffer overflow, and if the PE is still in the same outer transaction then a new *Transaction Start element* would be generated.

R_{DPNGP}   The *Transaction Start element* appears in program order relative to other *P0 elements*.

R_{CYHKB}   When a TSTART instruction for an outer transaction is traced and tracing continues in Transactional state, the trace unit generates a *Transaction Start element* after the *P0 element* that is generated by the TSTART instruction, and before any subsequent *P0 element*.

R_{RKGLY}   When a TSTART instruction for an outer transaction is not traced and tracing becomes active while the PE is in Transactional state, the trace unit generates a *Transaction Start element* after the *Trace On element* and before any *P0 elements*.

D2.7.2 Transaction Commit element

I_{XTXHN}   The *Transaction Commit element* indicates that the PE has exited Transactional state, that the transaction has completed successfully, and that all execution since the most recent *Transaction Start element* has been executed.

D2.7.3 Transaction Failure element

I_{XHLPG}   The *Transaction Failure element* indicates that the transaction did not complete successfully and the trace analyzer discards all the execution since the most recent *Transaction Start element*, including any *P0 elements* which have been committed by *Commit elements*.

I_{HLQGS}   A sophisticated trace analyzer might be able to use the discarded elements to create a heuristic on why the transaction failed.

Chapter D3
Instruction and Exception classifications

I_{NMJBZ} This chapter defines all of the *P0 instructions*.

R_{PBVZM} *P0 instructions* comprise all of the following:

- All direct *P0 instructions*.
- All indirect *P0 instructions*.

R_{GFNRJ} Direct *P0 instructions* comprise all of the following:

- All direct branch instructions.
- `ISB` instructions.
- `TSTART` instructions.
- `WFE`, `WFET`, `WFI`, and `WFIT` instructions, when indicated by TRCIDR2.WFXMODE.

R_{DJMQM} Indirect *P0 instructions* comprise all of the following:

- All indirect branch instructions.

R_{KJTCL} All uses of `ISB` in this specification apply to all variants of the `ISB` instruction, including the CP15ISB instruction.

D3.1 AArch64 A64

D3.1.1 Direct P0 instructions

Table D3.1: A64 direct P0 instructions

Instruction	Description
B	Unconditional Branch.
B.cond	Conditional Branch.
BC.cond	Branch Consistent conditionally.
BL	Branch with link.
CBZ or CBNZ	Compare with zero and branch.
ISB	Instruction Synchronization Barrier.
TBZ or TBNZ	Test and branch.
TSTART	Initiates a new transaction.
WFE, WFET	Wait For Event
WFI, WFIT	Wait For Interrupt

D3.1.2 Indirect P0 instructions

Table D3.2: A64 indirect P0 instructions

Instruction	Description
BLR	Branch with link to register.
BLRAA	Authenticate and branch with link.
BLRAAZ	Authenticate and branch with link.
BLRAB	Authenticate and branch with link.
BLRABZ	Authenticate and branch with link.
BR	Branch to register.
BRAA	Authenticate and branch.
BRAAZ	Authenticate and branch.
BRAB	Authenticate and branch.
BRABZ	Authenticate and branch.
ERET	Return From Exception.
ERETAA	Authenticate and Exception return.
ERETAB	Authenticate and Exception return.
RET	Return From subroutine.

Instruction	Description
RETAA	Authenticate and function return.
RETAB	Authenticate and function return.

D3.1.3 Branch with link instructions

Table D3.3: A64 branch with link instructions

Instruction	Description
BL	Branch with link.
BLR	Branch with link to register.
BLRAA	Authenticate and branch with link.
BLRAAZ	Authenticate and branch with link.
BLRAB	Authenticate and branch with link.
BLRABZ	Authenticate and branch with link.

D3.1.4 Meaning of Atom elements

Table D3.4: Meaning of Atom elements in AArch64 A64

Instruction	E	N
B	The branch was taken.	RESERVED.
B.cond	The branch was taken.	The branch was not taken.
BC.cond	The branch was taken.	The branch was not taken.
BL	The branch was taken.	RESERVED.
BLR	The branch was taken.	RESERVED.
BLRAA	The branch was taken.	RESERVED.
BLRAAZ	The branch was taken.	RESERVED.
BLRAB	The branch was taken.	RESERVED.
BLRABZ	The branch was taken.	RESERVED.
BR	The branch was taken.	RESERVED.
BRAA	The branch was taken.	RESERVED.
BRAAZ	The branch was taken.	RESERVED.
BRAB	The branch was taken.	RESERVED.
BRABZ	The branch was taken.	RESERVED.
CBZ or CBNZ	The branch was taken.	The branch was not taken.

Instruction	E	N
ERET	The branch was taken and the PE returned from the Exception.	RESERVED.
ERETAA	The branch was taken and the PE returned from Exception.	RESERVED.
ERETAB	The branch was taken and the PE returned from Exception.	RESERVED.
ISB	The ISB performed a Context synchronization event and is considered as taken.	RESERVED.
RET	The branch was taken and the PE returned from the subroutine.	RESERVED.
RETAA	The branch was taken and the PE returned from the subroutine.	RESERVED.
RETAB	The branch was taken and the PE returned from the subroutine.	RESERVED.
TBZ or TBNZ	The branch was taken.	The branch was not taken.
TSTART	Transaction started and the instruction is considered as taken.	RESERVED.
WFE, WFET	The instruction was executed and is considered as taken.	RESERVED.
WFI, WFIT	The instruction was executed and is considered as taken.	RESERVED.

D3.2 AArch32 A32

D3.2.1 Direct P0 instructions

Table D3.5: A32 direct P0 instructions

Instruction	Description
B	Unconditional Branch.
B.cond	Conditional Branch.
BL	Branch with link
BLX <immed>	Branch with link and exchange.
ISB	Instruction Synchronization Barrier.
WFE	Wait For Event
WFI	Wait For Interrupt

D3.2.2 Indirect P0 instructions

Table D3.6: A32 indirect P0 instructions

Instruction	Description
BLX <reg>.	Branch with Link and Exchange.
BX	Branch and Exchange.
BXJ	Branch and Exchange.
Data processing instructions that modify the PC.	-
ERET	Exception Return.
LDM including the PC.	Load Multiple to the PC.
LDR PC	Load a word to the PC.
RFE	Return From Exception.

D3.2.3 Branch with link instructions

Table D3.7: A32 branch with link instructions

Instruction	Description
BL	Branch with link
BLX <immed>	Branch with link and exchange.
BLX <reg>.	Branch with Link and Exchange.

D3.2.4 Meaning of Atom elements

Table D3.8: Meaning of Atom elements in AArch32 A32

Instruction	E	N
B	The branch was taken.	The branch was not taken.
B.cond	The branch was taken.	The branch was not taken.
BL	The branch was taken.	The branch was not taken.
BLX <immed>	The branch was taken.	The branch was not taken.
BLX <reg>.	The branch was taken.	The branch was not taken.
BX	The branch was taken.	The branch was not taken.
BXJ	The branch was taken.	The branch was not taken.
Data processing instructions that modify the PC.	The branch was taken.	The branch was not taken.

Instruction	E	N
ERET	The branch was taken and the PE returned from an Exception.	The branch was not taken and the PE did not return from an Exception.
ISB	The ISB performed a Context synchronization event and is considered as taken	The ISB did not perform a Context synchronization event and is considered as not taken.
LDM including the PC.	The branch was taken.	The branch was not taken.
LDR PC	The branch was taken.	The branch was not taken.
RFE	The branch was taken and the PE returned from the Exception.	RESERVED.
WFE	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken.	The instruction failed its condition code check and is considered as not taken.
WFI	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken.	The instruction failed its condition code check and is considered as not taken.

D3.3 AArch32 T32

D3.3.1 Direct P0 instructions

Table D3.9: T32 direct P0 instructions

Instruction	Description
B	Unconditional Branch.
B<cc>	Conditional Branch.
BL	Branch with Link.
BLX <immed>	Branch with Link and Exchange.
CBNZ	Compare and Branch on Nonzero.
CBZ	Compare and Branch on Zero.
ISB	Instruction Synchronization Barrier, including CP15 encodings.
WFE	Wait For Event
WFI	Wait For Interrupt

D3.3.2 Indirect P0 instructions

Table D3.10: T32 indirect P0 instructions

Instruction	Description
BLX <reg>	Branch with Link and Exchange.
BX	Branch and Exchange.
BXJ	Branch and Exchange.
Data processing instructions that modify the PC.	-
LDM including the PC.	Load Multiple including to the PC.
LDR to the PC.	Load to the PC.
POP {..,PC}	Load the PC from the stack.
RFE	Return From Exception.
TBB	Table Branch.
TBH	Table Branch.

D3.3.3 Branch with link instructions

Table D3.11: T32 branch with link instructions

Instruction	Description
BL	Branch with Link.
BLX <immed>	Branch with Link and Exchange.
BLX <reg>	Branch with Link and Exchange.

D3.3.4 Meaning of Atom elements

Table D3.12: Meaning of Atom elements in AArch32 T32

Instruction	E	N
B	The branch was taken.	The branch was not taken.
B<cc>	The branch was taken.	The branch was not taken.
BL	The branch was taken.	The branch was not taken.
BLX <immed>	The branch was taken.	The branch was not taken.
BLX <reg>	The branch was taken.	The branch was not taken.
BX	The branch was taken.	The branch was not taken.
BXJ	The branch was taken.	The branch was not taken.

Instruction	E	N
CBNZ	The branch was taken.	The branch was not taken.
CBZ	The branch was taken.	The branch was not taken.
Data processing instructions that modify the PC.	The branch was taken.	The branch was not taken.
ISB	The ISB performed a Context synchronization event and is considered as taken.	The ISB did not perform a Context synchronization event and is considered as not taken.
LDM including the PC.	The branch was taken.	The branch was not taken.
LDR to the PC.	The branch was taken.	The branch was not taken.
POP {..,PC}	The branch was taken.	The branch was not taken.
RFE	The branch was taken and the PE returned from the Exception.	The branch was not taken and the PE did not return from the Exception.
TBB	The branch was taken.	The branch was not taken.
TBH	The branch was taken.	The branch was not taken.
WFE	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken.	The instruction failed its condition code check and is considered as not taken.
WFI	The instruction either passed its condition code check or failed its condition code check, but it is considered as taken.	The instruction failed its condition code check and is considered as not taken.

D3.4 WFI and WFE Instructions

WFE, WFET, WFI, and WFIT instructions, when indicated by TRCIDR2.WFXMODE, are *P0 instructions*.

D3.4.1 WFxT

R_{BBQHN} If FEAT_WFxT is implemented and TRCIDR2.WFXMODE is 0b1, WFE, WFET, WFI, and WFIT instructions are classified as direct branch instructions.

D3.5 Exceptions to Exception element encodings

Table D3.13: Exception mapping for exceptions taken to AArch64 state

Reason	Type
Branch Target exception	Inst fault
Breakpoint	Inst debug
FIQ	FIQ
HVC	Call
Halting debug event	Debug halt
IRQ	IRQ
Illegal execution state	Trap
Instruction Abort	Inst fault
Instruction or event trapped by a control bit	Trap
MemCopy or MemSet	Trap
Misaligned PC	Alignment
PAC Fail	Data fault
SError interrupt	System Error
SMC	Call
SVC	Call
Software Breakpoint Instruction	Inst debug
Software Step	Inst debug
Stack Pointer Misalignment	Alignment
Synchronous Data Abort	Data fault
UNDEFINED instruction	Trap
Watchpoint	Data debug

Table D3.14: Exception mapping for exceptions taken to AArch32 state

Reason	Type
Breakpoint	Inst fault
FIQ	FIQ
HVC	Call
Halting debug event	Debug halt
IRQ	IRQ
Illegal execution state	Trap

Reason	Type
Instruction or event trapped by a control bit	Trap
Prefetch Abort	Inst fault
SError interrupt	System Error
SMC	Call
SVC	Call
Software Breakpoint Instruction	Inst fault
Synchronous Data Abort	Data fault
UNDEFINED instruction	Trap
Vector Catch exception	Inst fault
Watchpoint	Data fault

Chapter D4
Recommended Configurations

D4.1 Configurations

I_{YCKVP} This section describes which ETE features Arm recommends are implemented. For optional features not described here, it is IMPLEMENTATION DEFINED whether the feature is implemented. For features which have an IMPLEMENTATION DEFINED size or number, and are not described here, the size or number of that feature is IMPLEMENTATION DEFINED.

Parameter	Description	Configuration
ATBTRIG	ATB Trigger Support	Yes, if ATB is implemented
NUMACPAIRS	Address Comparator pairs	4
NUMCIDC	Context Identifier Comparators	>= 1
NUMVMIDC	Virtual Context Identifier Comparators	>= 1, if EL2 is implemented
NUMCNTR	Number of Counters	2
NUMEVENT	Number of ETEEvents	4
NUMEXTINSEL	Number of External Input Selectors	4
NUMRSPAIR	Number of Resource selection pairs	>= 8
NUMSEQSTATE	Number of Sequencer states	4
NUMSSCC	Number of Single-shot Comparator Controls	>= 1
RETSTACK	Return stack	Yes
STALLCTL	*Processing Element* (PE) stalling capability	Yes
TRACEIDSIZE	Trace ID size	7-bits, if ATB is implemented
CCITMIN	Cycle count minimum threshold	4
CCSIZE	Cycle counter size	>= 12
WFXMODE	WFI, WFIT, WFE, and WFET instruction classification	WFI, WFIT, WFE, and WFET instructions are classified as *P0 instructions*

Chapter D5
Protocol Description

D5.1 Introduction

I_{XCCWQ} An ETE trace unit generates a trace byte stream. The protocol is a byte-based packet protocol, which means that the trace byte stream is constructed of multiple packets, where each packet contains one or more bytes of data.

R_{BVTNX} A packet consists of a single header byte, followed by zero or more payload bytes.

D5.2 Summary

Header byte	Name	Purpose
`00000000`	Alignment Synchronization Packet	Identifies a packet boundary.
`00000000`	Discard Packet	Indicates a *Discard element*.
`00000000`	Overflow Packet	Indicates that a trace unit buffer overflow has occurred.
`00000001`	Trace Info Packet	Resets trace compression to a known architectural state.
`0000001x`	Timestamp Packet	Indicates a *Timestamp element*.
`00000100`	Trace On Packet	Indicates that there has been a discontinuity in the trace element stream.
`00000110`	PE Reset Packet	Indicates that a PE Reset has occurred.
`00000110`	Transaction Failure Packet	Indicates that a Transaction Failure has occurred.
`00000110`	Exception 32-bit Address IS0 with Context Packet	Indicates that an exception has occurred.
`00000110`	Exception 32-bit Address IS1 with Context Packet	Indicates that an exception has occurred.
`00000110`	Exception 64-bit Address IS0 with Context Packet	Indicates that an exception has occurred.
`00000110`	Exception 64-bit Address IS1 with Context Packet	Indicates that an exception has occurred.
`00000110`	Exception Exact Match Address Packet	Indicates that an exception has occurred.
`00000110`	Exception Short Address IS0 Packet	Indicates that an exception has occurred.
`00000110`	Exception Short Address IS1 Packet	Indicates that an exception has occurred.
`00000110`	Exception 32-bit Address IS0 Packet	Indicates that an exception has occurred.
`00000110`	Exception 32-bit Address IS1 Packet	Indicates that an exception has occurred.
`00000110`	Exception 64-bit Address IS0 Packet	Indicates that an exception has occurred.
`00000110`	Exception 64-bit Address IS1 Packet	Indicates that an exception has occurred.
`00001010`	Transaction Start Packet	Indicates that the PE has started to execute in Transactional state.
`00001011`	Transaction Commit Packet	Indicates that the PE has successfully finished an outer transaction and is leaving Transactional state.
`00001100`	Cycle Count Format 2_0 small commit Packet	Indicates a *Commit element* and a *Cycle Count element*.
`00001101`	Cycle Count Format 2_1 Packet	Indicates a *Cycle Count element*.
`00001101`	Cycle Count Format 2_0 large commit Packet	Indicates a *Commit element* and a *Cycle Count element*.
`00001110`	Cycle Count Format 1_1 with count Packet	Indicates a *Cycle Count element*.
`00001110`	Cycle Count Format 1_0 with count Packet	Indicates zero or one *Commit elements* followed by a *Cycle Count element*.
`00001111`	Cycle Count Format 1_1 unknown count Packet	Indicates a *Cycle Count element*.

Header byte	Name	Purpose
`00001111`	Cycle Count Format 1_0 unknown count Packet	Indicates zero or one *Commit elements* followed by a *Cycle Count element* with an UNKNOWN cycle count value.
`000100xx`	Cycle Count Format 3_1 Packet	Indicates a *Cycle Count element*.
`0001xxxx`	Cycle Count Format 3_0 Packet	Indicates a *Commit element* and a *Cycle Count element*.
`00101101`	Commit Packet	Indicates a *Commit element*.
`0010111x`	Cancel Format 1 Packet	Indicates a *Cancel element* optionally followed by a *Mispredict element*.
`001100xx`	Mispredict Packet	Indicates 0-2 E or N *Atom elements* followed by one *Mispredict element*.
`001101xx`	Cancel Format 2 Packet	Indicates zero or more E or N *Atom elements* followed by a *Cancel element* and a *Mispredict element*.
`00111xxx`	Cancel Format 3 Packet	Indicates zero or one E *Atom element* followed by a *Cancel element* with a payload of 2-5 and one *Mispredict element*.
`01110000`	Ignore Packet	To align packet boundary to memory boundary.
`0111xxxx`	Event Packet	Indicates 1-4 *Event elements*.
`10000000`	Context Same Packet	Indicates a *Context element*.
`10000001`	Context Packet	Indicates a *Context element*.
`10000010`	Target Address with Context 32-bit IS0 Packet	Indicates a *Target Address element* and a *Context element*.
`10000011`	Target Address with Context 32-bit IS1 Packet	Indicates a *Target Address element* and a *Context element*.
`10000101`	Target Address with Context 64-bit IS0 Packet	Indicates a *Target Address element* and a *Context element*.
`10000110`	Target Address with Context 64-bit IS1 Packet	Indicates a *Target Address element* and a *Context element*.
`10001000`	Timestamp Marker Packet	Indicates a *Timestamp Marker element*.
`100100xx`	Target Address Exact Match Packet	Indicates a *Target Address element*.
`10010101`	Target Address Short IS0 Packet	Indicates a *Target Address element*.
`10010110`	Target Address Short IS1 Packet	Indicates a *Target Address element*.
`10011010`	Target Address 32-bit IS0 Packet	Indicates a *Target Address element*.
`10011011`	Target Address 32-bit IS1 Packet	Indicates a *Target Address element*.
`10011101`	Target Address 64-bit IS0 Packet	Indicates a *Target Address element*.
`10011110`	Target Address 64-bit IS1 Packet	Indicates a *Target Address element*.
`101000xx`	Q with Exact match address Packet	Indicates that some instructions have executed with an address of the next instruction.
`10100101`	Q short address IS0 Packet	Indicates that some instructions have executed with an address of the next instruction.
`10100110`	Q short address IS1 Packet	Indicates that some instructions have executed with an address of the next instruction.
`10101010`	Q 32-bit address IS0 Packet	Indicates that some instructions have executed with an address of the next instruction.
`10101011`	Q 32-bit address IS1 Packet	Indicates that some instructions have executed with an address of the next instruction.

Header byte	Name	Purpose
`10101100`	Q with count Packet	Indicates that some instructions have executed.
`10101111`	Q Packet	Indicates that some instructions have executed, without a count of the number of instructions.
`101100xx`	Source Address Exact Match Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10110100`	Source Address Short IS0 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10110101`	Source Address Short IS1 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10110110`	Source Address 32-bit IS0 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10110111`	Source Address 32-bit IS1 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10111000`	Source Address 64-bit IS0 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`10111001`	Source Address 64-bit IS1 Packet	Indicates the source address of a *P0 instruction*, and that the instruction was taken.
`110101xx`	Atom Format 5.2 Packet	Indicates five *Atom elements*.
`110110xx`	Atom Format 2 Packet	Indicates two *Atom elements*.
`110111xx`	Atom Format 4 Packet	Indicates four *Atom elements*.
`11110101`	Atom Format 5.1 Packet	Indicates five *Atom elements*.
`1111011x`	Atom Format 1 Packet	Indicates one *Atom element*.
`11111xxx`	Atom Format 3 Packet	Indicates three *Atom elements*.
`11xxxxxx`	Atom Format 6 Packet	Indicates 3-23 E *Atom elements*, plus a subsequent E Atom or N *Atom element*.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

D5.3 Encoding Schemes

D5.3.1 Field encodings

I_{TGRRZ} **Bit Replacement** The packet outputs bits which update a piece of state. Bits output by the packet replace only those bits in the piece of state. Bits not output by the packet remain unchanged in the piece of state.

I_{NKPMZ} **Unsigned LE128n** The data is encoded as an unsigned number. The least significant bits of the number are output in the least significant bits of the packet. Bits not output by the packet are 0.

I_{WYBBG} **POD** The encoding is specific to the packet.

I_{QXHHT} **Unary code** The sequence for this variable is one of the following:

- A 0.
- A number of 1 followed by a 0.
- All 1 for the size of the variable, as defined by the packet.

For example the permitted values for a 4-bit variable are:

- 0.
- 10.
- 110.
- 1110.
- 1111.

D5.3.2 Instruction set encoding

R_{FXNDF} For any virtual instruction address, the instruction set is output as a combination of the following two pieces of information:

- The SF bit encoded in Context packets.
- The sub_isa encoded by the type of the following groups of packets:
  - Target Address packets.
  - Exception packets.
  - Q packets.
  - Source Address packets.

The sub_isa indicates either:

- IS0.
- IS1.

Table D5.2 indicates how the combination of the SF bit and sub_isa indicate the instruction set.

Table D5.2: Instruction set encodings

SF Bit	sub_isa	Instruction Set
0b0	IS0	AArch32 A32
0b0	IS1	AArch32 T32
0b1	IS0	AArch64 A64

I_{WKMNL} The sub_isa also indicates the alignment of the virtual instruction addresses. Table D5.3 indicates the alignment of each sub_isa.

Table D5.3: Virtual instruction address alignment

sub_isa	Alignment
IS0	Word-Aligned
IS1	Halfword-Aligned

I_{NSZMB} The following packets encode the sub_isa:

- Exception Short Address IS0 Packet.
- Exception Short Address IS1 Packet.
- Exception 32-bit Address IS0 Packet.
- Exception 32-bit Address IS1 Packet.
- Exception 64-bit Address IS0 Packet.
- Exception 64-bit Address IS1 Packet.
- Exception 32-bit Address IS0 with Context Packet.
- Exception 32-bit Address IS1 with Context Packet.
- Exception 64-bit Address IS0 with Context Packet.
- Exception 64-bit Address IS1 with Context Packet.
- Target Address Short IS0 Packet.
- Target Address Short IS1 Packet.
- Target Address 32-bit IS0 Packet.
- Target Address 32-bit IS1 Packet.
- Target Address 64-bit IS0 Packet.
- Target Address 64-bit IS1 Packet.
- Target Address with Context 32-bit IS0 Packet.
- Target Address with Context 32-bit IS1 Packet.
- Target Address with Context 64-bit IS0 Packet.
- Target Address with Context 64-bit IS1 Packet.
- Source Address Short IS0 Packet.
- Source Address Short IS1 Packet.
- Source Address 32-bit IS0 Packet.
- Source Address 32-bit IS1 Packet.
- Source Address 64-bit IS0 Packet.
- Source Address 64-bit IS1 Packet.
- Q short address IS0 Packet.
- Q short address IS1 Packet.
- Q 32-bit address IS0 Packet.
- Q 32-bit address IS1 Packet.

D5.4 Alignment Synchronization Packet

Purpose Identifies a packet boundary.

Configurations All.

This packet forms a unique bit and byte pattern. Searching for this pattern allows the trace analyzer to identify packet boundaries.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0

Figure D5.1: Alignment Synchronization Packet

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

R_{BXZZJ} Any byte that follows this unique sequence of bits is the header byte of a new packet.

R_{VRKLP} This packet must be output before the first Trace Info packet.

D5.5 Discard Packet

Purpose Indicates a *Discard element*.

Configurations All.

Indicates a *Discard element*.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1

Figure D5.2: Discard Packet

Element sequence

This packet encodes the following sequence:

1. *Discard element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

I_{RTFPP} This packet is used to discard any speculative trace that the trace analyzer might still be holding onto.

D5.6 Overflow Packet

Purpose Indicates that a trace unit buffer overflow has occurred.

Configurations All.

Indicates that a trace unit buffer overflow has occurred and data might have been lost.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	1

Figure D5.3: Overflow Packet

Element sequence

This packet encodes the following sequence:

1. *Overflow element*.
2. *Discard element*.

Additional information

For more information about the decoding of this packet see decode.

D5.7 Trace Info Packet

Purpose Resets trace compression to a known architectural state.

Configurations All.

The trace info packet resets the trace compression to a known state.

Any fields which are not output are treated as if the value is zero.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0

Figure D5.4: Trace Info Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1
(0)	T	(0) (0) (0) (0) (0)					CC

Figure D5.5: Trace Info Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	0
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			

Figure D5.6: Trace Info Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	1
(0)	T	(0) (0) (0) (0) (0)					CC
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			

Figure D5.7: Trace Info Packet (4)

Packet Layout - Variant 5

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0
C1	CYCT[6:0]						
(0) (0) (0)			CYCT[11:7]				

Figure D5.8: Trace Info Packet (5)

Packet Layout - Variant 6

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	1
(0)	T	(0) (0) (0) (0) (0)					CC
C1	CYCT[6:0]						
(0) (0) (0)			CYCT[11:7]				

Figure D5.9: Trace Info Packet (6)

Packet Layout - Variant 7

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	1	0	0
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			
C1	CYCT[6:0]						
(0) (0) (0)			CYCT[11:7]				

Figure D5.10: Trace Info Packet (7)

Packet Layout - Variant 8

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1
0	0	0	0	1	1	0	1
(0)	T	(0) (0) (0) (0) (0)					CC
C0	SPEC[6:0]						
C0	SPEC[13:7]						
C0	SPEC[20:14]						
C0	SPEC[27:21]						
(0) (0) (0) (0)				SPEC[31:28]			
C1	CYCT[6:0]						
(0) (0) (0)			CYCT[11:7]				

Figure D5.11: Trace Info Packet (8)

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

CC Cycle count enable indicator.

When this field is not output, it is treated as if it is zero.

The encoding for this field is POD.

`0b0`	Cycle counting is not enabled.
`0b1`	Cycle counting is enabled.

CYCT The cycle count threshold.

When this field is not output, it is treated as if it is zero.

The encoding for this field is unsigned LE128n.

SPEC The number of uncommitted *P0 elements* in the trace.

When this field is not output, it is treated as if it is zero.

The encoding for this field is unsigned LE128n.

T Transactional state indicator.

When this field is not output, it is treated as if it is zero.

The encoding for this field is POD.

`0b0`	The PE is not currently executing in Transactional state.
`0b1`	The PE is currently executing in Transactional state.

Element sequence

This packet encodes the following sequence:

1. *Trace Info element*.

Additional information

For more information about the decoding of this packet see decode.

D5.8 Trace On Packet

Purpose Indicates that there has been a discontinuity in the trace element stream.

Configurations All.

A Trace On packet indicates to a trace analyzer that the trace unit has generated a *Trace On element*.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0
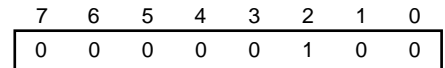
Figure D5.12: Trace On Packet

Element sequence

This packet encodes the following sequence:

1. *Trace On element*.

Additional information

For more information about the decoding of this packet see decode.

D5.9 Timestamp Packet

Purpose Indicates a *Timestamp element*.

Configurations TRCIDR0.TSSIZE != 0b00000.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0
C0	TS[6:0]						
C0	TS[13:7]						
C0	TS[20:14]						
C0	TS[27:21]						
C0	TS[34:28]						
C0	TS[41:35]						
C0	TS[48:42]						
C0	TS[55:49]						
TS[63:56]							

Figure D5.13: Timestamp Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1
C0	TS[6:0]						
C0	TS[13:7]						
C0	TS[20:14]						
C0	TS[27:21]						
C0	TS[34:28]						
C0	TS[41:35]						
C0	TS[48:42]						
C0	TS[55:49]						
TS[63:56]							
C1	COUNT[6:0]						
C1	COUNT[13:7]						
(0)	(0)	COUNT[19:14]					

Figure D5.14: Timestamp Packet (2)

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT The number of PE clock cycles between the most recent *Cycle Count element* and the element related to the Timestamp.

The encoding for this field is unsigned LE128n.

TS Timestamp Value.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Timestamp element*.

Additional information

For more information about the decoding of this packet see decode.

D5.10 Timestamp Marker Packet

Purpose Indicates a *Timestamp Marker element*.

Configurations TRCIDR0.TSSIZE != 0b00000 and TRCIDR0.TSMARK == 0b1

Packet Layout
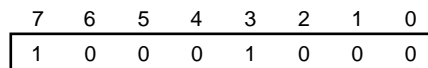
7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0

Figure D5.15: Timestamp Marker Packet

D5.11 Transaction Start Packet

Purpose Indicates that the PE has started to execute in Transactional state.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0

Figure D5.16: Transaction Start Packet

Element sequence

This packet encodes the following sequence:

1. *Transaction Start element*.

Additional information

For more information about the decoding of this packet see decode.

D5.12 Transaction Commit Packet

Purpose Indicates that the PE has successfully finished an outer transaction and is leaving Transactional state.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1
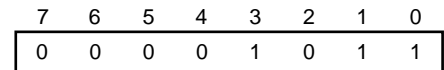
Figure D5.17: Transaction Commit Packet

Element sequence

This packet encodes the following sequence:

1. *Transaction Commit element*.

Additional information

For more information about the decoding of this packet see decode.

D5.13 Exception Exact Match Address Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	0	0	A	
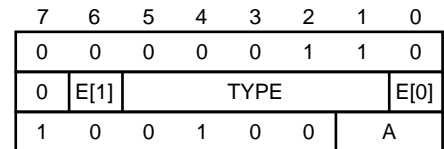
Figure D5.18: Exception Exact Match Address Packet

Field descriptions

A Preferred Exception Return address.

The encoding for this field is POD.

0b00	The Preferred Exception Return is the same as address history buffer entry 0.
0b01	The Preferred Exception Return is the same as address history buffer entry 1.
0b10	The Preferred Exception Return is the same as address history buffer entry 2.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

0b01	1. *Exception element* (TYPE, ADDRESS).
0b10	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE The exception type.

The encoding for this field is POD.

0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.

`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.
`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.14 Exception Short Address IS0 Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	0	1	0	1
C0	A[8:2]						
A[16:9]							

Figure D5.19: Exception Short Address IS0 Packet

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.
`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.15 Exception Short Address IS1 Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	0	1	1	0
C0	A[7:1]						
A[15:8]							
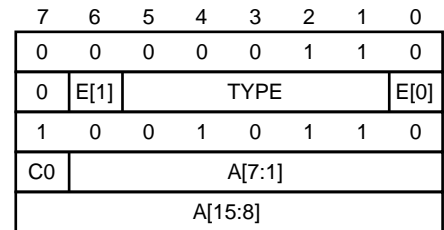
Figure D5.20: Exception Short Address IS1 Packet

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.
`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.16 Exception 32-bit Address IS0 Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]		TYPE				E[0]
1	0	0	1	1	0	1	0
(0)		A[8:2]					
(0)		A[15:9]					
A[23:16]							
A[31:24]							
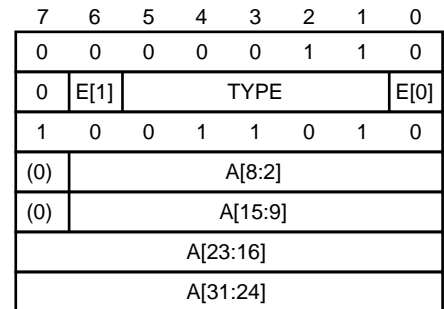
Figure D5.21: Exception 32-bit Address IS0 Packet

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.

`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.
`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.17 Exception 32-bit Address IS1 Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	1	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
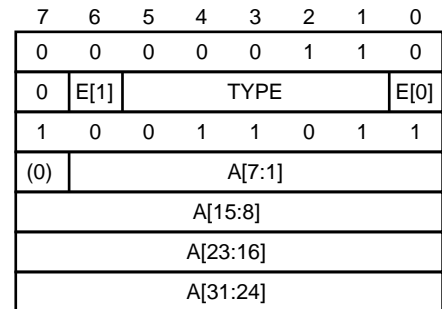
Figure D5.22: Exception 32-bit Address IS1 Packet

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.

0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.18 Exception 64-bit Address IS0 Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	1	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5.23: Exception 64-bit Address IS0 Packet

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE The exception type.

The encoding for this field is POD.

0b00000	PE Reset, also see PE Reset Packet.
0b00001	Debug halt.
0b00010	Call.
0b00011	Trap.
0b00100	System Error.
0b00110	Inst debug.
0b00111	Data debug.
0b01010	Alignment.
0b01011	Inst Fault.
0b01100	Data Fault.
0b01110	IRQ.
0b01111	FIQ.
0b10000	IMPLEMENTATION DEFINED 0.
0b10001	IMPLEMENTATION DEFINED 1.
0b10010	IMPLEMENTATION DEFINED 2.
0b10011	IMPLEMENTATION DEFINED 3.
0b10100	IMPLEMENTATION DEFINED 4.
0b10101	IMPLEMENTATION DEFINED 5.
0b10110	IMPLEMENTATION DEFINED 6.
0b10111	IMPLEMENTATION DEFINED 7.
0b11000	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.19 Exception 64-bit Address IS1 Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	1	1	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
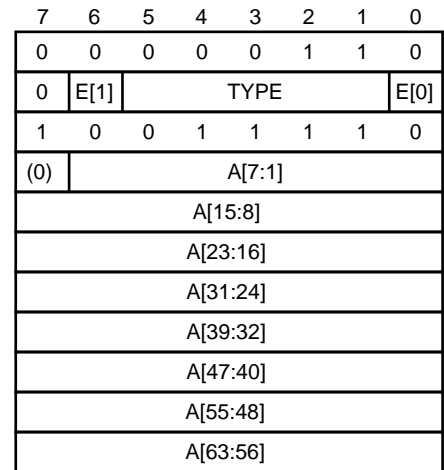
Figure D5.24: Exception 64-bit Address IS1 Packet

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`   1. *Exception element* (TYPE, ADDRESS).

`0b10`   1. *Target Address element* (ADDRESS).
                    2. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.
`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.20 Exception 32-bit Address IS0 with Context Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	0	NS	SF	(0)	(0)	EL	

Figure D5.25: Exception 32-bit Address IS0 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.26: Exception 32-bit Address IS0 with Context Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.27: Exception 32-bit Address IS0 with Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.28: Exception 32-bit Address IS0 with Context Packet (4)

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

NS Security state

When this field is not output, the Security state is the same as the most recently output Security state.

The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.
`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.21   Exception 32-bit Address IS1 with Context Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	0	NS	SF	(0)	(0)	EL	
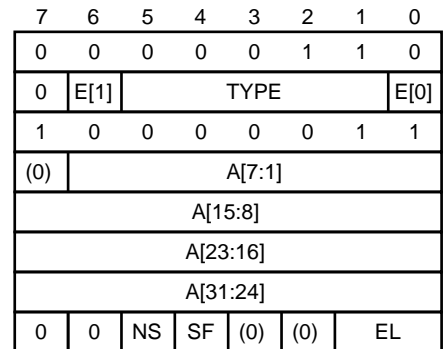
Figure D5.29: Exception 32-bit Address IS1 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							
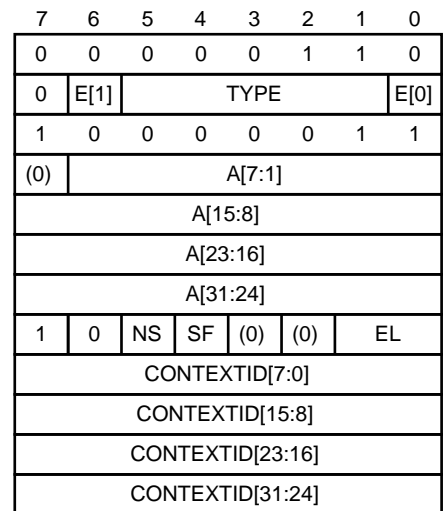
Figure D5.30: Exception 32-bit Address IS1 with Context Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
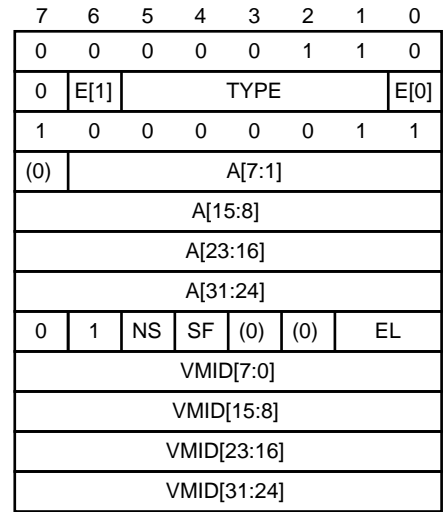
Figure D5.31: Exception 32-bit Address IS1 with Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.32: Exception 32-bit Address IS1 with Context Packet (4)

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

NS Security state.

The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.
`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.22 Exception 64-bit Address IS0 with Context Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	(0)	(0)	EL	

Figure D5.33: Exception 64-bit Address IS0 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]		TYPE				E[0]
1	0	0	0	0	1	0	1
(0)		A[8:2]					
(0)		A[15:9]					
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.34: Exception 64-bit Address IS0 with Context Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]		TYPE				E[0]
1	0	0	0	0	1	0	1
(0)		A[8:2]					
(0)		A[15:9]					
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.35: Exception 64-bit Address IS0 with Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.36: Exception 64-bit Address IS0 with Context Packet (4)

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

NS Security state

The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.

`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.23 Exception 64-bit Address IS1 with Context Packet

Purpose Indicates that an exception has occurred.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	(0)	(0)	EL	

Figure D5.37: Exception 64-bit Address IS1 with Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]		TYPE				E[0]
1	0	0	0	0	1	1	0
(0)		A[7:1]					
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.38: Exception 64-bit Address IS1 with Context Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]		TYPE				E[0]
1	0	0	0	0	1	1	0
(0)		A[7:1]					
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.39: Exception 64-bit Address IS1 with Context Packet (3)

Packet Layout - Variant 4

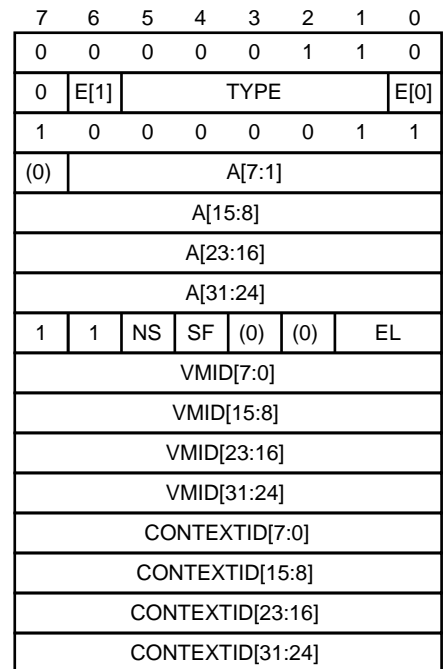| 7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	TYPE					E[0]
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							
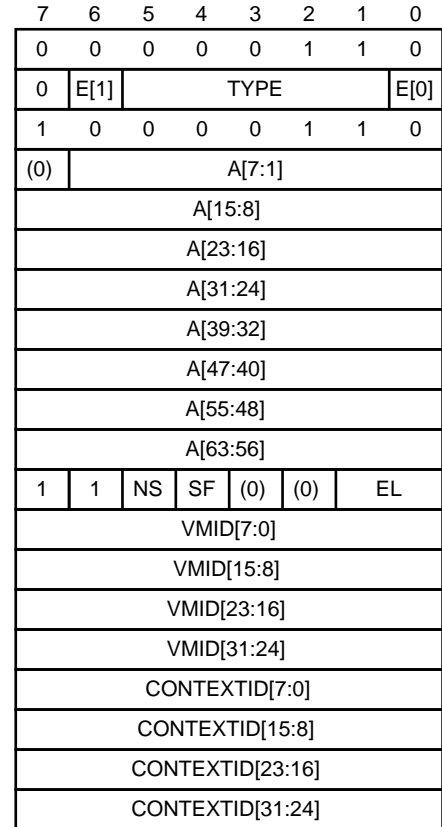
Figure D5.40: Exception 64-bit Address IS1 with Context Packet (4)

Field descriptions

A Preferred Exception Return address.

Preferred Exception Return address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Context element*. 2. *Exception element* (TYPE, ADDRESS).
`0b10`	1. *Target Address element* (ADDRESS). 2. *Context element*. 3. *Exception element* (TYPE, ADDRESS).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

EL Exception level at the Preferred Exception Return address.

The encoding for this field is POD.

`0b00`	EL0.
`0b01`	EL1.
`0b10`	EL2.
`0b11`	EL3.

NS Security state

The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

TYPE The exception type.

The encoding for this field is POD.

`0b00000`	PE Reset, also see PE Reset Packet.
`0b00001`	Debug halt.
`0b00010`	Call.
`0b00011`	Trap.
`0b00100`	System Error.
`0b00110`	Inst debug.

`0b00111`	Data debug.
`0b01010`	Alignment.
`0b01011`	Inst Fault.
`0b01100`	Data Fault.
`0b01110`	IRQ.
`0b01111`	FIQ.
`0b10000`	IMPLEMENTATION DEFINED 0.
`0b10001`	IMPLEMENTATION DEFINED 1.
`0b10010`	IMPLEMENTATION DEFINED 2.
`0b10011`	IMPLEMENTATION DEFINED 3.
`0b10100`	IMPLEMENTATION DEFINED 4.
`0b10101`	IMPLEMENTATION DEFINED 5.
`0b10110`	IMPLEMENTATION DEFINED 6.
`0b10111`	IMPLEMENTATION DEFINED 7.
`0b11000`	Reserved. See Transaction Failure Packet.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

VMID Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.24 Transaction Failure Packet

Purpose Indicates that a Transaction Failure has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	1	1	0	0	0	E[0]
0	1	1	1	0	0	0	0

Figure D5.41: Transaction Failure Packet

Field descriptions

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

`0b01`	1. *Transaction Failure element*.
`0b10`	1. *Target Address element* (UNKNOWN). 2. *Transaction Failure element*.

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.25 PE Reset Packet

Purpose Indicates that a PE Reset has occurred.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0
0	E[1]	0	0	0	0	0	E[0]
0	1	1	1	0	0	0	0

Figure D5.42: PE Reset Packet

Field descriptions

E Identifies the elements that are indicated by this packet.

The encoding for this field is POD.

0b01	1. *Exception element* (PE_Reset, UNKNOWN).
0b10	1. *Target Address element* (UNKNOWN). 2. *Exception element* (PE_Reset, UNKNOWN).

All other values are reserved. Reserved values might be defined in a future version of the architecture.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.26 Cycle Count Format 1_0 unknown count Packet

Purpose Indicates zero or one *Commit elements* followed by a *Cycle Count element* with an UNKNOWN cycle count value.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b0`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1
C0	COMMIT[6:0]						
C0	COMMIT[13:7]						
C0	COMMIT[20:14]						
C0	COMMIT[27:21]						
(0) (0) (0) (0)				COMMIT[31:28]			

Figure D5.43: Cycle Count Format 1_0 unknown count Packet

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COMMIT If this field is zero, there is no *Commit element*. Otherwise, there is a *Commit element* before the *Cycle Count element* and this field indicates the number of *P0 elements* committed by the *Commit element*.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element* with an unknown cycle count.

Additional information

For more information about the decoding of this packet see decode.

D5.27 Cycle Count Format 1_1 unknown count Packet

Purpose Indicates a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == 0b1.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1
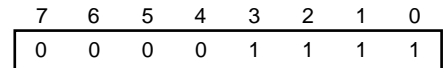
Figure D5.44: Cycle Count Format 1_1 unknown count Packet

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element* with an unknown cycle count.

Additional information

For more information about the decoding of this packet see decode.

D5.28 Cycle Count Format 1_0 with count Packet

Purpose Indicates zero or one *Commit elements* followed by a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0
C0	COMMIT[6:0]						
C0	COMMIT[13:7]						
C0	COMMIT[20:14]						
C0	COMMIT[27:21]						
(0) (0) (0) (0)				COMMIT[31:28]			
C1	COUNT[6:0]						
C1	COUNT[13:7]						
(0) (0)		COUNT[19:14]					

Figure D5.45: Cycle Count Format 1_0 with count Packet

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

0b0	Last byte in this section.
0b1	At least one more byte follows in this section.

C1 Continuation Bit.

The encoding for this field is Unary code.

0b0	Last byte in this section.
0b1	At least one more byte follows in this section.

COMMIT If this field is zero, there is no *Commit element*. Otherwise, there is a *Commit element* before the *Cycle Count element* and this field indicates the number of *P0 elements* committed by the *Commit element*.

The encoding for this field is unsigned LE128n.

COUNT Indicates the number of PE clock cycles that have occurred between the 2 most recent *Commit elements* that both had a *Cycle Count element* associated with them. The cycle count is COUNT+cc_threshold.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see decode.

D5.29 Cycle Count Format 1_1 with count Packet

Purpose Indicates a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == 0b1.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0
C0	COUNT[6:0]						
C0	COUNT[13:7]						
(0)	(0)	COUNT[19:14]					

Figure D5.46: Cycle Count Format 1_1 with count Packet

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

0b0	Last byte in this section.
0b1	At least one more byte follows in this section.

COUNT Indicates the number of PE clock cycles that have occurred between the 2 most recent *Commit elements* that both had a *Cycle Count element* associated with them. The cycle count is COUNT+cc_threshold.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see decode.

D5.30 Cycle Count Format 2_0 small commit Packet

Purpose Indicates a *Commit element* and a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b0`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0
AAAA				BBBB			

Figure D5.47: Cycle Count Format 2_0 small commit Packet

Field descriptions

AAAA Indicates the number of *P0 elements* to be resolved indicated by this field + 1.

The encoding for this field is POD.

BBBB Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see decode.

D5.31 Cycle Count Format 2_0 large commit Packet

Purpose Indicates a *Commit element* and a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1
AAAA				BBBB			

Figure D5.48: Cycle Count Format 2_0 large commit Packet

Field descriptions

AAAA Indicates the number of *P0 elements* to be resolved indicated by TRCIDR8.MAXSPEC + field - 15.

The number of P0 elements to be resolved must be greater than 0.

If the number of P0 elements to be resolved is less than 17 then it is preferred that a Cycle Count Format 2_0 small commit Packet is used.

The encoding for this field is POD.

BBBB Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see decode.

D5.32 Cycle Count Format 2_1 Packet

Purpose Indicates a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == 0b1.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1
1	1	1	1	BBBB			

Figure D5.49: Cycle Count Format 2_1 Packet

Field descriptions

BBBB Indicates the cycle value. The cycle count is calculated from cc_threshold + BBBB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see decode.

D5.33 Cycle Count Format 3_0 Packet

Purpose Indicates a *Commit element* and a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == 0b0.
- TRCIDR0.TRCCCI == 0b1.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	1	AA		BB	
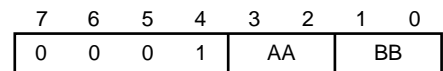
Figure D5.50: Cycle Count Format 3_0 Packet

Field descriptions

AA The number of *P0 elements* to be resolved indicated by this field + 1.

The encoding for this field is POD.

BB Indicates the cycle value. The cycle count is calculated from cc_threshold + BB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.
2. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see decode.

D5.34 Cycle Count Format 3_1 Packet

Purpose Indicates a *Cycle Count element*.

Configurations All the following conditions must be met:

- TRCIDR0.COMMOPT == `0b1`.
- TRCIDR0.TRCCCI == `0b1`.

Packet Layout

7	6	5	4	3	2	1	0
0	0	0	1	0	0	BB	
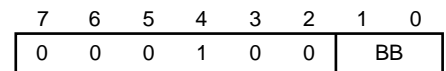
Figure D5.51: Cycle Count Format 3_1 Packet

Field descriptions

BB Indicates the cycle value. The cycle count is calculated from cc_threshold + BB.

The encoding for this field is POD.

Element sequence

This packet encodes the following sequence:

1. *Cycle Count element*.

Additional information

For more information about the decoding of this packet see decode.

D5.35 Commit Packet

Purpose Indicates a *Commit element*.

Configurations TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	1
C0	COMMIT[6:0]						
C0	COMMIT[13:7]						
C0	COMMIT[20:14]						
C0	COMMIT[27:21]						
(0) (0) (0) (0)				COMMIT[31:28]			

Figure D5.52: Commit Packet

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

0b0	Last byte in this section.
0b1	At least one more byte follows in this section.

COMMIT The number of *P0 elements* to be resolved.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Commit element*.

Additional information

For more information about the decoding of this packet see decode.

D5.36 Cancel Format 1 Packet

Purpose Indicates a *Cancel element* optionally followed by a *Mispredict element*.

Configurations TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	0	1	1	1	M
C0	CANCEL[6:0]						
C0	CANCEL[13:7]						
C0	CANCEL[20:14]						
C0	CANCEL[27:21]						
(0) (0) (0) (0)				CANCEL[31:28]			

Figure D5.53: Cancel Format 1 Packet

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

CANCEL The number of *P0 elements* to be canceled.

The encoding for this field is unsigned LE128n.

`0b0`	Reserved

M Mispredict element included in the packet.

The encoding for this field is POD.

`0b0`	No *Mispredict element* occurred
`0b1`	A *Mispredict element* occurred after the *Cancel element*

Additional information

For more information about the decoding of this packet see decode.

D5.37 Cancel Format 2 Packet

Purpose Indicates zero or more E or N *Atom elements* followed by a *Cancel element* and a *Mispredict element*.

Configurations TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	1	0	1	A	
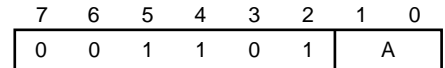
Figure D5.54: Cancel Format 2 Packet

Field descriptions

A Indicates the number of *Atom elements* that occurred before the *Cancel element*.

The encoding for this field is POD.

0b00
1. *Cancel element.*
2. *Mispredict element.*

0b01
1. E *Atom element.*
2. *Cancel element.*
3. *Mispredict element.*

0b10
1. E *Atom element.*
2. E *Atom element.*
3. *Cancel element.*
4. *Mispredict element.*

0b11
1. N *Atom element.*
2. *Cancel element.*
3. *Mispredict element.*

Additional information

For more information about the decoding of this packet see decode.

D5.38 Cancel Format 3 Packet

Purpose Indicates zero or one E *Atom element* followed by a *Cancel element* with a payload of 2-5 and one *Mispredict element*.

Configurations TRCIDR8.MAXSPEC > 0.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	1	1	CC		A
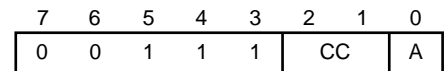
Figure D5.55: Cancel Format 3 Packet

Field descriptions

A Indicates the number of *Atom elements* that occurred before the *Cancel element*.

The encoding for this field is POD.

`0b0`	1. *Cancel element*.
`0b1`	1. E *Atom element*. 2. *Cancel element*.

CC The number of *P0 elements* to be canceled.

The encoding for this field is POD.

`0b00`	Cancel 2 *P0 elements*
`0b01`	Cancel 3 *P0 elements*
`0b10`	Cancel 4 *P0 elements*
`0b11`	Cancel 5 *P0 elements*

Additional information

For more information about the decoding of this packet see decode.

D5.39 Mispredict Packet

Purpose Indicates 0-2 E or N *Atom elements* followed by one *Mispredict element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	0	1	1	0	0	A	
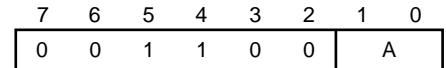
Figure D5.56: Mispredict Packet

Field descriptions

A Indicates the number of *Atom elements* that occurred before the *Mispredict element*.

   The encoding for this field is POD.

`0b00`
   1. *Mispredict element*.

`0b01`
   1. E *Atom element*.
   2. *Mispredict element*.

`0b10`
   1. E *Atom element*.
   2. E *Atom element*.
   3. *Mispredict element*.

`0b11`
   1. N *Atom element*.
   2. *Mispredict element*.

Additional information

For more information about the decoding of this packet see decode.

D5.40 Atom Format 1 Packet

Purpose Indicates one *Atom element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	A
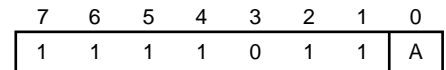
Figure D5.57: Atom Format 1 Packet

Field descriptions

A Indicates a single *Atom element*.

The encoding for this field is POD.

`0b0`	One N *Atom element*
`0b1`	One E *Atom element*

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.41 Atom Format 2 Packet

Purpose Indicates two *Atom elements*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	0	1	1	0	A	
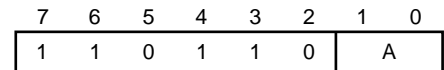
Figure D5.58: Atom Format 2 Packet

Field descriptions

A Indicates a specific sequence of *Atom elements*.

The encoding for this field is POD.

`0b00`	1. N *Atom element*. 2. N *Atom element*.
`0b01`	1. E *Atom element*. 2. N *Atom element*.
`0b10`	1. N *Atom element*. 2. E *Atom element*.
`0b11`	1. E *Atom element*. 2. E *Atom element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.42 Atom Format 3 Packet

Purpose Indicates three *Atom elements*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	1	1	1	A		
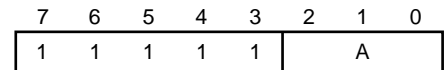
Figure D5.59: Atom Format 3 Packet

Field descriptions

A Indicates a specific sequence of *Atom elements*.

   The encoding for this field is POD.

   `0b000`
   1. N *Atom element*.
   2. N *Atom element*.
   3. N *Atom element*.

   `0b001`
   1. E *Atom element*.
   2. N *Atom element*.
   3. N *Atom element*.

   `0b010`
   1. N *Atom element*.
   2. E *Atom element*.
   3. N *Atom element*.

   `0b011`
   1. E *Atom element*.
   2. E *Atom element*.
   3. N *Atom element*.

   `0b100`
   1. N *Atom element*.
   2. N *Atom element*.
   3. E *Atom element*.

   `0b101`
   1. E *Atom element*.
   2. N *Atom element*.
   3. E *Atom element*.

0b110

1. N *Atom element*.
2. E *Atom element*.
3. E *Atom element*.

0b111

1. E *Atom element*.
2. E *Atom element*.
3. E *Atom element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.43 Atom Format 4 Packet

Purpose Indicates four *Atom elements*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	0	1	1	1	A	A
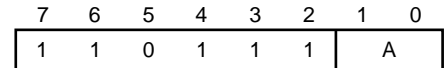
Figure D5.60: Atom Format 4 Packet

Field descriptions

A Indicates a specific sequence of *Atom elements*.

The encoding for this field is POD.

0b00
1. N *Atom element*.
2. E *Atom element*.
3. E *Atom element*.
4. E *Atom element*.

0b01
1. N *Atom element*.
2. N *Atom element*.
3. N *Atom element*.
4. N *Atom element*.

0b10
1. N *Atom element*.
2. E *Atom element*.
3. N *Atom element*.
4. E *Atom element*.

0b11
1. E *Atom element*.
2. N *Atom element*.
3. E *Atom element*.
4. N *Atom element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.44 Atom Format 5.1 Packet

Purpose Indicates five *Atom elements*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	1
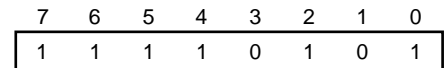
Figure D5.61: Atom Format 5.1 Packet

Element sequence

This packet encodes the following sequence:

1. N *Atom element*.
2. E *Atom element*.
3. E *Atom element*.
4. E *Atom element*.
5. E *Atom element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.45 Atom Format 5.2 Packet

Purpose Indicates five *Atom elements*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	0	1	0	1	A	
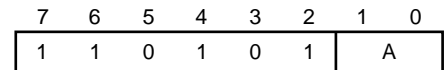
Figure D5.62: Atom Format 5.2 Packet

Field descriptions

A Indicates a specific sequence of *Atom elements*.

The encoding for this field is POD.

0b01
1. N *Atom element*.
2. N *Atom element*.
3. N *Atom element*.
4. N *Atom element*.
5. N *Atom element*.

0b10
1. N *Atom element*.
2. E *Atom element*.
3. N *Atom element*.
4. E *Atom element*.
5. N *Atom element*.

0b11
1. E *Atom element*.
2. N *Atom element*.
3. E *Atom element*.
4. N *Atom element*.
5. E *Atom element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.46 Atom Format 6 Packet

Purpose Indicates 3-23 E *Atom elements*, plus a subsequent E Atom or N *Atom element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	1	A	COUNT				
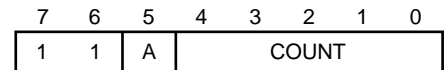
Figure D5.63: Atom Format 6 Packet

Field descriptions

A Indicates an E *Atom element* or N *Atom element*, after the E *Atom elements* indicated by COUNT.

The encoding for this field is POD.

0b0	One E *Atom element*
0b1	One N *Atom element*

COUNT Indicates a number of E *Atom elements*. The number is 3 + COUNT. Permitted values of COUNT are `0b00000` to `0b10100`.

The encoding for this field is POD.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.47 Target Address Short IS0 Packet

Purpose Indicates a *Target Address element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	0	1	0	1
C0	A[8:2]						
A[16:9]							
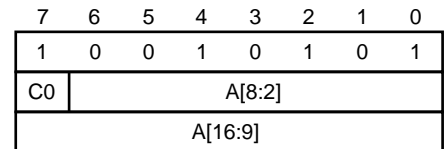
Figure D5.64: Target Address Short IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.48 Target Address Short IS1 Packet

Purpose Indicates a *Target Address element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	0	1	1	0
C0	A[7:1]						
A[15:8]							
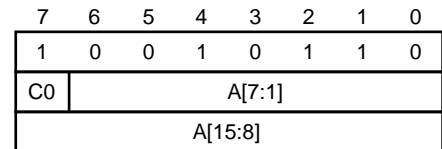
Figure D5.65: Target Address Short IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.49 Target Address 32-bit IS0 Packet

Purpose Indicates a *Target Address element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							

Figure D5.66: Target Address 32-bit IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.50 Target Address 32-bit IS1 Packet

Purpose Indicates a *Target Address element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
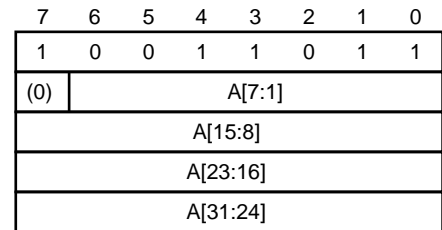
Figure D5.67: Target Address 32-bit IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.51 Target Address 64-bit IS0 Packet

Purpose Indicates a *Target Address element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5.68: Target Address 64-bit IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.52 Target Address 64-bit IS1 Packet

Purpose Indicates a *Target Address element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5.69: Target Address 64-bit IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.53 Target Address Exact Match Packet

Purpose Indicates a *Target Address element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	1	0	0	QE	
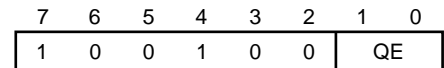
Figure D5.70: Target Address Exact Match Packet

Field descriptions

QE Instruction virtual address.

The encoding for this field is POD.

0b00	The address is the same as address history buffer entry 0.
0b01	The address is the same as address history buffer entry 1.
0b10	The address is the same as address history buffer entry 2.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.54 Context Same Packet

Purpose Indicates a *Context element*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0
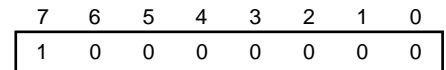
Figure D5.71: Context Same Packet

Element sequence

This packet encodes the following sequence:

1. *Context element*.

Additional information

For more information about the decoding of this packet see decode.

D5.55 Context Packet

Purpose Indicates a *Context element*.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
0	0	NS	SF	(0)	(0)	EL	

Figure D5.72: Context Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.73: Context Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.74: Context Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	1
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.75: Context Packet (4)

Field descriptions

CONTEXTID Context identifier.

> When this field is not output, the Context identifier is the same as the most recently output Context identifier.

> If Context identifier tracing is disabled, then one of the following must occur:

> - This field is not traced.
> - This field contains a value of zero.

> The encoding for this field is POD.

> See Context identifier tracing.

EL Exception level.

> The encoding for this field is POD.

`0b00`	EL0
`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS Security state.

> The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Context element*.

Additional information

For more information about the decoding of this packet see decode.

D5.56 Target Address with Context 32-bit IS0 Packet

Purpose Indicates a *Target Address element* and a *Context element*.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	0	NS	SF	(0)	(0)	EL	

Figure D5.76: Target Address with Context 32-bit IS0 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.77: Target Address with Context 32-bit IS0 Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.78: Target Address with Context 32-bit IS0 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.79: Target Address with Context 32-bit IS0 Packet (4)

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL Exception level at this address.

The encoding for this field is POD.

`0b00`	EL0
`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS Security state.

The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID Virtual context identifier.

When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

2. *Context element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.57 Target Address with Context 32-bit IS1 Packet

Purpose Indicates a *Target Address element* and a *Context element*.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	0	NS	SF	(0)	(0)	EL	

Figure D5.80: Target Address with Context 32-bit IS1 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.81: Target Address with Context 32-bit IS1 Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.82: Target Address with Context 32-bit IS1 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.83: Target Address with Context 32-bit IS1 Packet (4)

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

> When this field is not output, the Context identifier is the same as the most recently output Context identifier.
>
> If Context identifier tracing is disabled, then one of the following must occur:
>
> - This field is not traced.
> - This field contains a value of zero.
>
> The encoding for this field is POD.
>
> See Context identifier tracing.

EL Exception level at this address.

> The encoding for this field is POD.

`0b00`	EL0
`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS Security state.

> The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

> The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID Virtual context identifier.

> When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.
>
> If Virtual context identifier tracing is disabled, then one of the following must occur:
>
> - This field is not traced.
> - This field contains a value of zero.
>
> The encoding for this field is POD.
>
> See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.

2. *Context element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.58 Target Address with Context 64-bit IS0 Packet

Purpose Indicates a *Target Address element* and a *Context element*.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	(0)	(0)	EL	

Figure D5.84: Target Address with Context 64-bit IS0 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.85: Target Address with Context 64-bit IS0 Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.86: Target Address with Context 64-bit IS0 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	1	0	1
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.87: Target Address with Context 64-bit IS0 Packet (4)

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL Exception level at this address.

The encoding for this field is POD.

`0b00`	EL0

`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS Security state.

    The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

    The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID Virtual context identifier.

    When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

    If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

    The encoding for this field is POD.

    See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.
2. *Context element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.59 Target Address with Context 64-bit IS1 Packet

Purpose Indicates a *Target Address element* and a *Context element*.

Configurations All.

Packet Layout - Variant 1

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	0	NS	SF	(0)	(0)	EL	
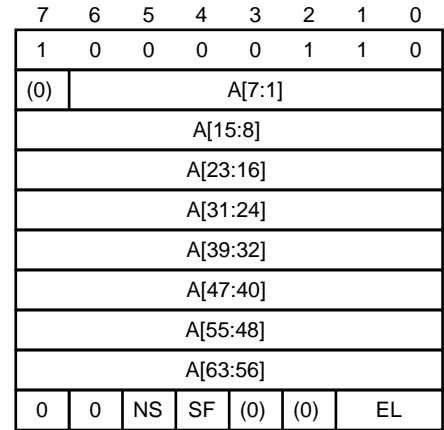
Figure D5.88: Target Address with Context 64-bit IS1 Packet (1)

Packet Layout - Variant 2

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	0	NS	SF	(0)	(0)	EL	
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							
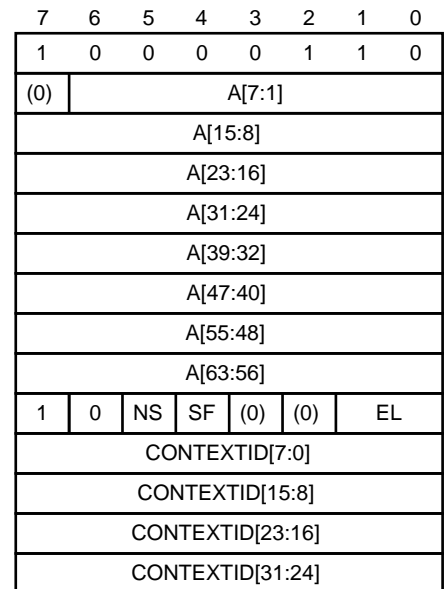
Figure D5.89: Target Address with Context 64-bit IS1 Packet (2)

Packet Layout - Variant 3

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
0	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							

Figure D5.90: Target Address with Context 64-bit IS1 Packet (3)

Packet Layout - Variant 4

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	0
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							
1	1	NS	SF	(0)	(0)	EL	
VMID[7:0]							
VMID[15:8]							
VMID[23:16]							
VMID[31:24]							
CONTEXTID[7:0]							
CONTEXTID[15:8]							
CONTEXTID[23:16]							
CONTEXTID[31:24]							

Figure D5.91: Target Address with Context 64-bit IS1 Packet (4)

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

CONTEXTID Context identifier.

When this field is not output, the Context identifier is the same as the most recently output Context identifier.

If Context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

The encoding for this field is POD.

See Context identifier tracing.

EL Exception level at this address.

The encoding for this field is POD.

`0b00`	EL0

`0b01`	EL1
`0b10`	EL2
`0b11`	EL3

NS Security state.

    The encoding for this field is POD.

`0b0`	The PE is in Secure state.
`0b1`	The PE is in Non-secure state.

SF AArch64 state.

    The encoding for this field is POD.

`0b0`	The PE is in AArch32 state.
`0b1`	The PE is in AArch64 state.

VMID Virtual context identifier.

    When this field is not output, the Virtual context identifier is the same as the most recently output Virtual context identifier.

    If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is not traced.
- This field contains a value of zero.

    The encoding for this field is POD.

    See Virtual context identifier tracing.

Element sequence

This packet encodes the following sequence:

1. *Target Address element*.
2. *Context element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.60 Source Address Short IS0 Packet

Purpose Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations All.

Packet Layout

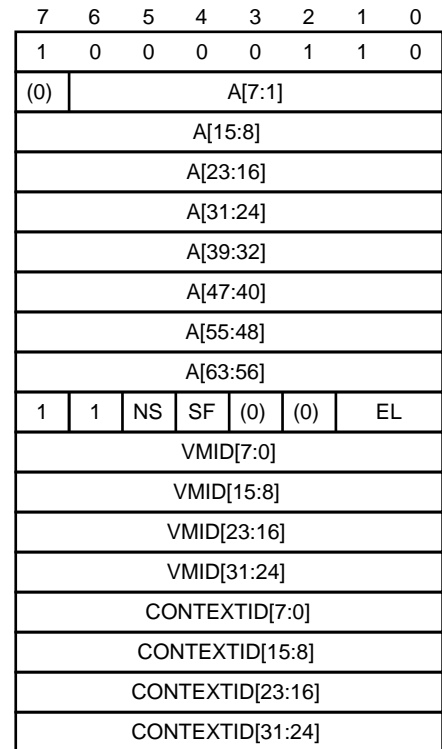| 7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	0
C0	A[8:2]						
A[16:9]							
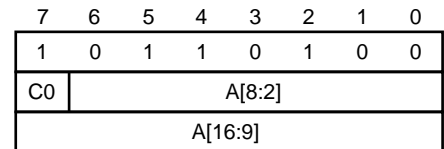
Figure D5.92: Source Address Short IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.61 Source Address Short IS1 Packet

Purpose Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	1	0	1
C0	A[7:1]						
A[15:8]							
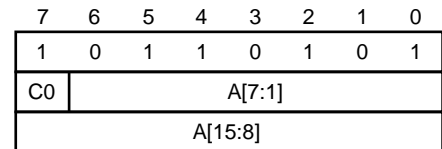
Figure D5.93: Source Address Short IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.62 Source Address 32-bit IS0 Packet

Purpose Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							

Figure D5.94: Source Address 32-bit IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.63 Source Address 32-bit IS1 Packet

Purpose Indicates the source address of a *P0 instruction*, and that the instruction was taken.

**Configurations** All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
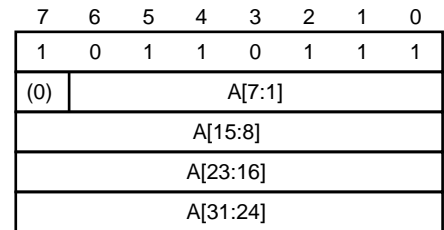
Figure D5.95: Source Address 32-bit IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.64 Source Address 64-bit IS0 Packet

Purpose Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5.96: Source Address 64-bit IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.65 Source Address 64-bit IS1 Packet

Purpose Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations All.

Packet Layout

| 7	6	5	4	3	2	1	0
1	0	1	1	1	0	0	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
A[39:32]							
A[47:40]							
A[55:48]							
A[63:56]							

Figure D5.97: Source Address 64-bit IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.66 Source Address Exact Match Packet

Purpose Indicates the source address of a *P0 instruction*, and that the instruction was taken.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	1	0	0	QE	
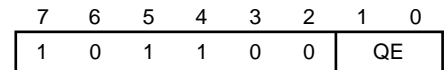
Figure D5.98: Source Address Exact Match Packet

Field descriptions

QE Instruction virtual address.

The encoding for this field is POD.

0b00	The address is the same as address history buffer entry 0.
0b01	The address is the same as address history buffer entry 1.
0b10	The address is the same as address history buffer entry 2.

Element sequence

This packet encodes the following sequence:

1. *Source Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.67 Ignore Packet

Purpose To align packet boundary to memory boundary.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	1	1	1	0	0	0	0
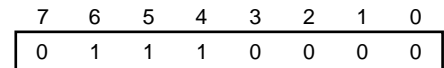
Figure D5.99: Ignore Packet

D5.68 Event Packet

Purpose Indicates 1-4 *Event elements*.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
0	1	1	1	V3	V2	V1	V0
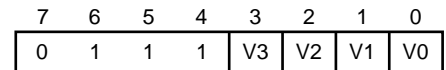
Figure D5.100: Event Packet

Field descriptions

V0 Event 0 indicator.

The encoding for this field is POD.

`0b0`	*ETEEvent* 0 did not occur
`0b1`	*ETEEvent* 0 occurred

V1 Event 1 indicator.

The encoding for this field is POD.

`0b0`	*ETEEvent* 1 did not occur
`0b1`	*ETEEvent* 1 occurred

V2 Event 2 indicator.

The encoding for this field is POD.

`0b0`	*ETEEvent* 2 did not occur
`0b1`	*ETEEvent* 2 occurred

V3 Event 3 indicator.

The encoding for this field is POD.

`0b0`	*ETEEvent* 3 did not occur
`0b1`	*ETEEvent* 3 occurred

Additional information

For more information about the decoding of this packet see decode.

Note

[V3, V2, V1, V0] != `0b0000` as this is decoded as an Ignore Packet.

D5.69 Q Packet

Purpose Indicates that some instructions have executed, without a count of the number of instructions.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	1	1	1

Figure D5.101: Q Packet

Element sequence

This packet encodes the following sequence:

1. *Q element*.

Additional information

For more information about the decoding of this packet see decode.

D5.70 Q with count Packet

Purpose Indicates that some instructions have executed.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	1	0	0

C0	COUNT[6:0]
C0	COUNT[13:7]
C0	COUNT[20:14]
C0	COUNT[27:21]
(0) (0) (0) (0)	COUNT[31:28]

Figure D5.102: Q with count Packet

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

0b0	Last byte in this section.
0b1	At least one more byte follows in this section.

COUNT The number of instructions executed.

The encoding for this field is unsigned LE128n.

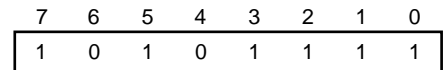### Element sequence

This packet encodes the following sequence:

1. *Q element*.

Additional information

For more information about the decoding of this packet see decode.

D5.71 Q with Exact match address Packet

Purpose Indicates that some instructions have executed with an address of the next instruction.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	0	0	TYPE	
C0	COUNT[6:0]						
C0	COUNT[13:7]						
C0	COUNT[20:14]						
C0	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5.103: Q with Exact match address Packet

Field descriptions

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT The number of instructions executed.

The encoding for this field is unsigned LE128n.

TYPE The TYPE field indicates what form the rest of the Packet takes.

The encoding for this field is POD.

`0b00`	A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 0.
`0b01`	A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 1.
`0b10`	A packet with this TYPE value also indicates a Target Address element with an address the same as address history buffer entry 2.
`0b11`	RESERVED

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

D5.72   Q short address IS0 Packet

Purpose Indicates that some instructions have executed with an address of the next instruction.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1
C0	A[8:2]						
A[16:9]							
C1	COUNT[6:0]						
C1	COUNT[13:7]						
C1	COUNT[20:14]						
C1	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5.104: Q short address IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT The number of instructions executed.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.73 Q short address IS1 Packet

Purpose Indicates that some instructions have executed with an address of the next instruction.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	0	1	1	0
C0	A[7:1]						
A[15:8]							
C1	COUNT[6:0]						
C1	COUNT[13:7]						
C1	COUNT[20:14]						
C1	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5.105: Q short address IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

C1 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT The number of instructions executed.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the generation of this packet see generation.

For more information about the encoding of this packet see encoding.

D5.74 Q 32-bit address IS0 Packet

Purpose Indicates that some instructions have executed with an address of the next instruction.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0
(0)	A[8:2]						
(0)	A[15:9]						
A[23:16]							
A[31:24]							
C0	COUNT[6:0]						
C0	COUNT[13:7]						
C0	COUNT[20:14]						
C0	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5.106: Q 32-bit address IS0 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bits[1:0] always have the value `0b00`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT The number of instructions executed.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the encoding of this packet see encoding.

D5.75 Q 32-bit address IS1 Packet

Purpose Indicates that some instructions have executed with an address of the next instruction.

Configurations All.

Packet Layout

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	1
(0)	A[7:1]						
A[15:8]							
A[23:16]							
A[31:24]							
C0	COUNT[6:0]						
C0	COUNT[13:7]						
C0	COUNT[20:14]						
C0	COUNT[27:21]						
(0) (0) (0) (0)				COUNT[31:28]			

Figure D5.107: Q 32-bit address IS1 Packet

Field descriptions

A Instruction virtual address.

Instruction virtual address bit[0] always has the value `0b0`.

The address is compressed relative to address history buffer entry 0.

The encoding for this field is Bit replacement.

C0 Continuation Bit.

The encoding for this field is Unary code.

`0b0`	Last byte in this section.
`0b1`	At least one more byte follows in this section.

COUNT The number of instructions executed.

The encoding for this field is unsigned LE128n.

Element sequence

This packet encodes the following sequence:

1. *Q element*.
2. *Target Address element*.

Additional information

For more information about the decoding of this packet see decode.

For more information about the encoding of this packet see encoding.

Chapter D6
Trace Unit
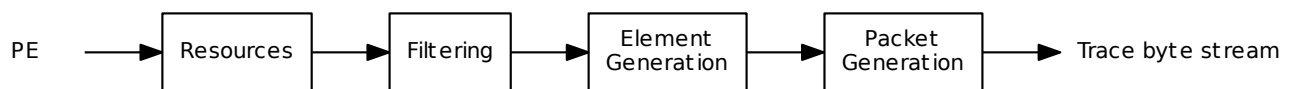
This chapter describes the behavior of a ETE trace unit.

PE → Resources → Filtering → Element Generation → Packet Generation → Trace byte stream

Figure D6.1: Stages of trace generation

D6.1 Resetting the trace unit

D6.1.1 Trace unit reset

R$_{\text{PCXJC}}$ A trace unit has a *trace unit reset*, that resets all trace unit trace registers and trace unit management registers.

R$_{\text{PTZDH}}$ When the trace unit Core power domain is powered up, a trace unit reset is applied.

I$_{\text{ZXRHG}}$ It is IMPLEMENTATION DEFINED whether the system has a mechanism to initiate a trace unit reset on demand.

S$_{\text{WVMHS}}$ In a *Processing Element* (PE) with FEAT_TRF, a PE Cold reset causes EDSCR.TFO to be reset to '0b0' which means that tracing is prohibited after the Cold reset until explicitly permitted by software. If tracing from a Cold reset is required, the debugger needs to ensure any relevant controls, including EDSCR.TFO, are programmed to permit tracing. Programming such registers might involve causing the PE to enter Debug state to ensure the registers can be programmed before the PE starts executing instructions.

Behavior on a trace unit reset

R$_{\text{WKLGX}}$ When a trace unit reset is applied, the trace unit resets the values of all trace unit registers to the values described in the individual register descriptions.

Note

Some previous trace architectures from Arm supported multiple types of reset for the trace unit.

D6.2 System Behaviors

R_{GFMRH}   The trace unit outputs all of the trace byte stream, without external stimulus, within finite time.

D6.2.1 Behavior on enabling

R_{VBMLV}   While both of the following are true, the trace unit is enabled:

- TRCPRGCTLR.EN is set to `0b1`.
- The OS Lock is unlocked.

Note

Previous trace architectures from Arm had a dedicated trace unit OS Lock, whereas ETE is dependent on the PE OS Lock.

R_{KBFFQ}   While the trace unit is enabled, the trace unit can trace all PE execution, except when any of the following are true:

- A trace unit buffer overflow occurs.
- The authentication interface prohibits the tracing of certain pieces of code.
- FEAT_TRF or FEAT_TRBE prohibit the tracing of certain pieces of code.

I_{KCDMH}   No sequences of code or PE operations are exempt from this requirement. However, while the trace unit is transitioning from an enabled to a disabled state, or from a disabled to an enabled state, some loss of trace is permitted.

I_{GDNWY}   While the trace unit is enabled, writes to most trace unit trace registers might be ignored. It is UNKNOWN whether writes to these registers succeed. When the writes are successful, the behavior of the trace unit is UNPREDICTABLE.

S_{BXQJH}   Trace analyzers must not write to most trace unit trace registers while the trace unit is enabled or TRCSTATR.IDLE indicates that the trace unit is not idle.

I_{TPTRW}   While the trace unit is enabled or idle, all resources that are visible in the programmers' model might have unstable values. Therefore, a trace analysis tool must be aware that the following values might be dynamically changing as they are being read:

- The Counter values. These are indicated by the TRCCNTVR<n>.
- The Sequencer state. This is indicated by TRCSEQSTR.
- The ViewInst start/stop function. This is indicated by TRCVICTLR.
- The Single-shot Comparator Control status. This is indicated by the TRCSSCSR<n>.

R_{VNGFG}   When the trace unit becomes enabled, the trace unit does not reset the state of any of the resources in the trace unit, including the Counters, the Sequencer, and the ViewInst start/stop function.

S_{LMPNV}   While the trace unit is disabled, and before it is enabled, a trace analyzer ensures the trace unit resources are programmed with a valid initial state.

D6.2.2 Behavior on disabling

I_{GZPBM}   While the trace unit is disabled, the trace unit is not enabled to generate trace and the trace unit resources are disabled.

R_{TMLTF}   While either of the following is true, the trace unit is disabled:

- TRCPRGCTLR.EN is set to `0b0`.
- The OS Lock is locked.

Note

Previous trace architectures from Arm had a dedicated trace unit OS Lock, whereas ETE is dependent on the PE OS Lock.

R_{ZDTLK} When the trace unit becomes disabled, the trace unit stops generating trace and empties the trace buffers by outputting any data in them.

R_{TNYDD} When the trace buffers are empty, after the trace unit has become idle after becoming disabled, TRCSTATR.IDLE indicates that the trace unit is idle.

R_{TMVLW} When the trace unit becomes disabled, all resources that are visible in the programmers' model retain their values and become stable at those values.

R_{QVYMJ} When the trace unit becomes disabled, when the resources are stable, TRCSTATR.PMSTABLE indicates that the programmers' model is stable.

For more information, see D7.1.2 *Behavior of the resources while in the Pausing state*.

R_{GLBHL} When the trace unit becomes disabled after the trace unit has generated *Event elements*, the trace unit outputs the *Event elements* before TRCSTATR.IDLE indicates that the trace unit is idle.

R_{YFLJT} While the trace unit is disabled, the following are true:

- No trace is generated.
- All trace unit resources and ETEEvents are disabled.

D6.2.3 Behavior on flushing

I_{XRMWS} The trace unit is allowed to buffer the trace byte stream to make efficient use of system infrastructure.

I_{WHZBD} As the trace unit is allowed to delay the output of the trace byte stream to the system infrastructure, there are system events that require all of the trace byte stream to be observable to other observers in the system.

I_{CXLCR} Making the trace byte stream visible to other observers is known as a trace unit flush.

R_{JLRQH} When any of the following occur, a trace unit flush is requested:

- The trace unit transitions from an enabled to a disabled state.
- The trace capture infrastructure requests a trace unit flush.
- A `TSB CSYNC` instruction is executed in a Trace Prohibited region while the Trace Buffer Extension is implemented and enabled.

I_{KGJRL} A trace unit flush might be requested for IMPLEMENTATION DEFINED reasons. For example:

- Before the trace unit enters either:
    – The low-power state.
    – A powerdown state.
- The PE enters Debug state.

I_{ZWHKM} An example of a trace unit flush is one requested on an Arm AMBA ATB interface *AMBA ATB Protocol Specification* [4].

R_{HGYLG} When a trace unit flush is requested, the trace unit performs the following tasks before responding to the flush request:

1. Encode any remaining elements into trace packets.
2. Complete any packets that are in the process of being generated.
3. Output all trace packets for all PE execution that occurred before the flush request was received.

I_{LMVMT} An example of when the trace unit might need to encode remaining elements into trace packets before a trace unit flush is when there are *Commit elements* that are not yet encoded.

R_{TWBVY} When a trace unit flush occurs while the trace unit is recovering from a trace unit buffer overflow, the trace unit outputs the corresponding *Overflow element* before responding to the flush request.

I_{GHKFH} When a trace unit flush occurs, the trace unit either continues to generate trace or stops generating trace, depending on what condition caused the trace flush. For example, if a flush occurs because the trace unit is entering a disabled state, then tracing becomes inactive after the trace flush.

R_{TTDBB} When a condition causes both a trace unit flush and the trace unit to stop generating trace, the trace unit stops generating trace before responding to the flush request, and before indicating that the trace unit is idle.

I_{NHBMZ} On entry to Debug state, Arm recommends that the *Exception element* indicating entry to Debug state is included in the flushed trace data if tracing is active.

R_{PFHWW} When a trace unit flush is requested, the trace unit outputs the data within a finite period.

R_{DKFRL} When a trace unit flush is requested and the cause of the flush request requires an acknowledgement, the trace unit generates the acknowledgement within a finite period.

I_{SCBMG} The flush request mechanism on AMBA ATB is an example of a cause of a flush request that also requires an acknowledgement.

D6.2.4 Low-power state

X_{GHHNW} The low-power state in the trace unit is a mechanism to improve energy efficiency during periods where trace generation is limited.

Scenarios where the trace unit might be in the low-power state are any of the following:

- The PE is in a low-power state.
- The PE is in Debug state.

R_{LHDSS} The trace unit is only permitted to be in the low-power state when any of the following are true:

- The PE is in a low-power state due to the Wait for Event mechanism.
- The PE is in a low-power state due to the Wait for Interrupt mechanism.
- The PE is in Debug state.
- The trace unit is Disabled.

D6.2.5 Trace unit behavior when the PE is in a low-power state

I_{MSTWP} The PE that is being traced might support a low-power state where no execution occurs. This low-power state might be invoked, for example, when the PE executes a `WFI`, `WFIT`, `WFE`, or `WFET` instruction.

R_{WMPTL} While the trace unit is in the Disabled state, the trace unit does not stop the PE from entering a low-power state.

R_{YLDDV} While the trace unit is in Low-power Override Mode, the trace unit does not affect the operation of the PE.

D6.2.6 Trace unit behavior in the low-power state

R_{FMXFM} While the trace unit is enabled, when the trace unit enters the low-power state, the trace unit continues to appear enabled throughout the time it is in the low-power state.

R_{KQVNN} When the trace unit enters or leaves the low-power state, the trace unit does not lose resource events that are in transition through the trace unit, except those permitted when moving through the Pausing state of the resources. See D7.1.2 *Behavior of the resources while in the Pausing state* for details on the resource events that are permitted to be lost when in the Pausing state.

I_{RVKHK} Observation of resource events that are in transition through the trace unit when it enters the low-power state might not occur until after the trace unit leaves the low-power state.

R_{RGFJY} While the trace unit is not in the low-power state, and before it enters the low-power state, the resources enter the Paused state. See D7.1.3 *Behavior of the resources while in the Paused state*.

I_{MXHHN} If WFI and WFE instructions are classified as *P0 instructions*, see TRCIDR2.WFXMODE, and the trace unit enters the low-power state as a result of a WFI or WFE instruction, Arm strongly recommends that the following elements are generated before the trace unit enters the low-power state:

- The *Atom element* that represents the a WFI, WFIT, WFE, or WFET instruction.
- Any pending *Commit elements*.

R_{LBDSM} While the trace unit is in the low-power state, the trace unit does not generate trace, including ETEEvent trace.

R_{MFBDT} While the trace unit is in the low-power state, the resources remain in the state that they were in before the trace unit entered the low-power state.

I_{QXBYK} The resources are:

- The Counters.
- The Sequencer.
- The ViewInst start/stop function.
- The Single-shot Comparator Controls.

R_{FHYHC} While the trace unit is in the low-power state, the trace unit drives all External Outputs low.

R_{DNKDV} While the trace-unit is in the low-power state, the PE and external debugger are able to access the trace unit trace registers and trace unit management registers unaffected.

R_{XTBQX} While the trace unit is in the low-power state, when a trace protocol synchronization request occurs, the trace unit handles the trace protocol synchronization request correctly. See D1.9 *Trace protocol synchronization* for information on how the trace unit handles trace protocol synchronization requests.

R_{TWQJT} While the trace unit is a retention state, external debugger accesses to the trace unit behave as if there is no power to the trace unit Core power domain.

I_{RCXZX} While the trace unit is in the low-power state, the trace unit might not recognize external events, such as the assertion of any External Inputs.

I_{BPQTL} While the trace unit is in the low-power state, it is IMPLEMENTATION DEFINED whether the cycle counter continues to count or not.

I_{VTRBC} While the trace unit is in the low-power state, timestamp requests might be ignored.

I_{ZTDMB} It is possible that the trace unit might intermittently leave and re-enter the low-power state while the PE is in a low-power state. If this happens, the trace unit resources might become intermittently active during this time. In addition, trace generation might also become intermittently active, and this means that the trace unit might output some packets. This behavior is IMPLEMENTATION DEFINED.

I_{ZVDSF} There is no additional requirement for the trace unit to generate a *Trace Info element* or *Trace On element* when leaving the low-power state. However, if the trace unit entered the low-power state because the PE was in Debug state, the normal requirements for restarting trace after leaving Debug state apply, including generation of a *Trace On element*. See D6.3 *Trace unit behavior while the PE is in Debug state*.

I_{LQFRD} The trace unit can be programmed so that it does not enter the low-power state, by enabling Low-power Override Mode. Low-power Override Mode is selected using TRCEVENTCTL1R.LPOVERRIDE.

R_{VHSFL} When Low-power Override Mode is enabled, the resources continue operating and the trace unit can generate trace.

I_{FRMMP} Low-power Override Mode does not affect the operation of the PE, however it is not required to prevent the PE from entering a low-power state. This means that even though the trace unit can generate trace, it might only generate *Event elements*.

D6.3 Trace unit behavior while the PE is in Debug state

R_{XJXQS} While ViewInst is active, when the PE enters Debug state, the trace unit generates an *Exception element* which indicates that the PE has entered Debug state.

R_{YMJFJ} When the PE enters Debug state, ViewInst becomes inactive, and remains inactive throughout the time the PE is in Debug state.

R_{DPKSC} While the PE is in Debug state, the trace unit does not trace instructions that are executed.

R_{HBNFJ} When the PE exits Debug state and ViewInst becomes active, the trace unit generates a *Trace On element*.

R_{TGFHM} While the PE is in Debug state, the ViewInst start/stop function maintains its state.

I_{WFYLQ} If an *Exceptional occurrence* occurs between the PE exiting Debug state and the PE executing the first instruction, the value of TRCRSR.TA is used to determine if the *Exceptional occurrence* is traced. In general, if the entry to Debug state was traced then TRCRSR.TA will be set to $0b1$, and therefore this *Exceptional occurrence* on exit from Debug state is traced.

I_{NPQLT} If a PE Reset occurs when the PE is in Debug state this is treated as leaving Debug state. This means that a *Trace On element* and an *Exception element* indicating a PE Reset are traced if tracing is not prohibited and either of the following are true:

- TRCRSR.TA is 0b1.
- Forced tracing of PE Resets is active.

D6.4 Trace unit behavior on a trace unit buffer overflow

R_{PQGXB} When a trace unit buffer overflow occurs, trace generation becomes inoperative until the trace unit can recover from the overflow.

R_{RQHFH} When a trace unit buffer overflow occurs, the trace unit does not output a partial trace packet, that is, the trace unit can only output complete packets.

I_{TDCNT} The *Overflow element* indicates to a trace analyzer that a trace unit buffer overflow has occurred. See D6.9.15 *Overflow Element* for details on the generation of an *Overflow element*.

I_{FQVRZ} See D6.9.6 *Event Element* for details of the effect of a trace unit buffer overflow on *Event element* generation.

R_{DQBDH} When the trace unit recovers from a trace unit buffer overflow, the following occur:

- Trace protocol synchronization is requested.
- Trace protocol synchronization occurs before the trace unit outputs any packets.

I_{VQYYH} When an Overflow packet is generated, the trace unit might output any of the following packets before it outputs an Alignment Synchronization packet:

- Event packet.
- Overflow packet.
- Discard packet.
- Ignore packet.

I_{YYNRQ} Arm recommends that the Alignment Synchronization packet is the first packet output after the Overflow packet.

D6.5 Trace unit power states

I_{GNFXM} The Arm architecture *Arm® Architecture Reference Manual, for A-profile architecture* [1] defines the following power-states:

Normal The trace unit Core power domain is fully powered up and the trace unit registers are accessible.

Standby The trace unit Core power domain is on, but there are measures to reduce energy consumption. Standby is transparent, meaning that to software and to an external debugger it is indistinguishable from normal operation.

Retention The OS takes some measures, including IMPLEMENTATION DEFINED code sequences and registers, to reduce energy consumption. Trace unit registers cannot be accessed. A trace unit reset does not occur on leaving Retention.

Powerdown The OS takes some measures to reduce energy consumption by turning the trace unit Core power domain off. Trace registers cannot be accessed. A trace unit reset occurs on leaving Powerdown.

I_{MNDVQ} A trace unit might support a low-power state, which is equivalent to the Standby state.

I_{BMSVN} A trace unit might support a Retention state or a Powerdown state, and both of these states are considered to be a state where the trace unit Core power domain is powered down.

I_{ZVBZF} If the trace unit is implemented in a power domain which is separate from the PE power domain, all of the following are true:

- The trace unit Core power domain might be able to be powered down without powering down the PE power domain.
- The trace unit Core power domain is always powered down when the PE power domain is powered down.

I_{CNZJH} A read of TRCPDSR returns information about the current state of the trace unit and Table D6.1 shows the meanings of the returned value.

Table D6.1: Meaning of TRCPDSR values

STICKYPD	POWER	Meaning
0b0	0b1	The trace unit Core power domain is powered and the trace unit registers are accessible.
0b1	0b1	The trace unit Core power domain is powered and the trace unit registers are accessible. A trace unit reset or power interruption has occurred so the trace unit register state might not be valid.

R_{CMKXK} When the trace unit Core power domain transitions from powered down to powered up, if the trace unit register state has been preserved over the power down then TRCPDSR.STICKYPD is restored to the value before power down.

R_{FQMXQ} When the trace unit Core power domain transitions from powered down to powered up, if the trace unit register state has not been preserved over the power down then TRCPDSR.STICKYPD is set to 0b1.

> **Note**
>
> Previous trace architectures from Arm supported multiple power domains in the trace unit. ETE only supports a single power domain and therefore TRCPDSR.POWER is always 0b1.

I_{FRMBB} A system might support a Debug power domain that contains the interface between the trace unit and the external debugger. The Debug power domain usually needs to be powered up when the external debugger needs to connect

to the system. Such a Debug power domain is described in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

R_{GYLKD} If the trace unit Core power domain can be powered down independently of the Debug power domain, Arm recommends the system implements an external debug component with a Power-up request mechanism which can request the trace unit Core power domain to be powered up.

R_{ZNSNS} Arm strongly recommends the Power-up request mechanism is a CoreSight Class 0x9 ROM Table containing a parent entry for the trace unit. A parent entry of a component is one of:

- An entry in the ROM table that locates the component.
- An entry in a first ROM table that locates a second ROM table that includes a parent entry for the component. The second ROM table is a descendant of the first ROM table.

I_{TLYXG} This definition of a parent entry is recursive, and therefore the Power-up request mechanism might be high up in a hierarchy of ROM tables. The ROM table and any descendants might describe other debug components, including debug components for other PEs. The ROM table might have parent entries in other ROM tables, and those parent entries might also have a Power-up request mechanism in those ROM tables.

R_{ZPCZC} If the Power-up request mechanism is implemented, in the Class 0x9 ROM Table containing the Power-up request mechanism for the trace unit:

- The POWERIDVALID bit in the parent entry must be `0b1`.
- The POWERID field in the parent entry has an IMPLEMENTATION DEFINED value.

I_{DXHPS} It is IMPLEMENTATION DEFINED whether the trace unit Core power domain is the PE Core power domain or some other power domain. For more information on the CoreSight Class 0x9 ROM Table, see the *ARM CoreSight Architecture Specification* [5].

D6.6 Visibility of the PE operation

I_{BMPFH} This section describes the ability of the trace unit to trace the execution of the operation of the PE.

R_{QCFMC} When the trace unit performs indirect reads of PE System registers, the trace unit complies with the rules associated with Context synchronization events.

R_{QHTYC} When the trace unit performs indirect reads of PE System registers, the trace unit complies with the rules associated with the TSB CSYNC instruction as defined in Chapter E1 *Trace Buffer Extension*.

R_{QCFSS} When instructions are executed outside of any Trace Prohibited region, the trace unit observes the execution.

R_{CDNKX} When observable instructions are executed, the trace unit observes all execution before a PE Context synchronization event occurs, as defined in Chapter E1 *Trace Buffer Extension*.

R_{QMBKJ} When an *Exceptional occurrence* occurs outside of any Trace Prohibited region, the trace unit observes the *Exceptional occurrence*.

I_{XBJFP} Executing a TSB CSYNC instruction generates a Trace synchronization event as defined in Chapter E1 *Trace Buffer Extension*.

R_{FCBLJ} When a TSB CSYNC instruction is executed in a Trace Prohibited region, the TSB CSYNC instruction does not become Microarchitecturally-finished until the resources are in the Paused state or the trace unit is in the Idle or Stable state.

I_{JYJDZ} While the PE is outside a transaction, after a TSB CSYNC instruction executed inside a Trace Prohibited region generates a Trace synchronization event, the Trace synchronization event is microarchitecturally-finished when the *trace operation* has microarchitecturally-finished for every instruction before the Context synchronization event before the TSB CSYNC instruction that generated the Trace synchronization event.

For more details on the TSB CSYNC instruction, see R_{MRVPT}.

R_{TSLRT} While the PE is inside a transaction, when a Trace synchronization event occurs, the Trace synchronization event becomes Microarchitecturally-finished within a finite period.

I_{HNSGS} While the PE is inside a transaction, the completion of a Trace synchronization event is not dependent on the resolution of the transaction. It might still be dependent on other aspects of the trace operation.

R_{XLVQM} When a TSB CSYNC instruction executed in a Trace Prohibited region becomes Microarchitecturally-finished, the trace unit generates no more trace until the PE leaves the Trace Prohibited region.

I_{CZLXW} When a TSB CSYNC is executed in a Trace Prohibited region, the rules around generation of a trace flush and requiring no more trace to be generated in the Trace Prohibited region mean that only whole trace packets are output, and the last byte output is the end of a packet.

X_{GSXJJ} These rules ensure that no new trace is generated and allows various system registers to be changed, such as those controlling the Trace Buffer Extension, without the risk of any trace being generated while those registers are being changed.

R_{XRWPV} When the trace unit becomes enabled in a Trace Prohibited region, the trace unit generates no trace until the PE leaves the Trace Prohibited region.

X_{TGNBT} The above rule ensures that no trace is generated until the PE leaves the Trace Prohibited region, and therefore allows various system registers to be changed, such as those controlling the Trace Buffer Extension, without the risk of any trace being generated while those registers are being changed.

I_{KXDDS} The trace operation as defined in Chapter E1 *Trace Buffer Extension* can be split into operations that are performed by one of the following:

- The PE.
- The ETE trace unit.
- The trace buffer.

The operation of the trace unit is defined by the *ETE trace operation*.

R_{RFJQN} If the Trace Buffer Unit is implemented and enabled, when a Trace synchronization event occurs, and after all of the trace byte stream generated by the trace unit is flushed to the trace buffer, the Trace synchronization event completes.

Table D6.2: Labels for ordering diagrams

Notation	Name	Description
po	program-order	*head* is in **program order** after *tail*.
rf	Reads-from	*tail* **Reads-from** *head*.
co	Coherence-after	*head* is **Coherence-after** *tail*.
fr	from-read	As *co*, except that the operation at *head* is a read.
ob	Observed-by	*tail* is **Observed-by** *head*. Only applies for different Observers.
tb	traced-by	*head* is the *trace operation* for the instruction at *tail*.
gb	generated by	*head* is an operation generated by the instruction at *tail*.
seo	speculative execution-order	The PE speculated that the instruction at *head* was executed after *tail*, but the instruction was later Canceled or was part of a Transaction that Failed or was Canceled. An *seo* arrow might be paired with a *can* arrow that shows this.
can	canceled	The instruction at *tail* was Canceled when the instruction at *head* was resolved, or the Transaction containing *tail* Failed or was Canceled.

D6.6.1   ETE trace operation

R_{YCJXC} Each instruction has all of the following state information:

- PC.
- PSTATE.T.
- PSTATE.EL.
- The Security state.
- CONTEXTIDR_EL1.PROCID.
- CONTEXTIDR_EL2.PROCID.
- TRFCR_EL1.
- TRFCR_EL2.
- MDCR_EL3.STE.
- TxNestingLevel.

I_{GCLMS} The trace information generated contains *Address information* in *Target Address elements*, *Source Address elements*, *Exception elements*, and *Q elements*. The *Address information* contains:

- The virtual address of an instruction.
- The instruction set, known as the sub_isa.

I_{JTLPL} The trace information generated contains *Context information* in *Context elements*. The *Context information* contains:

- The current Security state.
- The current Exception level.
- The current Execution state, which is AArch32 or AArch64.
- The current Context identifier, as stored in CONTEXTIDR_EL1.PROCID.

• The current Virtual context identifier, as stored in CONTEXTIDR_EL2.PROCID.

R_{WBCRN} When an instruction is executed and all the trace elements for the instruction have been generated, the trace operation for the instruction is complete.

I_{WXNXB} Trace elements generated for an instruction might include:

• Global timestamp elements.
• Cycle count elements.
• Speculation resolution elements.
• Transaction resolution elements.

I_{FKTKY} For example, the tracing of PE execution is where:

• Resolved instruction **A** is executed in program order before a Resolved instruction **B**.
• $\mathbf{t_A}$ is all the trace elements that are generated due to the tracing of instruction **A**.
• $\mathbf{t_B}$ is all the trace elements that are generated due to the tracing of instruction **B**.
• The trace elements for $\mathbf{t_A}$ must be observed before $\mathbf{t_B}$.

Figure D6.2 shows this.
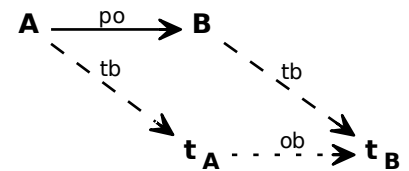
Figure D6.2: Trace operation

D6.6.2 Impact on PE Behavior

I_{LLKFT} The ETE architecture places no requirements on the impact that trace generation has on the functional performance of a PE. Arm expects that trace unit implementations are designed according to the market requirements of the PEs being traced, and according to the trace requirements for those PEs. For some markets and trace requirements, the trace solution might always have some performance impact on the PE and the ETE architecture does not prohibit this.

D6.6.3 Behavior on a PE Warm reset

R_{YYHBF} A PE Warm reset does not cause a Trace unit reset.

X_{ZRQTC} This ensures that tracing is possible through a PE Warm reset.

I_{QBSCX} A PE Warm reset might occur at the same time as a Trace unit reset, however, these are asynchronous and unrelated events.

D6.6.4 Instruction Block

X_{TYXHR} How instructions are executed can vary significantly between PE designs. To allow for these variations the ETE architecture allows some flexibility within the filtering model. Rather than applying the filtering model to individual instructions it is applied to blocks of instructions.

R_{BQTBL} An instruction block contains one or more instructions.

R$_{\text{GDZBX}}$ An instruction block can contain zero or one *P0 instructions*.

R$_{\text{CVJQH}}$ When an instruction block is generated which contains a *P0 instruction*, the instruction block has the *P0 instruction* as the last instruction in the block.

R$_{\text{HPJTP}}$ *Exceptional occurrences* do not occur between instructions in an instruction block.

R$_{\text{LDJXZ}}$ The addresses of the instructions within an instruction block are sequential.

I$_{\text{JCQHC}}$ The number of instructions in a block can vary from block to block and can vary each time the same sequence of instructions are executed.

I$_{\text{HRBJG}}$ For example, the tracing of an instruction block is where:

- Resolved instruction **A** is executed in program order before a Resolved instruction **B**.
- **t$_A$** is all the trace elements that are generated due to the tracing of instruction **A**.
- **t$_A$** is all the trace elements that are generated due to the tracing of instruction **B**.

Figure D6.3 shows this.
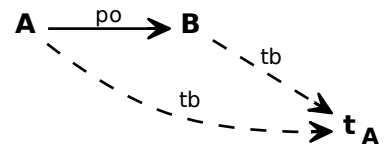
Figure D6.3: Instruction block trace operation

D6.6.5 Exposing Speculation

I$_{\text{DCDNQ}}$ For some PE microarchitectures the tracing of execution-order only might not be achievable. The ETE architecture provides the ability to trace speculatively executed instructions.

R$_{\text{TRVLX}}$ When speculative instructions are observed, the trace unit indicates whether each instruction is resolved or canceled with a resolve operation or a cancel operation.

R$_{\text{PPJSK}}$ A resolve operation indicates that one or more instructions have, or will be, architecturally executed.

R$_{\text{WZBLY}}$ A cancel operation indicates that one or more instructions, although observed by the trace unit, did not architecturally execute.

I$_{\text{KQYZB}}$ There is no requirement to expose any speculation to the trace unit.

I$_{\text{DKDHD}}$ For example, the tracing of speculation execution is where:

- **S** is executed in speculative execution-order after a Resolved instruction **A**.

- **A** is executed in program order before a Resolved instruction **B**.

- **S** is not in speculative execution-order after **B**.

- **Q** is executed in speculative execution-order after a Resolved instruction **B**.

Figure D6.4 shows this.

R$_{\text{HPJTP}}$

Figure D6.4: Observation of Speculative Trace operation

D6.6.6 Trace Prohibited regions
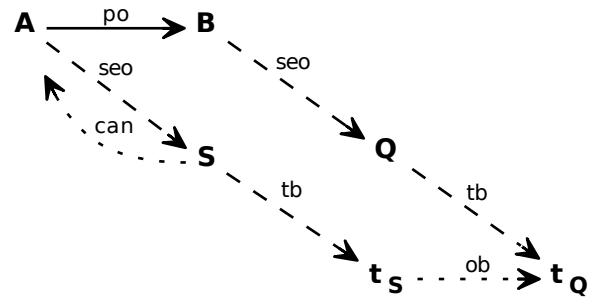
X_{THCBC} Trace Prohibited regions are instruction address regions or periods of execution by the PE that are not to be traced. Instructions and *Exceptional occurrences* which are not prohibited are not necessarily traced because the trace unit has a number of trace filtering functions to limit the amount of trace generated to the sections or periods of interest.

I_{RJYNL} An executable program might contain regions of code that are prohibited to trace. These regions might be associated with a higher Security state, or with the PE entering a privileged mode so that it can execute the instructions that are contained within them.

Tracing might be prohibited while the PE is operating in certain states or modes. For example:

- Non-invasive debug might be prohibited while the PE is in Secure state.
- FEAT_TRF might prohibit tracing.
- FEAT_TRBE might prohibit tracing.

Trace might also become prohibited if, while tracing program execution, an authentication interface changes the currently permitted level of non-invasive debug. For example, if trace is permitted and active while the PE is operating in Secure state, and then the permitted level of non-invasive debug changes from being permitted for Secure state, to not permitted, then trace becomes prohibited.

I_{FKMSC} The *Arm® Architecture Reference Manual, for A-profile architecture* [1] describes when FEAT_TRF prohibits tracing.

     Chapter E1 *Trace Buffer Extension* describes when FEAT_TRBE prohibits tracing.

R_{HYZLQ} If an optional authentication interface is implemented, while Secure non-invasive debug is disabled according to the optional authentication interface and while `SelfHostedTraceEnabled()` returns FALSE, then for execution in Secure state, the PE executes in a Trace Prohibited region.

I_{NVSDD} An example of an optional authentication interface is the CoreSight Authentication interface *ARM CoreSight Architecture Specification* [5].

R_{FFVYM} While the PE is executing code from a Trace Prohibited region, the trace unit does not trace instructions or *Exceptional occurrences*, including PE Resets.

R_{KTMLZ} While the PE is executing code from a Trace Prohibited region, instruction Address Comparators do not match on any instructions in the Trace Prohibited region.

R_{SZRZR} While cycle counting is enabled and the PE is executing code from a Trace Prohibited region, the cycle counter continues to count.

I_{MCCBH} When the PE leaves a Trace Prohibited region and tracing restarts, the cycle counter includes cycles spent in the Trace Prohibited region in the cycle count.

I$_{SDSGK}$ The behavior of the resources when entering a Trace Prohibited region is defined in D7.1.3 *Behavior of the resources while in the Paused state*.

R$_{VHRGM}$ While the PE is executing code from a Trace Prohibited region, the trace unit does not output any trace that might provide information about the execution in the Trace Prohibited region.

I$_{JSKLD}$ Examples of information about execution in a Trace Prohibited region that trace might provide are the context of execution, instruction addresses, and the address of the first instruction in the Trace Prohibited region.

I$_{MKQWS}$ The most common cause of an entry into a Trace Prohibited region is an *Exceptional occurrence* or Context synchronization event.

R$_{THBVD}$ When an *Exceptional occurrence* that must be traced causes the PE to enter a Trace Prohibited region, the trace unit generates an *Exception element* that indicates the exception type.

R$_{CKZMR}$ When the PE enters a Trace Prohibited region and there are unresolved speculative *P0 elements* remaining in the trace byte stream, when the resolution of the speculative elements is known the trace unit generates the appropriate *Commit elements* or *Cancel elements*.

R$_{RXMJF}$ When the PE leaves a Trace Prohibited region and ViewInst is active, that is, any filtering applied dictates that ViewInst is active, the trace unit generates a *Trace On element*.

I$_{QGFJF}$ The purpose of the trace unit generating a *Trace On element* when the PE exits a Trace Prohibited region and ViewInst is active is to indicate to the trace analyzer that there has been a discontinuity in the trace element stream.

I$_{JBFTB}$ If the PE leaves a Trace Prohibited region other than when a Context synchronization event occurs, the Trace Prohibited region is permitted to extend up to the next Context synchronization event. Typically, a PE leaves a Trace Prohibited region via a Context synchronization event, but a PE might leave a Trace Prohibited region when the authentication interface changes, or when moving from Secure to Non-secure state without an exception return.

I$_{DMXPF}$ If an *Exceptional occurrence* occurs between the PE exiting a Trace Prohibited region and the PE executing the first instruction, the value of TRCRSR.TA is used to determine if the *Exceptional occurrence* is traced.

D6.6.7 Multi-threaded processor

R$_{KBZTZ}$ Processors with multiple threads or PEs have a trace unit for each thread or PE.

I$_{KHSWQ}$ The processor might support enabling and disabling of threads, either at PE Reset time or dynamically. The trace units for the threads that are disabled might behave in one of the following ways:

- The trace unit Core power domain is powered down.
- The trace unit Core power domain is held in the trace unit reset state.

I$_{RFSNL}$ Arm recommends that the trace units for threads that are permanently disabled are not visible: either they are not included, or they are marked as not present in any ROM tables that describe the system.

D6.6.8 Sharing between multiple PEs

Note

Previous Trace architectures have allowed the trace unit to be shared between multiple PEs.

R$_{TLJSQ}$ A trace unit only traces a single PE, that is, it cannot be shared between multiple PEs.

D6.7 Speculation resolution

I_{RKYCD} The trace unit implements a maximum speculation depth, that is, the maximum permitted number of *P0 elements* that can be speculative at any instance. TRCIDR8.MAXSPEC indicates the IMPLEMENTATION DEFINED maximum speculation depth.

R_{GLQPL} The trace unit never outputs more speculative *P0 elements* than the maximum speculation depth.

I_{KCFGW} If a trace unit is not exposed to any speculative execution, then Arm recommends that the trace unit implements a maximum speculation depth of zero, and in this case:

- *Cancel elements* are not generated.
- *Commit elements* are generated after each *P0 element*, causing each *P0 element* to be immediately resolved when it is generated. The instruction trace protocol implicitly generates these *Commit elements* for each *P0 element*, meaning that explicit Commit packets are not required.
- *Mispredict elements* are not generated.

I_{QLKDL} ETE defines *Commit element* and *Cancel elements* to allow the speculation of the *P0 elements* to be resolved by the trace analyzer. The trace unit is required to calculate the number of *P0 elements* which are committed or canceled. There are many methods by which these numbers can be calculated, but in the generic case the trace unit can use the following mathematical procedure.

The PE can speculatively indicate blocks of instructions to the trace unit. Each block of instructions is given a tag where $tag \in 0, \ldots, m$ and m = "Number of rewind points supported by the PE".

The number of instructions per block can be random from the set $\mathbb{N}$ and there is a maximum of one *P0 instruction* per block. The order in which the tags are used can be random, but a tag cannot be reused until the previous block with that tag has been resolved, canceled or merged.

This procedure generates a transform from the potentially random sequence of core tags to a more useable space. The transform T evolves over time as the tags are reused and provides the mapping onto the new space,

$$T_t = [c_0, \ldots, c_m] \tag{D6.1}$$

and c_i is the mapping for core tag i.

$c_i \in 0, \ldots, q$, where $q > m$

D6.7.1 Initialization

I_{HRDQL} To perform the necessary calculations, the trace unit tracks the transform of the last resolved block. γ_t = "last committed indicator". The algorithm is initialized at $t = 0$ to

$$\forall i \in 0, \ldots, m : T_0[i] = \gamma_0 \tag{D6.2}$$
$$x_0 = \gamma_0 \tag{D6.3}$$

D6.7.2 New block operation

$\mathtt{I}_{\text{SMGSG}}$ The sequence of the numbers in the transformed space, x_t, is defined by the following equation:

$$x_{t+1} = \begin{cases} |(x_t + 1) \bmod q| & \text{If a traced *P0 instruction*} \\ x_t & \text{Otherwise} \end{cases} \tag{D6.4}$$

$$T_{t+1}[tag_t] = \begin{cases} |(x_t + 1) \bmod q| & \text{If a traced *P0 instruction*} \\ T_t[tag_t] & \text{Otherwise} \end{cases} \tag{D6.5}$$

D6.7.3 Resolved operation

$\mathtt{I}_{\text{BJFLR}}$ The PE can resolve one or more blocks in an atomic operation. This is performed by indicating the youngest block's tag to be resolved, and by inference all older blocks. $l = $ youngest block's tag

The number required by the *Commit element* is calculated by

$$n_+ = |(T_t[l] - \gamma_t) \bmod q| \tag{D6.6}$$

The state of the transform is updated by

$$\gamma_{t+1} = T_t[l] \tag{D6.7}$$

D6.7.4 Cancel operation

$\mathtt{I}_{\text{HJYQH}}$ The PE can cancel one or more blocks in an atomic operation. This is performed by indicating the oldest block to be canceled. $r = $ oldest block's tag

The number required by the *Cancel element* is calculated by

$$n_- = |(x_t - T_t[r]) \bmod q| \tag{D6.8}$$

The state of the transform is updated by

$$x_{t+1} = T_t[r] \tag{D6.9}$$

D6.8 Filtering trace generation

X_{CCWZG} The amount of trace that can be generated by the trace unit can be significant. Not all the operations of the PE are always relevant. The amount of trace generated can be reduced by the use of the trace unit filter functions.

D6.8.1 ViewInst function

R_{GQFSB} The filtering function of the instruction trace is expressed as a calculation evaluated for each instruction.

$$\text{ViewInst}_i = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug state} \\ S_i \wedge I_i \wedge E_i \wedge N_i & \text{Otherwise} \end{cases} \tag{D6.10}$$

$$S_i = \text{ViewInst start/stop function} \tag{D6.11}$$
$$I_i = \text{ViewInst include/exclude function} \tag{D6.12}$$
$$E_i = \text{Exception level filtering} \tag{D6.13}$$
$$N_i = \text{Resource event based filtering} \tag{D6.14}$$

R_{BZXFH} While ViewInst_i is high, the trace unit traces all instructions.

I_{MJMCV} Instructions for which ViewInst_i is low might be traced. This might be as a result of tracing the next *P0 element* or optimizations in the trace unit.

R_{XPNZL} When ViewInst_i is high for an instruction in an instruction block, the trace unit traces the entire instruction block.

R_{KHDNX} When ViewInst_i becomes high, the trace unit traces the next *P0 instruction* or *Exceptional occurrence*.

X_{NSMSV} Some instruction types cause the trace unit to generate *P0 elements*, so that they are explicitly traced. Other instruction types however are not explicitly traced. The execution of these other instruction types can be inferred from the *P0 elements*. This means that the following scenario is possible:

- While ViewInst is high, some instructions are executed. This means that ViewInst is indicating that those instructions must be traced. However, none of the executed instructions cause the trace unit to generate a *P0 element*, therefore none of the instructions are traced.
- ViewInst then goes low.
- The PE then executes an instruction that, whenever ViewInst is high, causes the trace unit to generate a *P0 element*.

In this scenario, although ViewInst is low when the instruction in step 3 is executed, indicating that the instruction is not traced, tracing of the instruction is permitted because this is the only way that the preceding instructions can be traced.

I_{FGSBW} There is no requirement for the target address of a *P0 instruction* or *Exceptional occurrence* to be traced if ViewInst has transitioned to a low state by the time program execution has moved to the target.

I_{LCXHX} Unless the target instruction block is traced, any *Target Address elements* indicating the target address of a *P0 instruction* or *Exceptional occurrence* cannot be relied upon.

Resource event based filtering

R_{BQNWP} The resource event based filtering part of the ViewInst function is expressed as the following equation:

$$N_i = \text{Fn}(\text{TRCVICTLR.EVENT.SEL}, \text{TRCVICTLR.EVENT.TYPE}) \tag{D6.15}$$

Where $\text{Fn}(\text{TRCVICTLR.EVENT.SEL}, \text{TRCVICTLR.EVENT.TYPE})$ selects the combination of Resource Selectors used for resource event based filtering.

I_{KMXNS} The timing of the resource event based filtering is IMPLEMENTATION SPECIFIC.

I_{DWWVR} Resource event based filtering can be used to make ViewInst active based on system conditions or on trace unit resources. For example:

- Sampling based on instruction counts.
- Activating tracing on the nth function call.
- Performance Monitoring Unit events.

R_{NBYKR} When an instruction block is processed by the trace unit during a cycle, the trace unit samples the ViewInst function resource event input during that cycle.

R_{SRMMD} When no instruction blocks are processed by the trace unit during a cycle, the trace unit does not sample the ViewInst function resource event input during that cycle.

Exception level filtering

I_{HNPYV} This function provides a simple method of filtering out information about different Exception levels without the need to use of additional resources.

R_{LWYJR} The Exception level based filtering part of the ViewInst function is expressed as the following equation:

$$E_i = \begin{cases} \neg\text{TRCVICTLR.EXLEVEL\_S\_EL0} & \text{Secure EL0} \\ \neg\text{TRCVICTLR.EXLEVEL\_S\_EL1} & \text{Secure EL1} \\ \neg\text{TRCVICTLR.EXLEVEL\_S\_EL2} & \text{Secure EL2} \\ \neg\text{TRCVICTLR.EXLEVEL\_S\_EL3} & \text{EL3} \\ \neg\text{TRCVICTLR.EXLEVEL\_NS\_EL0} & \text{Non-Secure EL0} \\ \neg\text{TRCVICTLR.EXLEVEL\_NS\_EL1} & \text{Non-Secure EL1} \\ \neg\text{TRCVICTLR.EXLEVEL\_NS\_EL2} & \text{Non-Secure EL2} \end{cases} \tag{D6.16}$$

D6.8.2 ViewInst start/stop function filtering

I_{PJQSC} The ViewInst start/stop function is useful when the requirement is to trace a particular piece of code with all the functions that the piece of code calls.

The ViewInst start/stop function uses the Single Address Comparators and the PE Comparator Inputs to define start points and stop points.

A start point is any of the following:

- The instruction address which matches a Single Address Comparator selected for the ViewInst start/stop function using TRCVISSCTLR.START.
- The instruction address which matches a PE Comparator selected for the ViewInst start/stop function using TRCVIPCSSCTLR.START.

A stop point is any of the following:

- The instruction address which matches a Single Address Comparator selected for the ViewInst start/stop function using TRCVISSCTLR.STOP.
- The instruction address which matches a PE Comparator selected for the ViewInst start/stop function using TRCVIPCSSCTLR.STOP.

Multiple start points can be selected. Multiple stop points can be selected.

R$_{MVDJT}$ When a start point is encountered, the ViewInst start/stop function becomes active for the instruction at the start point.

R$_{CDFBP}$ When a stop point is encountered, the ViewInst start/stop function becomes inactive immediately after the instruction at the stop point.

R$_{LMQPR}$ When the ViewInst start/stop function causes ViewInst to become active, the trace unit traces the instruction at the start address.

R$_{BWXLS}$ When the ViewInst start/stop function causes ViewInst to become inactive, the trace unit traces up to and including the instruction at the stop address.

R$_{XHFYQ}$ While a ViewInst start/stop function start address is the same as a stop address, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

R$_{KCYTR}$ The ViewInst start/stop function part of the ViewInst function is expressed as the following equations:

$$\text{TRCVICTLR.SSSTATUS}_{i+1} = S_i \wedge \neg\text{Stop}_i \tag{D6.17}$$
$$S_i = \text{TRCVICTLR.SSSTATUS}_i \vee \text{Start}_i \tag{D6.18}$$
$$\tag{D6.19}$$

If TRCIDR4.NUMPC == 0b0000 then

$$\text{Start}_i = \sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.START}[n]) \tag{D6.20}$$

$$\text{Stop}_i = \sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.STOP}[n]) \tag{D6.21}$$

If TRCIDR4.NUMPC != 0b0000 then

$$\begin{aligned} \text{Start}_i = &\sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.START}[n]) \\ &\vee \sum_m (\text{PECMP}[m] \wedge \text{TRCVIPCSSCTLR.START}[m]) \end{aligned} \tag{D6.22}$$

$$\begin{aligned} \text{Stop}_i = &\sum_n (\text{SAC}[n] \wedge \text{TRCVISSCTLR.STOP}[n]) \\ &\vee \sum_m (\text{PECMP}[m] \wedge \text{TRCVIPCSSCTLR.STOP}[m]) \end{aligned} \tag{D6.23}$$

R$_{MVFYN}$ The following have no effect on the ViewInst start/stop function:

- *Exceptional occurrences*.
- Execution in Debug state.
- Execution in a Trace Prohibited region.
- A trace unit buffer overflow.

R$_{GRSVY}$ When disabling the trace unit, the ViewInst start/stop function becomes static and retains its state until the trace unit is enabled again.

If required, the state of the ViewInst start/stop function can be changed while the trace unit is disabled.

S_{XMMLP} TRCVICTLR.SSSTATUS, TRCVISSCTLR and TRCVIPCSSCTLR, if implemented, must be programmed with an initial state when the trace unit is programmed before a trace session.

R_{HYLDM} If an implementation makes speculation visible to the trace unit, the ViewInst start/stop function behaves as if no speculation has occurred. That is, when the instruction at a start or stop point is executed speculatively and is later canceled, the ViewInst start/stop function behaves as if the instruction at the start or stop point was not executed.

R_{BZSXR} When the trace unit becomes disabled and there are instructions at start points or stop points which are still speculative, the behavior of the ViewInst start/stop function is IMPLEMENTATION DEFINED and one of the following:

- The ViewInst start/stop function behaves as if the instructions at the start points or stop points were incorrectly speculated. That is, the trace unit behaves as if those start points and stop points did not occur.
- The ViewInst start/stop function behaves as if the instructions at the start points or stop points were correctly speculated. That is, the trace unit updates the state of the ViewInst start/stop function as if those start points and stop points occurred.

R_{HMKSZ} When mis-speculation occurs and the PE returns to a point in execution before the trace unit was enabled, the ViewInst start/stop function reverts to the state it was in when the trace unit became enabled.

R_{CYQZV} When a transaction failure occurs the ViewInst start/stop function reverts back to the state to point immediately after the TSTART instruction for the outer transaction.

This is the value of TRCVICTLR.SSSTATUS$_i$ for the instruction block that contains the TSTART instruction for the outer transaction.

R_{ZBTPF} When a transaction failure causes the PE to return to a point in execution before the trace unit was enabled, the ViewInst start/stop function reverts to the state it was in when the trace unit became enabled.

R_{LRBDC} When the trace unit becomes disabled and the PE is executing in Transactional state, the behavior of the ViewInst start/stop function is IMPLEMENTATION DEFINED and one of the following:

- The ViewInst start/stop function behaves as if the transaction failed. That is, the trace unit behaves as if those start points and stop points did not occur.
- The ViewInst start/stop function behaves as if the transaction was successful. That is, the trace unit updates the state of the ViewInst start/stop function as if those start points and stop points occurred.

R_{FFZDC} When tracing becomes prohibited and the PE is executing in Transactional state, the behavior of the ViewInst start/stop function is IMPLEMENTATION DEFINED and one of the following:

- The ViewInst start/stop function behaves as if the transaction failed. That is, the trace unit behaves as if those start points and stop points did not occur.
- The ViewInst start/stop function behaves as if the transaction was successful. That is, the trace unit behaves as if those start points and stop points did occur.
- The ViewInst start/stop function uses the real resolution of the transaction, when that resolution is eventually known.

R_{PBFMY} When the state of the ViewInst start/stop function is changed by anything other than a direct write to TRCVICTLR, the PE considers the write to be an indirect write to TRCVICTLR.SSSTATUS.

Note

In many common usage scenarios, entry to a Trace Prohibited region or disabling of the trace unit does not occur while in a transaction.

Instruction blocks

R_{PTTKL} When an instruction block that contains instructions at ViewInst start points and no instructions at ViewInst stop points is executed, the ViewInst start/stop function remains active for the entire instruction block.

R_{WJLMR}   While the ViewInst start/stop function is active, when an instruction block is executed that contains at least one ViewInst stop address and no ViewInst start addresses, the ViewInst start/stop function remains active for the whole instruction block and becomes inactive for the next instruction block, unless the next instruction block contains a ViewInst start address.

R_{SMGJZ}   When an instruction block that contains at least one instruction at a ViewInst start point and at least one instruction at a ViewInst stop point is executed, the ViewInst start/stop function obeys the order of the start and stop points in the block, with the following consequences:

- The ViewInst start/stop function is active for the whole of the instruction block.
- When the final instruction in the block at a ViewInst start or stop point is at a ViewInst start point, the ViewInst start/stop function is active for the next instruction block.
- When the final instruction in the block at a ViewInst start or stop point is at a ViewInst stop point, the ViewInst start/stop function is inactive for the next instruction block, unless the next block contains an instruction at a new ViewInst start point.

R_{SFXZB}   The trace analyzer ensures that for all of the *Single Address Comparators* (SACs) selected for ViewInst start points or stop points, any *Single Address Comparator* (SAC) programmed with a lower address than another SAC is a lower-numbered SAC than the other SAC. That is, the SACs contain addresses in address order.

I_{RYPMM}   While the SACs selected for ViewInst do not contain addresses in address order, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

R_{ZLDKC}   The trace analyzer ensures that for all of the PE Comparator Inputs selected for ViewInst start points or stop points, any PE comparator programmed with a lower address than another PE comparator is a lower-numbered PE comparator than the other PE comparator. That is, the PE comparators contain addresses in address order.

I_{CKTWT}   While the PE Comparator Inputs selected for ViewInst do not contain addresses in address order, the behavior of the ViewInst start/stop function is UNPREDICTABLE.

Note

If more than one instruction Address Comparator is programmed with the same instruction address, then programming one or more of those comparators as start comparators, and one or more as stop comparators, results in the following CONSTRAINED UNPREDICTABLE behavior of the ViewInst start/stop function:

- The ViewInst start/stop function is either active or inactive for the instruction at that address.
- The ViewInst start/stop function is either active or inactive after that instruction.

D6.8.3   ViewInst include/exclude function filtering

I_{NDYFG}   The ViewInst include/exclude function is useful if:

- Specific ranges of instructions are required to be included in the trace.
- Specific ranges of instructions are required to be excluded from the trace.
- A combination of including and excluding instruction ranges is required.

I_{LDDGL}   The ViewInst include/exclude function is comprised of two functions:

ViewInst include function	Includes one or more instruction address ranges
ViewInst exclude function	Excludes one or more instruction address ranges

There are between zero and eight instruction Address Range Comparators available for the ViewInst include/exclude

---

function. Some of these comparators can be selected for the ViewInst include function, and some for the ViewInst exclude function.

S_{VNWYT} For example, if all instructions in the address range from `0x00` to `0x2C` are required, but no other instructions are required, an Address Range Comparator can be selected for the ViewInst include function and be programmed with these two addresses. All instructions that are in this address range, including those at the start and end addresses, are traced.

I_{NKLJR} The ViewInst include/exclude function differs from the ViewInst start/stop function in the following ways:

- When the ViewInst start/stop function is used, the trace unit starts tracing on a specified start instruction address and stops tracing on a specified stop instruction address. However, if execution branches or jumps to an address between the start and stop points, without first accessing the instruction at the start address, then the instruction that it has branched or jumped to is not traced. Instructions in the start/stop range are only traced if the instruction at the start address is accessed, so that the trace unit is triggered to start tracing. When triggered, and as execution continues sequentially towards the stop address, all functions that the piece of code calls are traced.

- When the ViewInst include/exclude function is used, for example by programming an Address Range Comparator with an include address range, then if execution branches or jumps to any instruction address anywhere in that range, that instruction is always traced. This is true regardless of whether the instruction at the start address has been accessed or not.

In addition, unlike the ViewInst start/stop function, as program execution continues through the address range towards the end address, no functions that the piece of code calls are traced.

R_{SKZHH} The ViewInst include/exclude function part of the ViewInst function is expressed as the following equations:

$$I_i = \text{Include}_i \wedge \neg\text{Exclude}_i \tag{D6.24}$$

$$\text{Include}_i = \sum_n (\text{ARC}[n] \wedge \text{TRCVIIECTLR.INCLUDE}[n]) \vee \prod_n \neg\text{TRCVIIECTLR.INCLUDE}[n] \tag{D6.25}$$

$$\text{Exclude}_i = \sum_n (\text{ARC}[n] \wedge \text{TRCVIIECTLR.EXCLUDE}[n]) \tag{D6.26}$$

D6.8.3.1 Instruction blocks

R_{RNPWD} When an instruction in an instruction block is included to be traced by the ViewInst include/exclude function, the ViewInst include/exclude function includes all of the instruction block.

R_{PLCQJ} When an instruction block contains at least one instruction excluded by the ViewInst include/exclude function, and only when all the instructions in the instruction block are excluded, the ViewInst include/exclude function excludes the instruction block.

I_{LBNVM} If a block of instructions is not entirely covered by at least one individual *Address Range Comparator* (ARC) selected by TRCVIIECTLR.EXCLUDE, it is CONSTRAINED UNPREDICTABLE whether the block is excluded or not. This applies even if other *Address Range Comparators* (ARCs) selected by TRCVIIECTLR.EXCLUDE cover the rest of the block.

D6.8.4 Guidelines for interpreting the ViewInst function result

I_{TCGMV} The result of the ViewInst function is either:

High	Indicates that instructions being executed must be traced

Low	It is expected that instructions being executed are not traced

If it is expected that instructions being executed are not traced, then there are occasions when it is permitted to trace some of those instructions. This section provides guidelines for when it is permitted to trace instructions that ViewInst indicates are not traced.

D6.8.4.1   When ViewInst transitions from low to high

I_{GMYYC} If execution occurs while ViewInst is low, it is permitted for a trace unit to trace instructions in certain circumstances. See D6.8.4.2 *Occasions when tracing instructions when ViewInst is low is permitted*.

I_{RVCQT} Tracing of instructions is permitted while ViewInst is low, but if no instructions or *Exceptional occurrences* that occur are traced, then there is a discontinuity in the trace. When a discontinuity in the trace occurs, when ViewInst becomes high, a *Trace On element* must be generated.

I_{YXGLK} Any instructions that are executed while ViewInst is high must be traced.

D6.8.4.2   Occasions when tracing instructions when ViewInst is low is permitted

I_{VFYXG} ETE permits tracing of instructions when ViewInst is low, in the following scenarios:

- When the instruction that ViewInst indicates is not to be traced is in the same instruction block as an instruction that ViewInst indicates must be traced. This is because the only way to trace the instruction that must be traced is to trace the whole instruction block.

- When the instruction that ViewInst indicates is not to be traced is in an instruction block that precedes or follows an instruction block containing an instruction that ViewInst indicates must be traced.

An implementation always traces the instruction block that contains an instruction that must be traced. However, additional blocks of instructions might also be traced. This is more likely to occur when many instructions are executed in close proximity.

I_{CDLHB} Except for the scenarios that are mentioned, if the ViewInst function indicates that an instruction is not to be traced, then in general it is expected that it is not traced.
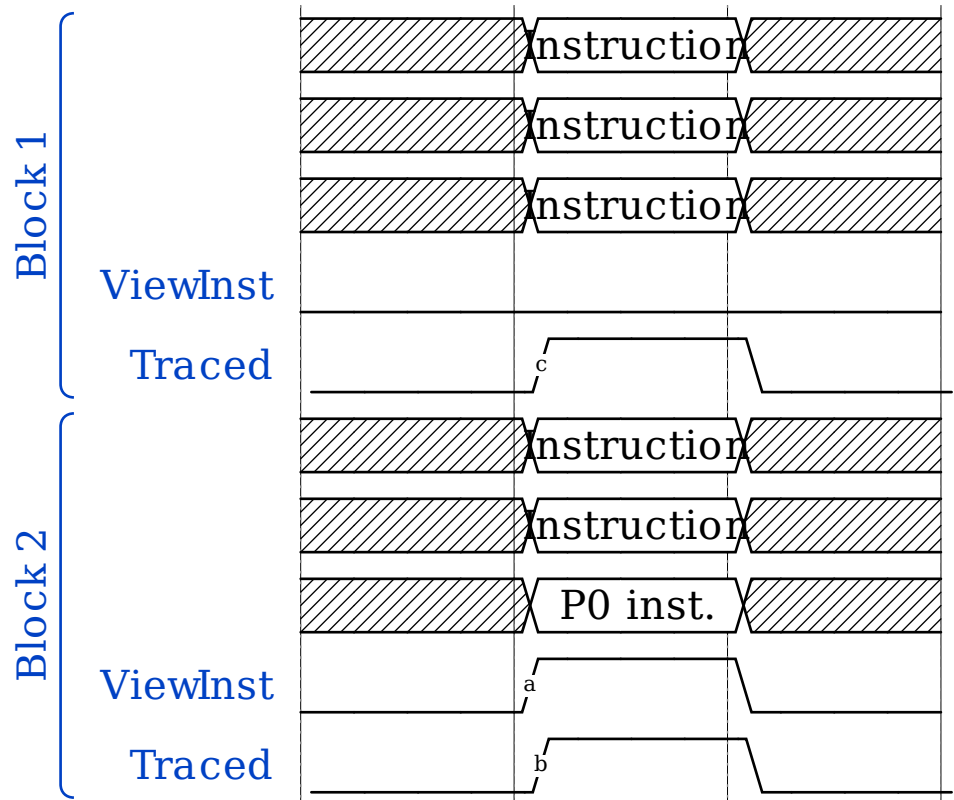
Figure D6.5: Example of close proximity

I_{FJMHL} In the above diagram the instruction block 1 is in execution order before instruction block 2. The ViewInst calculation for the second block returns true, as indicated by the transition labeled (a). As ViewInst is true for this instruction block then all the instructions in this block must be traced, as indicated by the transition labeled (b). Instruction block 1 might be traced as it is in the same PE cycle as block 2, as indicated by the transition labeled (c).

D6.8.5   Rules for tracing Exceptional occurrences

R_{CGVJD} When an *Exceptional occurrence* occurs, the *Exceptional occurrence* does not affect the comparators used by the ViewInst function, and none of the comparators used by the ViewInst function match.

I_{VFDMQ} The comparators used by the ViewInst function include the following:

- Single Address Comparators.
- Address Range Comparators.
- Context Identifier Comparators.
- Virtual Context Identifier Comparators.

I_{VFNZR} When an *Exception element* is traced, it might indicate execution of instructions up to a specified address. These instructions might have an impact on the comparators, but the *Exceptional occurrence* itself does not.

This means that when an *Exceptional occurrence* occurs, the ViewInst function does not indicate whether the *Exceptional occurrence* must be traced. However, it is useful to trace *Exceptional occurrences*, to determine why execution has departed from the normal program flow.

I_{BVGXZ} When an instruction executes or *Exceptional occurrence* occurs outside a Trace Prohibited region, the trace unit remembers whether the instruction or *Exceptional occurrence* was traced. The trace unit performs indirect writes to TRCRSR.TA to store this state. When an *Exceptional occurrence* occurs, the trace unit uses TRCRSR.TA to determine whether to trace the *Exceptional occurrence*.

R_{BFSWZ} When an instruction executes or *Exceptional occurrence* occurs outside a Trace Prohibited region and the instruction or *Exceptional occurrence* is traced, TRCRSR.TA is set to 0b1.

R_{MLTTK} When an instruction executes or *Exceptional occurrence* occurs outside a Trace Prohibited region and the instruction or *Exceptional occurrence* is not traced, TRCRSR.TA is set to 0b0.

R_{CJTJM} When an instruction or *Exceptional occurrence* is canceled, TRCRSR.TA is set to the value of TRCRSR.TA immediately before the canceled instruction or *Exceptional occurrence*.

R_{DPMBQ} When an *Exceptional occurrence* occurs and TRCRSR.TA is 0b1, the *Exceptional occurrence* is traced.

R_{BJQDP} While any of the following are true, TRCRSR.TA is unchanged by any execution:

- The PE is in Debug state.
- The PE is in a Trace Prohibited region.

R_{QZPJD} When a trace unit buffer overflow occurs, the behavior of TRCRSR.TA is IMPLEMENTATION DEFINED and is one of the following:

- TRCRSR.TA is set to 0b0.
- TRCRSR.TA is set to the value of TRCRSR.TA for the most recent instruction or *Exceptional occurrence* before the trace unit buffer overflow occurred.

D6.8.6 Forced tracing of Exceptional occurrences

I_{MFQND} The trace unit can be programmed so that it always traces certain *Exceptional occurrences*, regardless of whether the instruction or *Exceptional occurrence* immediately before the *Exceptional occurrence* must be traced. This option is enabled by setting either or both:

- TRCVICTLR.TRCERR to 0b1. This forces the trace unit to trace System Error exceptions regardless of the value of ViewInst.
- TRCVICTLR.TRCRESET to 0b1. This forces the trace unit to trace PE Resets regardless of the value of ViewInst.

R_{SJXYS} While the PE is executing in a Trace Prohibited region, forced tracing of System Error exceptions is inactive.

R_{VLNMN} While the PE is not executing a Trace Prohibited region and forced tracing of System Error exceptions is enabled, forced tracing of System Error exceptions is active.

R_{LTLBB} While forced tracing of System Error exceptions is active, when a System Error exception occurs, the trace unit generates an *Exception element* indicating a System Error exception, regardless of the value of ViewInst.

R_{NCXJN} While the PE is executing in a Trace Prohibited region, forced tracing of PE Resets is inactive, regardless of whether the PE Reset causes the PE to leave a Trace Prohibited region or not.

R_{NQJNL} While the PE is not executing in a Trace Prohibited region, while forced tracing of PE Resets is enabled, forced tracing of PE Resets is active.

R_{GPKSH} While forced tracing of PE Resets is active, when a PE Reset occurs, the trace unit generates an *Exception element* indicating a PE Reset, regardless of the value of ViewInst.

R_{BBBBT} While tracing is inactive, before an *Exception element* is generated due to forced tracing of either a PE Reset of a System Error exception, the trace unit generates a *Trace On element* and then a *Target Address element*.

I_{LXLKS} When an *Exception element* is generated as a result of forced tracing, the *Trace On element* generated before the *Exception element* indicates that tracing becomes active, and the *Target Address element* indicates where tracing becomes active.

R_{NZNDM} When a System Error exception occurs and TRCRSR.TA is `0b0` and the exception is traced because forced tracing of System Error exceptions is enabled, then it is IMPLEMENTATION DEFINED whether TRCRSR.TA is set to `0b1` or remains at `0b0`.

R_{YFKGM} When a PE Reset occurs and TRCRSR.TA is `0b0` and the PE Reset is traced because forced tracing of PE Resets is enabled, then it is IMPLEMENTATION DEFINED whether TRCRSR.TA is set to `0b1` or remains at `0b0`.

I_{GWHRM} In scenarios where a System Error exception occurs at approximately the same time as an exit from a Trace Prohibited region, after all execution inside the Trace Prohibited region and before any instruction execution outside the Trace Prohibited region, it is UNPREDICTABLE whether the System Error exception is considered to have occurred inside or outside the Trace Prohibited region. It is also UNPREDICTABLE whether the forced tracing of System Error exceptions is active for this exception.

     These scenarios do not include scenarios where the System Error exception caused the exit from a Trace Prohibited region, because the System Error exception occurred inside the Trace Prohibited region.

I_{JGVSY} In scenarios where a System Error exception occurs at approximately the same time as an entry to a Trace Prohibited region, after all execution before the Trace Prohibited region and before any instruction execution inside the Trace Prohibited region, it is UNPREDICTABLE whether the System Error exception is considered to have occurred inside or outside the Trace Prohibited region. It is also UNPREDICTABLE whether the forced tracing of System Error exceptions is active for this exception.

     These scenarios do not include scenarios where the System Error exception caused the entry to a Trace Prohibited region, because the System Error exception occurred outside the Trace Prohibited region.

R_{TSVKN} When a System Error exception occurs immediately after the PE exits a Trace Prohibited region and the System Error exception is traced, the preferred exception return address in the *Exception element* indicating the System Error exception does not include information about the Trace Prohibited region.

I_{PXJWM} In scenarios where a PE Reset occurs at approximately the same time as an exit from a Trace Prohibited region, after all execution inside the Trace Prohibited region and before any instruction execution outside the Trace Prohibited region, it is UNPREDICTABLE whether the PE Reset is considered to have occurred inside or outside the Trace Prohibited region. It is UNPREDICTABLE whether the forced tracing of PE Resets is active for this PE Reset.

     These scenarios do not include scenarios where the PE Reset caused the exit from a Trace Prohibited region, because the PE Reset occurred inside the Trace Prohibited region.

I_{JKQHF} In scenarios where a PE Reset occurs at approximately the same time as an entry to a Trace Prohibited region, after all execution before the Trace Prohibited region and before any instruction execution inside the Trace Prohibited region, it is UNPREDICTABLE whether the PE Reset is considered to have occurred inside or outside the Trace Prohibited region. It is UNPREDICTABLE whether the forced tracing of PE Resets is active for this PE Reset.

     These scenarios do not include scenarios where the PE Reset caused the entry to a Trace Prohibited region, because the PE Reset occurred outside the Trace Prohibited region.

R_{NRNFS} When a PE Reset occurs immediately after the PE exits a Trace Prohibited region and the PE Reset is traced, the preferred exception return address in the *Exception element* indicating the PE Reset does not include information about the Trace Prohibited region.

D6.9 Element Generation

D6.9.1 Trace Info Element Generation

R_{TFQRM} When a trace protocol synchronization request is serviced, the trace unit generates a *Trace Info element*.

Note

There is no requirement to generate a new *Trace Info element* every time that ViewInst becomes active. This is because, despite the discontinuity in the trace that is caused by the filtering, the programming of the trace remains the same.

R_{WJJNK} While the PE is in Transactional state and the trace unit has previously generated a *Transaction Start element* for this transaction, when a *Trace Info element* is generated, the trace unit sets the Transactional state indicator in the *Trace Info element* to 0b1.

R_{WMXVM} While the PE is not in Transactional state, or the PE is in Transactional state but the trace unit has not generated a *Transaction Start element* for this transaction, when a *Trace Info element* is generated, the trace unit sets the Transactional state indicator in the *Trace Info element* to 0b0.

R_{QMTSR} When the trace unit generates the first *Trace Info element* after an *Overflow element*, the Transactional state indicator is set to 0b0.

R_{CRPJZ} When an *Overflow element* is generated, before any subsequent *P0 elements* indicating execution in Transactional state are traced, the trace unit generates a new *Transaction Start element*, even if a *Transaction Start element* has previously been traced for this transaction prior to the *Overflow element*.

D6.9.2 Atom Element

R_{XCJGD} When a *P0 instruction* is taken, the trace unit generates one of the following:

- An E *Atom element*.
- A *Source Address element*.

R_{SRYKV} When a *P0 instruction* is not taken, the trace unit generates one of the following:

- An N *Atom element*.
- Nothing.

R_{TZZRH} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, for all future not taken *P0 instructions* until the next taken *P0 instruction* or *Exceptional occurrence*, the trace unit does not generate an N *Atom element*.

R_{NZTGQ} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, when an *Exceptional occurrence* occurs before the next taken *P0 instruction*, the trace unit generates an *Exception element*.

R_{FWCQR} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, when no *Exceptional occurrence* occurs before the next taken *P0 instruction*, the trace unit generates a *Source Address element* for the next taken *P0 instruction*.

R_{NTMMM} When a *P0 instruction* is not taken and the trace unit does not generate an N *Atom element*, and the *P0 instruction* is subsequently mispredicted, the trace unit generates a *Source Address element* and does not generate a *Mispredict element*.

R_{ZRYPK} The trace unit generates *Atom elements* in the program order in which they occur, and the trace protocol encode and decode process maintains this order.

I_{TGQNB}   For conditional branch instructions, an E *Atom element* indicates that the instruction passed its condition code check, and an N *Atom element* indicates that the instruction failed its condition code check, although a trace unit might use a *Mispredict element* to modify the *Atom element*.

I_{BXBNQ}   The trace unit might trace unconditional *P0 instructions* using an E *Atom element* or an N *Atom element*.

R_{RNYNV}   When an unconditional *P0 instruction* is traced using an N *Atom element*, the trace unit generates either a *Mispredict element* or a *Cancel element* to correct the N *Atom element*.

R_{LMDZV}   When an ISB instruction does not pass the condition code check, and the ISB instruction does not perform a Context synchronization event, the trace unit treats the ISB instruction as a not taken instruction.

R_{NZPLB}   When an ISB instruction does not pass the condition code check, and the ISB instruction performs a Context synchronization event, the trace unit treats the ISB instruction as a taken instruction.

R_{CPFBS}   When an ISB instruction passes the condition code check, the trace unit treats the ISB instruction as a taken instruction.

Note

For an ISB instruction, a trace analyzer must not infer the value of the PSTATE condition flags from an *Atom element*.

R_{QBGXJ}   It is IMPLEMENTATION DEFINED whether the trace unit classifies WFI, WFIT, WFE, and WFET instructions as *P0 instructions*. When WFI, WFIT, WFE, and WFET are classified as *P0 instructions*, execution of these instructions generates an *Atom element*. See D6.2.4 *Low-power state* and TRCIDR2.WFXMODE.

R_{HJHHV}   When WFI, WFIT, WFE, and WFET instructions are classified as *P0 instructions* and a conditional WFE or WFI instruction is executed, if the instruction passes its condition code check then an E *Atom element* is generated.

R_{BMXDT}   When WFI, WFIT, WFE, and WFET instructions are classified as *P0 instructions* and a conditional WFE or WFI instruction is executed, if the instruction fails its condition code check then either an E *Atom element* or an N *Atom element* is generated.

Note

For a WFI, WFIT, WFE, or WFET instruction which is classified as a *P0 instruction*, a trace analyzer must not infer the value of the PSTATE condition flags from an E *Atom element*.

I_{PZRCT}   *P0 instructions* that fail or are predicted to fail their condition code check either generate an Undefined Instruction exception or are executed as a NOP, if the instruction is also UNDEFINED.

R_{YCRVD}   When a *P0 instruction* fails or is predicted to fail its condition code check, and the *P0 instruction* is executed as a NOP, the trace unit generates an N *Atom element* for the *P0 instruction*.

R_{TSQGH}   When a *P0 instruction* fails or is predicted to fail its condition code check, and the *P0 instruction* generates an Undefined Instruction exception, the trace unit does not generate an *Atom element* for the instruction and generates an *Exception element* instead. The preferred exception return address for the generated *Exception element* is the undefined instruction, which indicates that the instruction did not execute.

R_{NQPPX}   The trace unit generates all *Atom elements* speculatively, and explicitly resolves or cancels each *Atom element* by generating *Commit elements* or *Cancel elements*.

I_{NGJYB}   A trace analyzer can infer execution from an *Atom element*, but only after the *Atom element* has been resolved by a *Commit element*.

S_{MFDNZ}   For taken direct *P0 instructions*, a trace analyzer must infer the target address and instruction set of the instruction from the opcode in the program image.

S_{TJSRY}   If a taken direct *P0 instruction* is from a branch broadcasting region, the trace analyzer does not need to infer the target address and instruction set because this is explicitly traced using a *Target Address element*.

D6.9.3   Exception Element

R_{QHRVX}   When an *Exceptional occurrence* occurs, if the *Exceptional occurrence* is required to be traced, the trace unit generates an *Exception element*.

R_{LYYMG}   The trace unit generates *Exception elements* in program order relative to other *P0 elements*.

I_{PCMYT}   To be consistent with the rules for generating *Target Address elements*, under the following scenarios the trace unit must generate a *Target Address element* before an *Exception element*, unless the *Target Address element* would be removed due to a return stack match:

- The *Exceptional occurrence* is taken from the target of a taken indirect *P0 instruction*.
- The *Exceptional occurrence* is taken from the target of a taken direct *P0 instruction* in a branch broadcasting region.
- The *Exceptional occurrence* is taken from the target of another *Exception element*.

R_{XGXKK}   When an *Exceptional occurrence* occurs, if the *Context information* changes at the target of the *P0 element* preceding the *Exceptional occurrence*, then the trace unit generates a *Context element* before the *Exception element*. The *Context element* provides *Context information* about the address and context where the *Exceptional occurrence* was taken from.

I_{CMRCN}   An invalid address is one where bits [63:P] are not all zeros or all ones, where P is defined as the maximum virtual address size supported by the PE.

R_{RJCBT}   When the PE attempts to execute an instruction at an invalid address and the trace unit generates an *Exception element*, the preferred exception return address in the *Exception element* indicates one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

I_{GYJKB}   Arm recommends that when the PE attempts to execute an instruction at an invalid address and the trace unit generates an *Exception element*, the preferred exception return address in the *Exception element* indicates the full 64-bit invalid address.

D6.9.4   Source Address Element

R_{DCHPQ}   When a *P0 instruction* which must be traced is not taken and the trace unit does not generate an N *Atom element*, then when a subsequent *P0 instruction* is taken, the trace unit generates a *Source Address element*.

I_{KDTRV}   A trace unit can generate a *Source Address element* to imply that at least one instruction has been executed, including a taken *P0 instruction*.

R_{WVBQW}   When the trace unit generates a *Source Address element* to imply that a taken *P0 instruction* has been executed, the address associated with the *Source Address element* is the virtual address of the taken *P0 instruction*.

D6.9.5   Q Element

R_{FZTZP}   A trace unit can generate a *Q element* to imply that at least one instruction has been executed, possibly including *P0 instructions*.

R_{WHLPS}   When a *Q element* is generated, the trace unit generates a *Target Address element* that indicates where execution is to continue after all the instructions that are implied by the *Q element* have been executed.

R_{MNWCK}   When a *Q element* is generated and the last instruction implied by the *Q element* is a *P0 instruction*, the trace unit generates a *Target Address element* that indicates the target of the *P0 instruction*.

R_{BJLRS}   When a *Q element* is generated and the last instruction implied by the *Q element* is not a *P0 instruction*, the trace unit generates a *Target Address element* that indicates the instruction address immediately following the last instruction that is implied by the *Q element*.

R_{FPHFM}   When the PE leaves a region where *Q elements* are permitted, either by a *P0 instruction* or by sequential execution out of the region, and a *Q element* implies the execution of the last instruction in the region, the *Q element* does not imply any more instructions after the last instruction in the region.

R_{DTWYZ}   When the PE enters a region where *Q elements* are permitted, either by a *P0 instruction* or by an *Exceptional occurrence*, the trace unit traces the *P0 instruction* or *Exceptional occurrence* using elements other than *Q elements*.

> **Note**
>
> Although the trace unit does not trace with *Q elements* a *P0 instruction* or *Exceptional occurrence* that causes the PE to enter a region where *Q elements* are permitted, any subsequent instructions in the region where *Q elements* are permitted might be traced using *Q elements*.

I_{YRZWY}   When the PE enters by sequential execution a region where *Q elements* are permitted, any instructions that are executed since the last *P0 element* outside the permitted region might be traced using a *Q element*. These instructions can always be inferred unambiguously from the *Q element*.

R_{SJSGX}   When the PE enters by sequential execution a region where *Q elements* are permitted, and *P0 instructions* executed since the last *P0 element* outside the permitted region are traced by a *Q element*, the *Q element* does not indicate execution of any *P0 instructions* outside the permitted region.

D6.9.6 Event Element

R_{WJPTB}   The trace unit generates *Event elements* independently of ViewInst.

R_{KKLYB}   While TRCEVENTCTL1R.INSTEN<n> is 0b1 and the resource event selected by TRCEVENTCTL0R.EVENT<n> is active, while trace generation is operative, the trace unit generates an *Event element* <n> on each PE clock cycle.

R_{SPYTT}   When an *Event element* is generated between two *P0 elements* or at the same time as a *P0 element* that follows another, the trace unit inserts the *Event element* after the first *P0 element* but before the *P0 element* that is an IMPLEMENTATION DEFINED number of *P0 elements* after the first *P0 element*.

I_{XLJKP}   Arm recommends that the IMPLEMENTATION DEFINED number of *P0 elements* is less than or equal to the number of *P0 elements* the PE can process simultaneously.

R_{SHYMY}   While trace generation is inoperative due to a trace unit buffer overflow, when a programmed ETEEvent <n> occurs, the trace unit generates at least one *Event element* <n> before it generates the *Overflow element* corresponding to the trace unit buffer overflow.

D6.9.7 Cancel Element Generation

R_{HYCXR}   When one or more *P0 elements* are canceled, the trace unit generates a *Cancel element*.

R_{GKDYR}   The trace unit generates *Cancel elements* in execution order relative to *P0 elements*.

R_{KDTVX}   When a *Cancel element* causes execution to return to a point in the program flow that is not adjacent to a *P0 instruction*, the trace unit generates an *Exception element* that indicates which instructions were executed up to that point in the program flow before it generates any *P0 elements*.

D6.9.8 Commit Element Generation

R$_{\text{QNLYJ}}$ When one or more traced *P0 elements* are resolved for execution, the trace unit generates a *Commit element*.

I$_{\text{MDSCN}}$ An *Atom element* might be corrected using a *Mispredict element* after it has been resolved.

R$_{\text{MNXTW}}$ The trace unit never generates more speculative *P0 elements* than the maximum speculation depth of the trace unit.

R$_{\text{XXFBC}}$ When trace generation becomes inoperative due to the trace unit being disabled, the trace unit outputs any *Commit elements* which have not been output.

I$_{\text{BNLZT}}$ If cycle counting is enabled some *Commit elements* have *Cycle Count elements* associated with them, that provide counts of processor clock cycles. The cycle count values given in *Cycle Count elements* can be used to obtain a cumulative count.

R$_{\text{ZHWDD}}$ *Commit elements* with associated *Cycle Count elements* cannot be merged with later *Commit elements*.

I$_{\text{PSFTD}}$ For more information, see D2.4.1 *Cycle Count Element*.

D6.9.9 Transaction Start

R$_{\text{DGRTZ}}$ When the PE enters an outer transaction, before the first instruction is traced, the trace unit generates a *Transaction Start element*.

I$_{\text{QLXNC}}$ A *Transaction Start element* is not required for each *Trace On element* if the instructions are all part of the same outer transaction.

R$_{\text{TWNQP}}$ When the PE leaves Transactional state and a *Transaction Start element* was generated for the transaction, the trace unit traces the result of the transaction using one of the following:

- A *Transaction Commit element*, if the transaction was successful.
- A *Transaction Failure element*, if the transaction failed.
- A *Cancel element* which cancels the *Transaction Start element*.

I$_{\text{GWZDH}}$ The trace element stream only indicates that the PE is in Transactional state. It does not indicate the transactional nesting depth.

D6.9.10 Transaction Commit

R$_{\text{BGMKL}}$ When the PE exits Transactional state successfully, and a *Transaction Start element* was generated for the transaction, the trace unit generates a *Transaction Commit element*.

R$_{\text{PCKKS}}$ When a *Transaction Commit element* is generated, the trace unit traces the *Transaction Commit element* after the *P0 element* which is generated before the TCOMMIT instruction, and before the next *Transaction Start element* is traced.

I$_{\text{CQLFV}}$ Arm recommends that the *Transaction Commit element* is generated and output as soon as possible after the PE leaves Transactional state.

> **Note**
>
> These rules mean that a *Transaction Commit element* is permitted to be output later than the *P0 element* which implies execution of the TCOMMIT instruction.
>
> The TCOMMIT instruction is not a *P0 instruction*. This means that the *Transaction Commit element* might be traced before the *P0 element* which implies execution of the TCOMMIT instruction.

D6.9.11 Transaction Failure

R_{MHBCG} When a transaction failure occurs, and a *Transaction Start element* was generated for the transaction, the trace unit generates a *Transaction Failure element*.

R_{XQSPC} When the PE enters a Trace Prohibited region and is in Transactional state, and a *Transaction Start element* was generated for the transaction, the trace unit generates a *Transaction Failure element*.

R_{ZJXHP} When the trace unit becomes disabled and the PE is in Transactional state, and a *Transaction Start element* was generated for the transaction, the trace unit generates a *Transaction Failure element*.

R_{YTKLN} When a trace unit buffer overflow occurs and the PE is in Transactional state, and a *Transaction Start element* was generated for the transaction, the trace unit generates a *Transaction Failure element*.

I_{DLKJR} A *Transaction Failure element* is encoded as an Exception packet with a type of Transaction Failure.

I_{FBFDB} When a *Transaction Failure element* is generated, the following behavior applies:

- The target address and target context of the previous *P0 element* might be UNKNOWN.
- If there are no *P0 elements* between a *Trace On element* and the *Transaction Failure element*, the initial address and context after the previous *Trace On element* might be UNKNOWN.

R_{QWGXL} When a PE Reset occurs and the PE is in Transactional state, and a *Transaction Start element* was generated for the transaction, the trace unit generates a *Transaction Failure element*.

Note

A *Transaction Failure element* caused by a PE Reset might be traced using any of the following:

- 1. A single Exception packet with TYPE indicating PE Reset. This packet can imply the *Transaction Failure element*.

- 1. An Exception packet with TYPE indicating Transaction Failure.
  2. An Exception packet with TYPE indicating PE Reset, if the PE Reset is required to be traced.

D6.9.12 Context Element

R_{BDFDQ} The trace unit generates a *Context element* in the following situations:

- While tracing is active, when any of the *Context information* changes, prior to any *P0 element* which indicates execution from the new context.
- After a *Trace Info element* is generated due to a non-periodic trace protocol synchronization request, and prior to any *P0 element*.
- After a *Trace Info element* is generated due to a periodic trace protocol synchronization request.
- When mis-speculation results in an incorrect *Context element* being output, prior to any *P0 element* which indicates execution from the correct context.

R_{JNXJT} While Virtual context identifier tracing is enabled and TRFCR_EL2.CX disallows the tracing of the Virtual context identifier, when the trace unit generates a *Context element*, the Virtual context identifier in the *Context element* has the value 0x0.

I_{TBJPL} A *Context element* might also be output at other points, which might include after all Context synchronization events, or at any other point at which the *Context information* changes.

R_{MKKZN} If the highest implemented Exception level is using AArch64, the Context identifier value is the value of CONTEXTIDR_EL1.

I_{WXVHT} Some of the *Context information* might change at points other than at Context synchronization events. These changes occur when system instructions are used to change a piece of *Context information*, including:

- Writes to the current CONTEXTIDR_EL1.
- Writes to the CONTEXTIDR_EL2.

R_{HXNXF}   When a system instruction writes to a system register and the *Context information* changes, the trace unit generates a *Context element* containing the new context value, after the *P0 element* prior to the system instruction but before the *P0 element* following a Context synchronization event after the system instruction.

Note

If the *Context element* is output before the first *P0 element* after the system instruction, this might imply that some instructions before the system instruction were executed with the new context. This is acceptable because the code which changes the context is usually executed in a state where it does not matter whether the old or new context values are used.

I_{WTSTB}   If the PE takes an exception after performing a write to a system register that changes the context, but a *P0 element* has not been generated since the write, then a *Context element* indicating the new context is not required to be output before the *Exception element*. This is because no instructions or *Exceptional occurrences* are indicated to have been executed from the new context. A *Context element* indicating the new context must be generated after the *Exception element* if the *Exceptional occurrence* is a Context synchronization event. If the *Exceptional occurrence* changes the context, then the *Context element* must indicate the new context. This might happen if, for example, the Security state changes.

R_{HTYGF}   When a PE Reset occurs, until the relevant PE registers are updated, the trace unit traces the Context identifier and Virtual context identifier as zero.

I_{KTPVJ}   A trace unit is not required to generate a *Context element* if tracing becomes inactive before any instructions are executed in the new context.

I_{QWSVJ}   Additional *Context elements* might be output by a trace unit in some scenarios, but these must only be output where they do no affect the analysis of the trace element stream. Such a scenario might include when a change in the *Context information* is incorrectly speculated and a subsequent *Context element* corrects the value of a previous incorrect *Context element*. Arm recommends that the generation of additional unnecessary *Context elements* is minimized to ensure trace bandwidth is minimized.

D6.9.13 Target Address Element

R_{HLRZJ}   When the trace analyzer cannot infer the address or instruction set from the trace, the trace unit generates a *Target Address element*.

I_{FRTGM}   Occasions when the trace analyzer might not be able to infer the address or instruction set from previous trace include:

- At the target of an indirect *P0 instruction* which is taken.
- At the target of a direct *P0 instruction* which is taken in a branch broadcasting region, see TRCBBCTLR for more information.
- At the target of an *Exceptional occurrence*.
- At the target of an *Transaction Failure element*.
- When mis-speculation occurs and the address cannot be inferred.
- After a *Q element* is generated.

R_{ZRYSN}   When the trace analyzer cannot infer the address or instruction set from the trace, the trace unit generates the resulting *Target Address element* before the next *P0 element*, unless any of the following are true:

- The *Target Address element* can be omitted because of a return stack match.
- Tracing is inactive at the target of the *P0 instruction* or *Exceptional occurrence*.
- A transaction failure occurs and tracing is inactive at the target of the transaction failure.

R_{RGPTK} When non-periodic trace protocol synchronization occurs, the trace unit generates a *Target Address element* after the *Trace Info element* and *Trace On element* corresponding to the non-periodic trace protocol synchronization, and before the next *P0 element* is generated.

R_{MDTYL} When periodic trace protocol synchronization occurs, after the corresponding *Trace Info element* has been generated, the trace unit generates a *Target Address element* containing the address of the target of the most recent *P0 element* before the *Target Address element*.

I_{YMLXQ} When non-periodic trace protocol synchronization occurs, the *Target Address element* does not need to indicate the target of the most recent *P0 element*, since tracing might not become active at the target of a *P0 element*.

I_{GBMWG} When periodic trace protocol synchronization occurs, the *Target Address element* needs to indicate the target of the most recent *P0 element*, since tracing is continuing from that *P0 element*. Furthermore, the *Target Address element* might indicate the target of a *P0 element* from before the *Trace Info element*.

R_{QCSJJ} When a *Trace On element* is generated, the trace unit generates a *Target Address element* before the next *P0 element*.

I_{DTQDH} Typically, a *Target Address element* is required after an *Exception element* to indicate the target of the *Exceptional occurrence*, since a trace analyzer is not usually able to infer the target of an *Exceptional occurrence*.

I_{YHQGL} In some scenarios, an *Exception element* might be generated in the trace where the *Exceptional occurrence* target address is the next sequential instruction from the last instruction before the *Exceptional occurrence*. This behavior depends on many factors and might only occur for IMPLEMENTATION DEFINED *Exceptional occurrences*. If an *Exceptional occurrence* is taken to the next sequential instruction, the trace unit is not required to output a *Target Address element* indicating the target address of the *Exceptional occurrence* because this can be inferred from the previous execution.

I_{GVZJZ} A trace analyzer needs both a *Target Address element* and a *Context element* before it can determine the instruction set in use, because the *Target Address element* provides the instruction set and the *Context element* provides information on whether the PE is in AArch32 or AArch64 state.

R_{RHDMW} When a change of instruction set occurs that switches between AArch32 state and AArch64 state, the trace unit generates a *Context element* indicating the new state.

I_{KZXQW} An invalid address is one where bits [63:P] are not all zeros or all ones, where P is defined as the maximum virtual address size supported by the PE.

R_{VWVWR} When the PE attempts to execute an instruction at an invalid address and the trace unit generates a *Target Address element*, the *Target Address element* indicates one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

I_{YJYFM} Arm recommends that when the PE attempts to execute an instruction at an invalid address and the trace unit generates a *Target Address element*, the *Target Address element* indicates the full 64-bit invalid address.

R_{SBCPK} While tagged addresses are in use, the virtual address in the trace element stream does not include the tag and is the PC value, that is, depending on the state of the PE at the address, bits[63:56] are one of the following:

- The sign-extension of bit[55].
- Zero.

I_{YGGGK} The Translation Control Registers, TCR_ELx, contain the TBI field for controlling whether to ignore the top byte of an address. If the current TBI field is changed from 0b0 to 0b1, and before the next Context synchronization event the PE takes an exception because of an invalid top address byte, the branch target address to the invalid address or the preferred exception return address of the *Exception element* might not contain the full invalid address and might contain the address with the top byte masked. Furthermore, the branch target address might be the invalid address and therefore might be different from the preferred exception return address. Trace analysis tools must be aware that if a branch target address is substantially different from a preferred exception return address which follows, then there might have been a change in the TBI field which caused this large change in address.

R_{HHKGB} When a pointer authentication check fails and an exception is taken from the resulting invalid address, the preferred exception return address is one of the following:

- The full 64-bit invalid address.
- Any other invalid address, with address bits [P-1:0] the same as the full invalid address.

I_{CRSTX} Arm recommends that when a pointer authentication check fails and an exception is taken from the resulting invalid address, the preferred exception return address is the full 64-bit invalid address.

R_{FSWRC} The bottom bits of an address are ignored, in the following way:

- Bits[1:0] of addresses that are used in A64 or A32 instructions are always traced as zero.

- Bit[0] of addresses that are used in T32 instructions is always traced as zero.

I_{MDZJL} Additional *Target Address elements* might be output by a trace unit in some scenarios, but these must only be output where they do not affect the analysis of the trace element stream. These scenarios include, but are not limited to:

- When an instruction address is incorrectly speculated, and a subsequent *Target Address element* corrects the value of a previous incorrect *Target Address element*.

- When an instruction address can be inferred by the trace analyzer, for example for the target of a direct *P0 instruction*, but a *Target Address element* is output anyway with the same address.

Arm recommends that the generation of additional unnecessary *Target Address elements* is minimized to ensure trace bandwidth is minimized.

D6.9.14 Mispredict Element

R_{YJCNT} When the status of the last non-canceled *Atom element* has been changed by the PE, the trace unit generates a *Mispredict element*.

R_{SCKBZ} The trace unit only generates a *Mispredict element* to change the status of an *Atom element*.

I_{XYDNP} A trace unit might generate multiple *Mispredict elements* for the same *Atom element*. A trace analyzer must use each *Mispredict element* to determine the final status of the *Atom element*. For example, if an E *Atom element* has two *Mispredict elements*, the first *Mispredict element* indicates the *Atom element* is an N *Atom element* and the second *Mispredict element* indicates it is an E *Atom element*.

I_{HVWCN} If a PE mispredicts only the target address of a *P0 element* then it does not generate a *Mispredict element*.

The trace unit uses a *Target Address element* to correct the mispredicted target address. When analyzing a *Mispredict element*, any *Target Address elements* between the mispredicted *Atom element* and the *Mispredict element* must be discarded.

D6.9.15 Overflow Element

R_{HRYKY} When a trace unit buffer overflow occurs, after all trace elements that were generated prior to the trace unit buffer overflow are output, the trace unit outputs an *Overflow element*.

R_{RPSPH} When a trace unit buffer overflow occurs, and the trace unit is disabled after recovering from the trace unit buffer overflow, the trace unit outputs the corresponding *Overflow element* before the trace unit becomes idle.

D6.9.16 Timestamp Element

R_{YYWTR} While TRCCONFIGR.TS is `0b1` and any of the following occur, a timestamp request occurs:

- The timestamp resource event occurs, as controlled by TRCTSCTLR.

- The trace unit generates a *Trace Info element*.

- The trace unit recovers from a trace unit buffer overflow.

- When not in a Trace Prohibited region and a Context synchronization event is caused by any of the following:

  - The PE takes an exception.

  - The PE returns from an exception.

  - An `ISB` instruction is executed.

- A trace unit flush is requested.

R_{CKNFV} While TRCCONFIGR.TS is `0b1` and when not in a Trace Prohibited region, a timestamp request might occur when any of the following occur but do not cause a Context synchronization event:

- The PE takes an exception.

- The PE returns from an exception.

- An `ISB` instruction is executed.

R_{NGXNQ} The state of the ViewInst function does not affect whether a timestamp request occurs.

R_{HZSYP} When a timestamp request occurs and ViewInst is inactive, the timestamp request is permitted to be delayed until the first of the following occurs:

- ViewInst becomes active.
- An *Event element* is generated.

I_{WFXVV} There is no requirement for a *Timestamp element* to be generated in the trace element stream on each occasion that ViewInst becomes active.

R_{RMSVV} When a timestamp request occurs and is not ignored, the trace unit generates a *Timestamp element*.

R_{DWCYP} When a timestamp request occurs but the trace unit does not have the capacity to generate the *Timestamp element* immediately, then the generation of the *Timestamp element* is delayed until there is capacity.

I_{TQDHQ} A trace unit might not have the capacity to generate a *Timestamp element* for multiple reasons, including avoiding a trace unit buffer overflow. A delayed *Timestamp element* means that a timestamp value might not be the exact time of the incident that resulted in the timestamp request. A timestamp is only a time indicator inserted in the trace element stream somewhere near the time of the request.

R_{XMJGY} When a timestamp request occurs while in a Trace Prohibited region, then the generation of the *Timestamp element* is delayed until the PE leaves the Trace Prohibited region.

R_{CWHHW} When the first timestamp request occurs after trace generation becomes operative, the trace unit delays generation of the corresponding *Timestamp element* until after the trace unit has generated either a *P0 element* or an *Event element*.

I_{SDPZZ} This is so that the timestamp value can correspond to the most recent of these elements.

R_{SZNMB} A timestamp request is permitted to be ignored if a previous timestamp request has not yet generated a *Timestamp element*, due to a delay in the generation.

R_{DWFTT} A trace unit might ignore the second request of two successive timestamp requests if all of the following are true:

- The second request is not caused as a result of a trace protocol synchronization request.
- No *P0 elements* or *Event elements* have been generated between the two requests.
- The timestamp value would be the same in both *Timestamp elements*.

R_{ZMQLT} While TRCCONFIGR.CCI is `0b1`, each *Timestamp element* contains a cycle count that indicates the number of cycles between the previous *Cycle Count element* and the element with which the Timestamp is associated.

R_{MWJHD} The cycle count associated with a *Timestamp element* is different from the *Cycle Count element* in the following ways:

- The cycle count does not affect the cumulative cycle count.
- The cycle count value can be zero.

I_{HTGCM} When the cycle count associated with a *Timestamp element* is zero, this indicates that no cycles passed between the previous *Cycle Count element* and the element with which the *Timestamp element* is associated.

Note

The cycle count associated with a *Timestamp element* is not a *Cycle Count element*, and therefore has no effect on the cycle counter.

R_{JNSWW} When the trace unit is first enabled, while cycle counting is enabled, when a *Timestamp element* is generated before any *Cycle Count elements*, the *Timestamp element* reports the cycle count as UNKNOWN.

R_{NPYKS} When a *Timestamp element* is generated and the cycle counter has exceeded the maximum supported value, the *Timestamp element* reports the cycle count as UNKNOWN.

D6.9.17 Trace On Element

R_{LMLSN} When an instruction block is traced immediately after an instruction block was not traced or a trace unit buffer overflow occurred, the trace unit generates a Trace On element.

I_{GPQZW} When an *Exception element* indicating a PE Reset is traced, the preferred exception return address is UNKNOWN. Any instructions since the most recent unresolved *P0 element* are not traced. If ViewInst was active for these instructions, this is not considered a gap in the trace element stream and a *Trace On element* is not required.

In some scenarios where mis-speculation occurs or instructions are canceled, after *Cancel elements* have been processed there might be *Trace On elements* in the trace element stream even though no trace discontinuity occurred in the architecturally-executed instruction trace. This typically only occurs when the trace is filtered using the ViewInst function, which causes the *Trace On element* to be inserted.

I_{MHFJB} Trace analyzers must be aware that these additional *Trace On elements* might be traced.

D6.9.18 Cycle Count Element

R_{TYNZR} The cycle counter has an IMPLEMENTATION DEFINED size of between 12 and 20 bits, as indicated by TRCIDR2.CCSIZE. The cycle counter therefore supports values from 1 to 2^{20}-1.

R_{GWQGS} While TRCCONFIGR.CCI is 0b1 and the cycle count is equal to or greater than the value of TRCCCCTLR.THRESHOLD, when a Commit element is generated, a *Cycle Count element* request occurs.

R_{KJXDK} While TRCCCCTLR.THRESHOLD is programmed with a value less than TRCIDR3.CCITMIN, the generation of *Cycle Count elements* is CONSTRAINED UNPREDICTABLE.

R_{GDTBW} When a request for a *Cycle Count element* occurs, one of the following occurs:

- The trace unit generates a *Cycle Count element* immediately and before any future *Commit element*.

- The trace unit delays generation of the *Cycle Count element* until after one or more further *Commit elements* have been generated.

I_{TCFRL} Arm recommends that when a request for a Cycle Count element occurs, the *Cycle Count element* is generated immediately, and that *Cycle Count element* generation is only delayed in rare and non-repetitive circumstances.

R_{BMVKB} When a *Cycle Count element* is generated, the *Cycle Count element* contains the value of the cycle counter at the time the most recent *Commit element* was generated, and the cycle counter is reset to the number of cycles since the most recent *Commit element* was generated.

R_{PFZXK} A value of 0 indicates that the cycle count value is UNKNOWN.

R_{YVWJW} When the cycle counter exceeds the maximum supported value, the cycle count value is UNKNOWN.

R_{YFMWB} When the trace unit becomes enabled, an UNKNOWN cycle count value occurs for the first *Cycle Count element* generated.

R_{HQKWH} When a trace unit buffer overflow occurs, an UNKNOWN cycle count value occurs for the first *Cycle Count element* generated.

I_{PDBDY} The first *Cycle Count element* after the PE clock has been restarted should have an UNKNOWN cycle count.

D6.9.19 Discard Element

R_{WDWGV} When trace generation becomes inoperative and any of the following are true, the trace unit generates a *Discard element*:

  • *P0 elements* have been generated but the trace unit is unable to output the resolution of those *P0 elements*.

  • A *Transaction Start element* has been generated and trace generation becomes inoperative before the transaction either succeeds or fails.

R_{FHQDX} When trace generation becomes inoperative due to the trace unit becoming disabled, and a *Discard element* is generated, the trace unit outputs the *Discard element* after all other elements.

R_{NSMJF} When a PE Reset occurs and any of the following are true, the trace unit generates a *Discard element*:

  • *P0 elements* have been generated but the trace unit is unable to output the resolution of those *P0 elements*.

  • A *Transaction Start element* has been generated and the PE Reset occurs before the transaction either succeeds or fails.

I_{SKJSP} A trace unit might not generate a *Discard element* if no *P0 elements* are speculative.

I_{TGXKV} A trace unit might generate a *Discard element* even if no *P0 elements* are speculative.

R_{CTYFK} When a *Discard element* is generated, all uncommitted *P0 elements* are discarded, that is, the current speculation depth is set to zero.

I_{TYXLL} When a *Discard element* is generated, and a *Transaction Start element* has been traced but the transaction has not succeeded or failed, the trace unit does not indicate the resolution of the transaction.

R_{WXTQS} When a *Discard element* is generated and tracing subsequently becomes operative for the same transaction, the trace unit generates a new *Transaction Start element* before any *P0 elements* are generated for the transaction.

D6.10 Trace unit features

I_{BFHRC} The architecture defines a number of optional and mandatory features that are provided to modify the trace element stream to provide additional information to aid debugging. These features include the following:

- *Q element* regions.
- Branch broadcasting.
- Context identifier tracing.
- Cycle counting.
- Event trace.
- No overflow.
- PE Stalling and overflow avoidance.
- Timestamping.
- Virtual context identifier tracing.

For the optional features, the inclusion of these optional features is indicated in TRCIDR0-TRCIDR13.

D6.10.1 Branch broadcasting

I_{WLPLZ} The branch broadcasting feature forces the trace unit to explicitly trace the target addresses of taken direct *P0 instructions*.

The target addresses are traced using *Target Address elements* in the instruction trace stream.

I_{DCYQT} Branch broadcasting is enabled by performing both of the following actions:

- Setting TRCCONFIGR.BB to `0b1`.

- Programming TRCBBCTLR to specify how branch broadcasting behaves. TRCBBCTLR selects Address Range Comparators to define when branch broadcasting is active, and selects the operating mode of branch broadcasting:

  - Branch broadcasting is active for all instructions inside the selected ranges. This is known as include mode.
  - Branch broadcasting is active for all instructions outside the selected ranges. This is known as exclude mode.

R_{MHYFV} When a direct *P0 instruction* for which branch broadcasting is active is taken, the trace unit generates a *Target Address element* to explicitly trace the target of the instruction, regardless of whether the *P0 instruction* is mispredicted.

R_{VQTVR} While branch broadcasting is enabled, while the return stack is enabled, the trace unit prioritizes branch broadcasting over the return stack, that is, the return stack does not match on the target of any instruction for which branch broadcasting is active.

R_{XSVSX} If TRCBBCTLR is not implemented, while branch broadcasting is enabled, branch broadcasting is active for all instructions.

D6.10.2 Q Regions

I_{XFPKH} *Q elements* are a optional feature, as indicated by TRCIDR0.QSUPP.

I_{FSXRY} The use of *Q elements* must be explicitly enabled if the trace unit is to use them.

I_{CGQZJ} While *Q elements* are enabled, the trace element stream might not contain enough information to determine the complete program flow, because some changes in flow might not be explicitly indicated.

I_{QLVSG} Arm recommends that *Q elements* are only used in cases where generating the full ETE instruction trace element stream might cause the performance of the PE being traced to degrade significantly.

I_{TNPWC} The use of *Q elements* degrades the information that can be extracted from the trace element stream. Arm recommends that *Q element* filtering, as indicated by TRCIDR0.QFILT, is also implemented.

I_{CBKXZ} If TRCQCTLR is implemented, the trace unit supports the ability to control when *Q elements* are permitted in the trace element stream using ARCs. The *Q element* filtering operates in either Exclude mode, or Include mode, selected by TRCQCTLR.MODE.

R_{DSFJZ} If *Q elements* are enabled and *Q element* filtering is in Include mode, the ARCs selected by TRCQCTLR.RANGE define where *Q elements* are permitted.

R_{WXRDY} If *Q elements* are enabled and *Q element* filtering is in Exclude mode, the ARCs selected by TRCQCTLR.RANGE define where *Q elements* are not permitted.

R_{RBPJF} When an instruction block contains at least one instruction where *Q elements* are permitted, the entire instruction block is permitted to generate *Q elements*.

R_{NQSLS} The following equation is calculated for each instruction block and defines when *Q elements* are permitted.

$$
\text{Q\_PERMITTED}_i = \begin{cases} \sum_n \text{ARC}_i[n] \wedge \text{TRCQCTLR.RANGE}[n] & \text{Include mode} \\ \neg \sum_n \text{ARC}_i[n] \wedge \text{TRCQCTLR.RANGE}[n] & \text{Exclude mode} \end{cases} \tag{D6.27}
$$

R_{SGFHP} While TRCCONFIGR.QE indicates that *Q elements* are disabled, the trace unit does not generate any *Q elements*.

R_{CGHSK} While TRCCONFIGR.QE indicates that *Q elements* are disabled, the trace unit is able to generate all of the elements required to trace the instruction sequence.

D6.10.3 Cycle Counting

I_{MKQVC} The use of the cycle counting feature introduces *Cycle Count elements* into the trace element stream to indicate the passage of PE clock cycles.

I_{BMCMB} Counting the number of clock cycles the PE uses to perform a certain function can be useful as a way of measuring program performance, or for profiling the PE.

R_{JMLLY} While cycle counting is enabled, the trace unit outputs Cycle Count packets that contain processor clock cycle count values.

I_{TVCCV} *Cycle Count elements* are associated with *Commit elements*, so that when a *Commit element* is generated, a *Cycle Count element* might also be generated. Whether a *Cycle Count element* is generated when a *Commit element* is generated depends on what cycle count threshold has been specified when programming TRCCCCTLR.THRESHOLD.

When a *Commit element* is generated and the cycle count value is equal to or more than the threshold value, then a *Cycle Count element* is generated and a Cycle Count packet is output. The cycle count value that is contained in that packet is associated with the *Commit element* that triggered it.

R_{BHYWB} While cycle counting is enabled, and when a *Commit element* is generated and the cycle count value is greater than or equal to the threshold value that is programmed in TRCCCCTLR.THRESHOLD, the trace unit generates a *Cycle Count element*.

I_{LFLCZ} Also, because cycle counting is associated with *Commit elements*, a Cycle Count packet might imply the generation of *Commit elements*, and so in addition to the cycle count value, some Cycle Count packets also contain a value for the number of *Commit elements* that the trace unit has generated.

I_{MWQNZ} The value of cycle count that is given in a new Cycle Count packet indicates the number of processor clock cycles between the new *Commit element* that the packet is associated with, and the most recent *Commit element* prior to the new *Commit element* that had a *Cycle Count element* associated with it.

This means that if there is a requirement for a cumulative cycle count total, the cycle count values from the successive Cycle Count packets can be added together to obtain this.

D6.10.4 Timestamping

I_{YVBBW} The trace unit supports Timestamping, where a common global time value is inserted in to the trace stream. These timestamps may be used to correlate between multiple trace streams, including those from other PEs or other sources of trace. These timestamps may be used to determine the passage of time, for analysing performance.

I_{VFWQS} When timestamping is enabled, the trace unit inserts *Timestamp elements* in to the trace stream. Each Timestamping element indicates the time that a recent *P0 element* or *Event element* occurred, and can be used to accurately determine when that element occurred.

I_{YFPDZ} The time value included in *Timestamp elements* is selected by TRFCR_EL1 and TRFCR_EL2 and is one of:

- Physical time, as seen by the generic timers in the PE.
- Virtual time, as seen by the generic timers in the PE.
- An IMPLEMENTATION DEFINED time value, often supplied by a CoreSight system.

I_{CLZKR} The insertion of *Timestamp elements* is controlled by TRCCONFIGR.TS and TRCTSCTLR.

D6.10.5 Stalling the execution of the PE

I_{MPBVQ} The trace unit can be programmed to reduce the likelihood of a trace unit buffer overflow. If the trace unit is configured to support PE stalling, TRCIDR3.STALLCTL indicates that PE stalling is implemented and TRCIDR3.SYSSTALL indicates that PE stalling is permitted, then the execution of the PE can be slowed.

I_{HPFQP} It is permissible that the operation of the PE can be affected by the programming of the trace unit. The amount of intrusion and when stalling occurs is IMPLEMENTATION DEFINED. Additional stalling of the PE execution can be achieved by enabling this feature.

I_{VZSVK} Trace unit stalling of the PE is independent of the operation of the PE.

R_{NVBGS} PE operations which explicitly interact with the trace unit complete independently of the programming of the ability of the trace unit to stall the PE.

R_{SCLVV} The trace unit does not stall the PE while any of the following are true:

- The trace unit is in the Disabled state.
- The PE is executing in a Trace Prohibited region (see D6.6.6 *Trace Prohibited regions*).
- The PE is in Debug State.
- The PE does not allow stalling, that is, TRCIDR3.SYSSTALL == 0b0.
- SelfHostedTraceEnabled()== FALSE and ExternalInvasiveDebugEnabled()== FALSE.
- When TRCSTALLCTLR.ISTALL == 0b0 and TRCSTALLCTLR.NOOVERFLOW == 0b0.
- Trace output is disabled.

R_{RWTYJ} When all of the following are true, the trace unit is permitted to stall the PE:

- Stalling of the PE is not prohibited by R_{SCLVV}.
- TRCSTALLCTLR.ISTALL == 0b1.
- Any of the following are true:
  - TRCSTALLCTLR.NOOVERFLOW == 0b1.
  - The available space in the internal storage of the trace unit is below the level indicated in TRCSTALLCTLR.LEVEL.

Otherwise, the trace unit does not stall the PE due to the stalling feature or no overflow feature.

R_{NVKXX} The trace unit does not indefinitely stall the operation of the PE.

I_{XPRQJ} In a multi-threaded processor, if the trace unit stalls a PE, Arm recommends that stalling or disruption of the processing of other PEs is minimized. In particular, if tracing of one or more PEs in a multi-threaded processor is enabled but tracing of other PEs in the multi-threaded processor is disabled, Arm recommends that if the PEs being traced are stalled by their respective trace units then the stall has minimal effect on the PEs which are not being traced.

I_{KBXXH} The levels indicated in TRCSTALLCTLR.LEVEL are the levels of intrusion allowed.

I_{ZRQBK} A summary of the stalling and no overflow scenarios is shown in Table D6.5, when TRCIDR3.STALLCTL == `0b1` and TRCIDR3.SYSSTALL == `0b1`.

Table D6.5: Summary of TRCSTALLCTLR stalling and no overflow features

ISTALL	NOOVERFLOW	LEVEL	Description
0	0	X	Stalling is disabled
0	1	X	It is CONSTRAINED UNPREDICTABLE whether the no overflow feature is enabled or stalling is disabled
1	0	zero	Stalling is enabled at the minimum level
1	0	non-zero	Stalling is enabled and is based on the value in TRCSTALLCTLR.LEVEL
1	1	zero	The no overflow feature is enabled, preventing overflows
1	1	non-zero	The no overflow feature is enabled, preventing overflows, and TRCSTALLCTLR.LEVEL might cause stalling earlier than necessary to prevent overflows

D6.10.6 No overflow

I_{DJCLX} A trace unit might include an optional feature to prevent overflows, which is indicated by TRCIDR3.NOOVERFLOW.

R$_{\text{YSPHL}}$ TRCSTALLCTLR.NOOVERFLOW controls the no overflow feature.

S$_{\text{BSPDF}}$ Enabling the no overflow feature might have a significant impact on PE performance.

R$_{\text{LCGZJ}}$ While the no overflow feature is enabled, and while the number or frequency of ETEEvents is below an IMPLE-MENTATION DEFINED threshold, the trace unit does not overflow.

R$_{\text{VHMZX}}$ The threshold is greater than or equal to one of each numbered ETEEvent, for each trace session.

R$_{\text{MYMKW}}$ When TRCIDR3.SYSSTALL is 0b0 the effective value of TRCSTALLCTLR.NOOVERFLOW is 0b0 which means the no overflow feature is disabled.

R$_{\text{JYYLV}}$ When TRCSTALLCTLR.ISTALL == 0b0 and TRCSTALLCTLR.NOOVERFLOW is 0b1, it is CONSTRAINED UNPREDICTABLE whether any stalling is disabled or whether the no overflow feature is enabled.

D6.10.7 Event Trace

I$_{\text{GGMQT}}$ The ETE architecture supports the tracing of additional information in the trace stream. These are known as ETEEvents, also known as Event trace. The trace unit supports up to 4 ETEEvents. The generation of ETEEvents is controlled by selecting resources selectors. The occurrence of ETEEvents can be communicated in the following ways:

- To the system by D7.10 *External Outputs*.
- To the trace analyzer by D5.68 *Event Packet*.

D6.10.8 Context identifier tracing

I$_{\text{RJJZW}}$ The trace unit can be programmed to include information about the current execution context of the program being executed on the PE, including:

- The current process identifier, stored in CONTEXTIDR_EL1. This is known as the *Context identifier*.
- The current virtual machine identifier, stored in CONTEXTIDR_EL2. This is known as the *Virtual context identifier*.

R$_{\text{GVMFG}}$ The trace unit supports tracing of the Context identifier, with TRCIDR2.CIDSIZE indicating a 32-bit Context identifier size.

D6.10.9 Virtual context identifier tracing

I$_{\text{VMGBJ}}$ Whether an implementation supports Virtual context identifier tracing is IMPLEMENTATION DEFINED. If it does, the trace unit can be programmed to output the identifier of a virtual machine that the PE is executing.

I$_{\text{FDDXM}}$ This option is enabled by setting TRCCONFIGR.VMID to 0b1.

R$_{\text{PTVYD}}$ If the PE implements EL2, the trace unit supports a 32-bit Virtual context identifier, with TRCIDR2.VMIDSIZE indicating a 32-bit Virtual context identifier size. The source of the Virtual context identifier is CONTEXTIDR_EL2.PROCID.

R$_{\text{BRDYF}}$ If the PE does not implement EL2, the trace unit does not support a Virtual context identifier, with TRCIDR2.VMIDSIZE indicating Virtual context identifier tracing is not supported.

> **Note**
>
> Previous trace architectures from Arm supported the ability to select the source of the Virtual context identifier. This specification does not support Virtual context identifier selection, and only permits CONTEXTIDR_EL2.PROCID as the source of the Virtual context identifier. See TRCIDR2.VMIDOPT

for more details.

D6.11 Compression

I_{WVBMT} Additional compression of the trace byte stream is achieved by the following methods:

- Removing elements that can be implied by the trace analyzer:
    - Implying the existence of *Commit elements* based on the tracing of other elements.
    - Removing *Target Address elements* that can calculated by the trace analyzer by analysis of previous traced PE execution.

- Combining multiple elements together into a single packet:
    - Combining *Atom elements* into a single packet.
    - Combining *Cancel elements* and *Mispredict elements* into a single packet.

D6.11.1 Implied commits

I_{BTGGX} The ETE trace protocol provides mechanisms to minimize the amount of *Commit elements* which need to be explicitly output in the trace byte stream. When a *P0 element* is output in the trace byte stream, if the number of speculative *P0 elements* output exceeds TRCIDR8.MAXSPEC, then a *Commit element* is implied which resolves the oldest speculative *P0 element*. For more details on the packets which imply *Commit elements*, see Chapter D5 *Protocol Description*.

R_{YKLRM} The trace unit does not generate commit packets for *Commit elements* that have been implied by the trace protocol.

D6.11.2 Atom packing

I_{QMVNP} The ETE trace protocol provides packets which allow groups of consecutive *Atom elements* to be packed into a single trace packet. The diagram below shows the decision tree for generating the different formats of Atom packets.
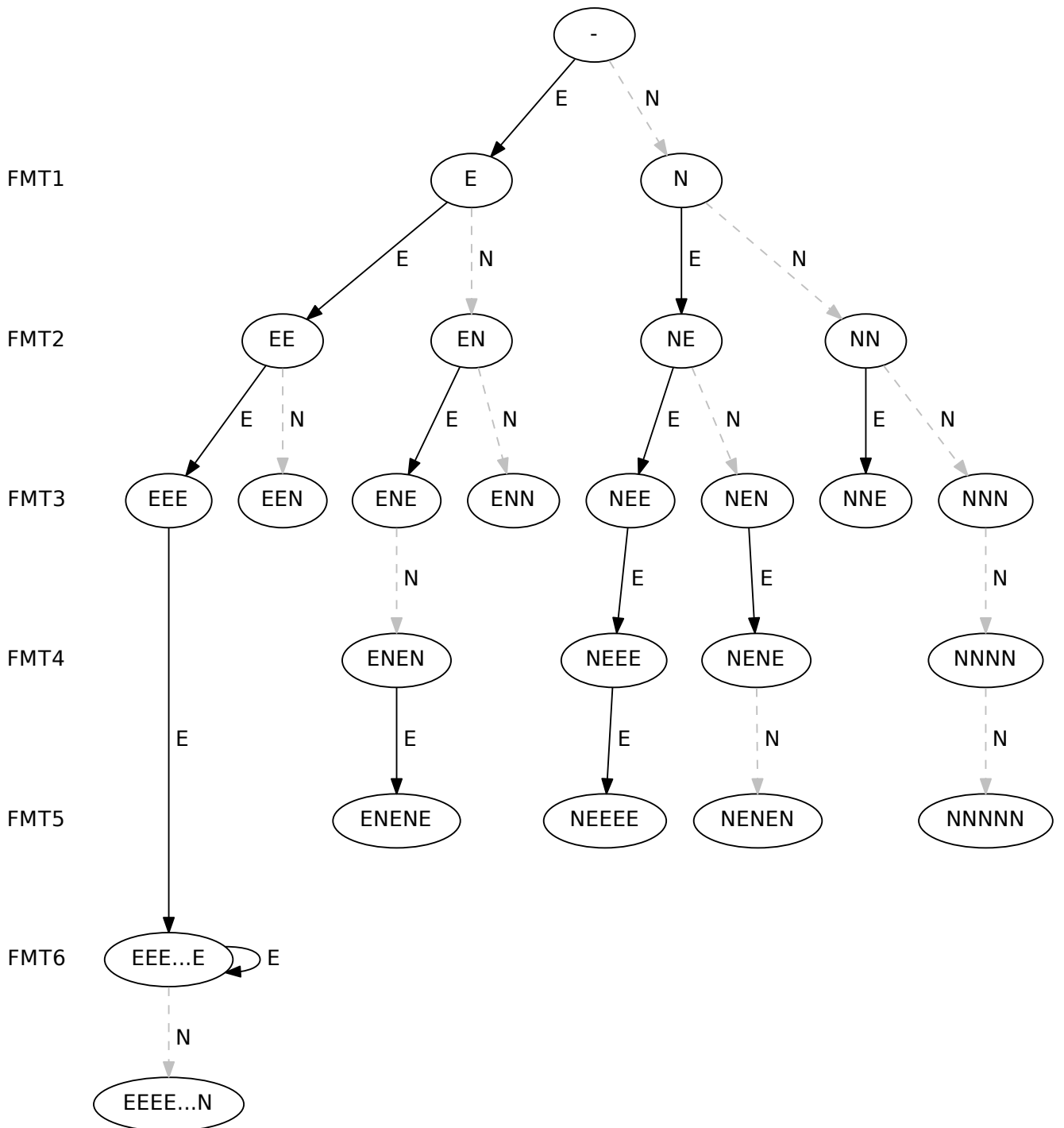
Figure D6.6: Atom packing

I_{XXSGS} Cancel Packets can indicate a number of *Atom elements* as well as the *Cancel element*.

I_{KWWHP} The Mispredict Packets can indicate a number of *Atom elements* as well as the *Mispredict element*.

D6.11.3 Address Compression

I_{LFFCR} The trace unit can remove program addresses from the trace stream. The trace analyzer can infer the addresses from the program image and previous history.

This includes the targets of direct *P0 instructions*, where the target address is encoded in the instruction itself.

R_{SJPYH} The trace unit retains the *Address information* of up to the last three addresses that were explicitly output in the trace protocol, as contained in:

- Target Address packets.
- Source Address packets.
- Exception packets.
- Transaction Failure packets.
- PE Reset packets.
- Q packets.

I_{SXNYK} The explicitly output addresses that the trace unit retains are known as the address history buffer.

I_{LPYRK} For optimized trace protocol efficiency, Arm recommends that the address history buffer is three entries deep.

R_{LMHFW} When any of the following packets are generated, the trace unit pushes the address value and sub_isa to the address history buffer:

- Target Address packet.
- Source Address packet.
- Q packet that implies a *Target Address element*.

R_{CMCRT} When an Exception packet is generated, the trace unit pushes the preferred exception return address value and sub_isa to the address history buffer.

R_{FPDFJ} When one of the following packets is generated with an UNKNOWN address, the trace unit pushes an address value of $0x0$ and sub_isa of IS0 to the address history buffer.

- Transaction Failure packet.
- PE Reset packet.

R_{BPRDC} When a Target Address packet is generated, the trace unit uses the address history buffer to identify when a Target Address Exact Match packet can be used. When a Target Address Exact Match packet cannot be used, the most recent entry in the address history buffer is used for the Target Address packet selection.

R_{GWKFD} When a Source Address packet is generated, the trace unit uses the address history buffer to identify when a Source Address Exact Match packet can be used. When a Source Address Exact Match packet cannot be used, the most recent entry in the address history buffer is used for the Source Address packet selection.

R_{YLXFK} When an Exception packet is generated, the trace unit uses the address history buffer to identify when an Exception Exact Match Address packet can be used. When an Exception Exact Match Address packet cannot be used, the most recent entry in the address history buffer is used for the Exception Address packet selection.

R_{YCMCG} When a Q packet is generated which implies a *Target Address element*, the trace unit uses the address history buffer to identify when a Q with Exact Match Address packet can be used. When a Q with Exact Match Address packet cannot be used, the most recent entry in the address history buffer is used for the Q Address packet selection.

R_{BTWGD} When a Trace Info packet is generated, the trace unit sets all entries of the address history buffer to have an address value of $0x0$ and sub_isa of IS0.

D6.11.4   Return Stack Address Matching

I_{NHWVZ} The trace unit might contain the optional return stack function. The return stack operates when Branch with Link instructions or indirect *P0 instructions* are taken, and provides a mechanism to allow the trace unit to remove certain *Target Address elements* from the trace element stream. The trace analyzer maintains an independent copy of the return stack which is used to determine when *Target Address elements* have been removed and then infer the target of indirect *P0 instructions*.

R_{HNDJJ} The depth of the return stack is IMPLEMENTATION DEFINED from 0 to 15 entries.

I_{BLYHW} For optimized trace protocol efficiency, Arm recommends the trace unit implements the return stack with at least 3 entries.

R_{HFCTC} While TRCCONFIGR.RS is `0b1`, when a Branch with Link instruction is predicted as taken and is traced, the trace unit pushes the following *Address information* to the return stack:

- The instruction address + the instruction size, that is, the return address for the Branch with Link instruction.
- The sub_isa from the instruction set encoding (see D5.3.2 *Instruction set encoding*).

R_{ZKTHK} When a return stack push occurs, all existing entries are shifted down one place on the return stack and the new entry is pushed to the top entry of the return stack.

R_{ZSVDQ} While the return stack is full, when a return stack push occurs, the oldest entry on the return stack is discarded.

R_{FFXPW} When a Branch with Link instruction is predicted as taken and traced with an E *Atom element*, when a return stack push occurs, the trace unit pushes to the return stack, even if the prediction is incorrect and is subsequently corrected to an N *Atom element*.

R_{NYHFH} When a Branch with Link instruction is predicted as not taken and traced with an N *Atom element*, the trace unit does not push to the return stack, even if the prediction is incorrect and is subsequently corrected to an E *Atom element*.

R_{GVLKJ} When a Branch with Link instruction is implied by a *Q element*, the trace unit does not push to the return stack.

R_{WRXCW} When a Branch with Link instruction is executed in a branch broadcasting region, the trace unit does not push to the return stack.

R_{QHSBN} When an indirect *P0 instruction* is taken and traced, and the *Address information* in the resultant *Target Address element* matches the address and sub_isa on the top of the return stack, the trace unit performs a return stack pop.

R_{HTKJS} When a return stack pop occurs, both of the following occur:

- The trace unit discards the *Target Address element* that matches the address and sub_isa on the top of the return stack.
- The trace unit removes the top entry of the return stack, and shifts each older entry up one position.

R_{WBCJG} When an indirect *P0 instruction* is implied by a *Q element*, the trace unit does not perform a return stack pop.

I_{BCWSQ} When an indirect *P0 instruction* is taken, it is possible that the target address is predicted incorrectly by the PE.

R_{YMRGB} When the target address of a taken indirect *P0 instruction* is incorrectly predicted, and the incorrect target address is traced with a *Target Address element*, the trace unit corrects the incorrect address by generating a new *Target Address element* with the correct target address, and neither of the target addresses cause a return stack pop.

R_{GBHNP} When the target address of a taken indirect *P0 instruction* is incorrectly predicted, and the incorrect target address matches the top entry of the return stack, the trace unit subsequently generates a *Target Address element* with the correct target address, and neither of the target addresses cause a return stack pop.

R_{ZCCBS} When the final status of the *Atom element* corresponding to an indirect *P0 instruction* is E, including when one or more Mispredict elements change the status of the *Atom element*, the trace unit performs a return stack pop.

Note

A return stack push only occurs if the initial *Atom element* state for the Branch with Link instruction is E. Conversely, a return stack pop only occurs if the final *Atom element* state for the indirect *P0 instruction* is E.

R_{SLDXR} When an instruction that is both a Branch with Link instruction and an indirect *P0 instruction* is executed, the trace unit performs the following actions on the return stack, in order:

1. Determine whether a return stack push is possible and push if required.
2. Determine whether a return stack pop is possible and pop if required.

> **Note**
>
> Some previous trace architectures from Arm use a different order of operations.

R_{SBZJB} When any of the following occur, the trace unit discards the contents of the return stack:

- The trace unit generates a *Trace Info element*.
- The trace unit generates a *Trace On element*.
- The PE enters a branch broadcasting region.

I_{XRQHQ} A trace unit might discard the contents of the return stack at any time.

I_{DCNGF} When the return stack contents are discarded, there is no requirement for the trace analyzer to be aware that this discard operation has occurred. This is because even though the contents of the trace unit return stack are discarded, there are no adverse consequences if the contents of the trace analyzer return stack are retained, but never used.

R_{GZSSX} After a *Trace Info element*, a *Target Address element* and a *Context element* are required but might not be generated immediately. If the *Target Address element* and the *Context element* are not generated before the next *P0 element*, then any Branch with Link instructions must not push on to the return stack until both the *Target Address element* and the *Context element* have been generated.

> **Note**
>
> This restriction prevents the trace unit from performing return stack pushes for instructions that the trace analyzer cannot analyze, because it is not yet fully synchronized.

D6.11.5 Timestamp Value Compression

I_{GYYNG} The trace analyzer maintains a copy of the last *Timestamp element* value broadcast. The *Timestamp element* value might be compressed relative to the last value and only the bits that have changed need to be encoded.

R_{GPGQQ} When a Trace Info packet is generated, the trace unit sets its maintained value of the last *Timestamp element* to zero, and when the trace unit generates a subsequent Timestamp packet the value is compressed relative to this new zero value. This means that the first Timestamp packet after a Trace Info packet contains all non-zero bits of the Timestamp value.

Chapter D7
Resources

I_{PKCDG}　　The ETE architecture has a number of resources that can be used to be provide advanced filtering functionality.

D7.1 Resource operation

R_{PWBZK} The resources operate in one of the following states:

Running All the resources are active.

Pausing The resources are progressing to the Paused state.

Paused All the resources are static and inactive except for External Input Selectors.
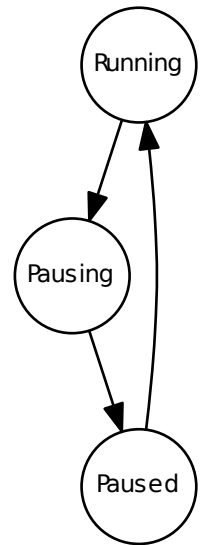
Figure D7.1: Resources operation

I_{HWYYK} As described in D6.2 *System Behaviors*, the trace unit can be disabled by either:

- Setting TRCPRGCTLR.EN to `0b0`.
- Locking the OS Lock, by setting OSLAR_EL1.OSLK to `0b1`.

R_{JLLVN} While the resources are in the Running state, and when any of the following are occur, the resources enter the Pausing state:

- The trace unit becomes disabled.
- The trace unit enters the low-power state.
- The *Processing Element* (PE) begins executing in a Trace Prohibited region.

R_{YWDVJ} While the resources are in the Pausing state, the resources enter the Paused state in finite time.

R_{LYFDT} While the trace unit is in the Paused state, when all of the following are true, the resources enter the Running state:

- The trace unit is enabled.
- The trace unit is not in the low-power state.
- The PE is not executing in a Trace Prohibited region.

R_{TMPZZ} A trace unit buffer overflow has no impact on the behavior of the resources.

D7.1.1 Behavior of the resources while in the Running state

S_{JVYQP} The time taken for the resources to operate might vary between different trace unit implementations.

D7.1.2 Behavior of the resources while in the Pausing state

R_{RDCGC} When the resources enter the Pausing state, the resources perform the following procedure:

1. All resources, except for the Sequencer and any Counters, are driven low as inputs to the Resource Selector logic. The Counters and the Sequencer behave as normal.

2. The states that the inputs were at before they were driven low are propagated through the Resource Selector logic.

3. The states of the Counters and the Sequencer are propagated through the Resource Selector logic one more time. That is, the states of the Counters and the Sequencer are propagated through the Resource Selector logic for the length of time that it takes for the state of a resource to be propagated through the Resource Selector logic.

4. The resources enter the Paused state.

I_{LGWRK} The procedure that the resources perform when the resources are in the Pausing state has the result that, for resource events that are activated by a resource that is not a Counter or a Sequencer, no activity is lost, because all those resource events are updated.

I_{CQNGN} When Counter and Sequencer states are propagated back as resources, so that a loop is created, then the following are true:

- If a Counter at zero resource is being used to activate either the Sequencer or a Counter, then that Counter at zero resource might be propagating through the Resource Selector logic at the time when the procedure ends. In this case, the Sequencer state resource or other Counter at zero resource that is activated by that Counter at zero resource might be lost.

- If a Sequencer state resource is being used to activate a Counter, then that Sequencer state resource might be propagating through the Resource Selector logic at the time when the procedure ends. In this case, the Counter at zero resource that is activated by that Sequencer state resource might be lost.

I_{BRZXY} When the trace unit becomes disabled, the behavior of the resources in the Pausing state ensures that the programmers model provides a consistent view of the state of the trace unit resources. That is, with regard to the Counters and the Sequencer, the following are true:

- If the state of the Sequencer is selected to be propagated back as a resource, then the view of the Sequencer as a resource event and the view of the Sequencer resource state each show the same Sequencer state.

- If the state of a Counter is propagated back as a resource, then the view of the Counter as a resource event and the view of the Counter resource state each show the same Counter state. The Counter state might be either of the following:
  - The Counter is at zero.
  - The Counter is not at zero.

D7.1.3 Behavior of the resources while in the Paused state

I_{YXKSQ} The behavior of the resources when the PE enters the low-power state or a Trace Prohibited region differs from other trace architectures defined by Arm.

R_{FHYQW} While the resources are in the Paused state and the trace unit is not disabled, the resources do not lose resource events that are in transition, except those permitted when moving through the Pausing state of the resources. See D7.1.2 *Behavior of the resources while in the Pausing state* for details on the resource events that are permitted to be lost when in the Pausing state.

I_{HZRSS} While the resources are in the Paused state, the resources might not observe resource events that are in transition until after the resources leave the Paused state.

R_{YWQNQ} While the resources are in the Paused state, the resources remain in the state they are in.

R_{BQSMN} While the resources are in the Paused state, the trace unit drives all External Outputs low.

R_{MVZYP} When the trace unit becomes disabled and the resources enter the Paused state, and not before, the trace unit might set TRCSTATR.PMSTABLE to `0b1`.

R_{RWNTS} While TRCSTATR.PMSTABLE is set to `0b1`, all resources and resource events remain in a quiescent state.

Note

The behavior of the External Input Selectors is detailed in D7.11.1 *Operation while in Paused state*.

D7.1.4 Behavior of resources on a Trace synchronization event

R_{RFSRY} When the following resources have finished calculations for all instructions prior to the previous Context synchronization event, a Trace synchronization event completes:

- Address Comparators.
- Context Identifier Comparators.
- Virtual Context Identifier Comparators.
- Single-shot Comparator Controls.

D7.2 Resource organization

I_{NJLRF} There are 2 types of resources:

- Precise resources.
- Imprecise resources.

I_{JCKLL} Each resources has a current state, which is output as a *Resource state*. The Resource state is selected by Resource Selectors, and then used by various trace unit functions as a *Resource event*, see Figure D7.2.
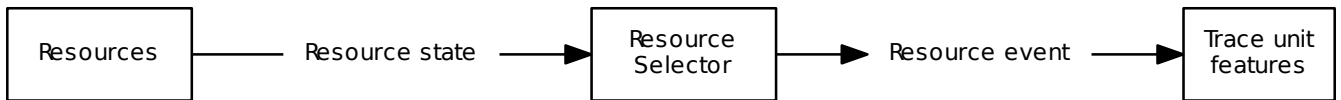
Figure D7.2: Resources organization

D7.2.1 Precise Resources

I_{QSPKY} The precise resources are used in the evaluation of the ViewInst include/exclude function and the ViewInst start/stop function.

R_{WNGDH} The trace unit evaluates the precise resources for each instruction block. See D6.6.4 *Instruction Block* for more details.

R_{NFDCZ} The trace unit maintains execution order of the precise resources.
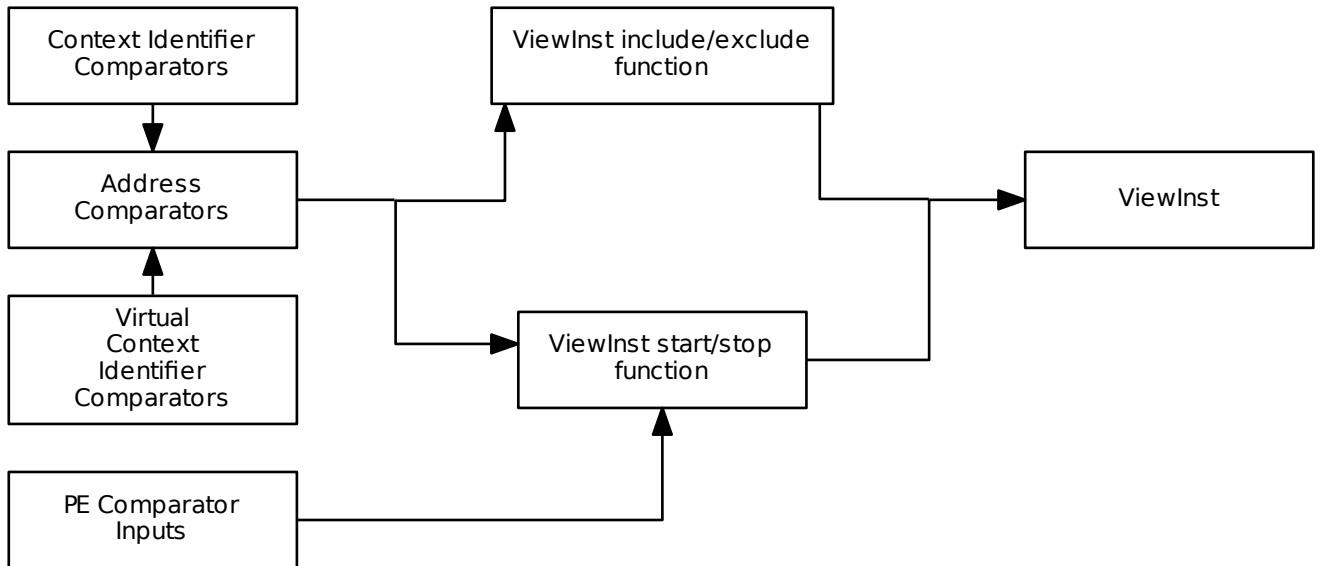
Figure D7.3: Precise Resource Path

D7.2.2 Imprecise Resources
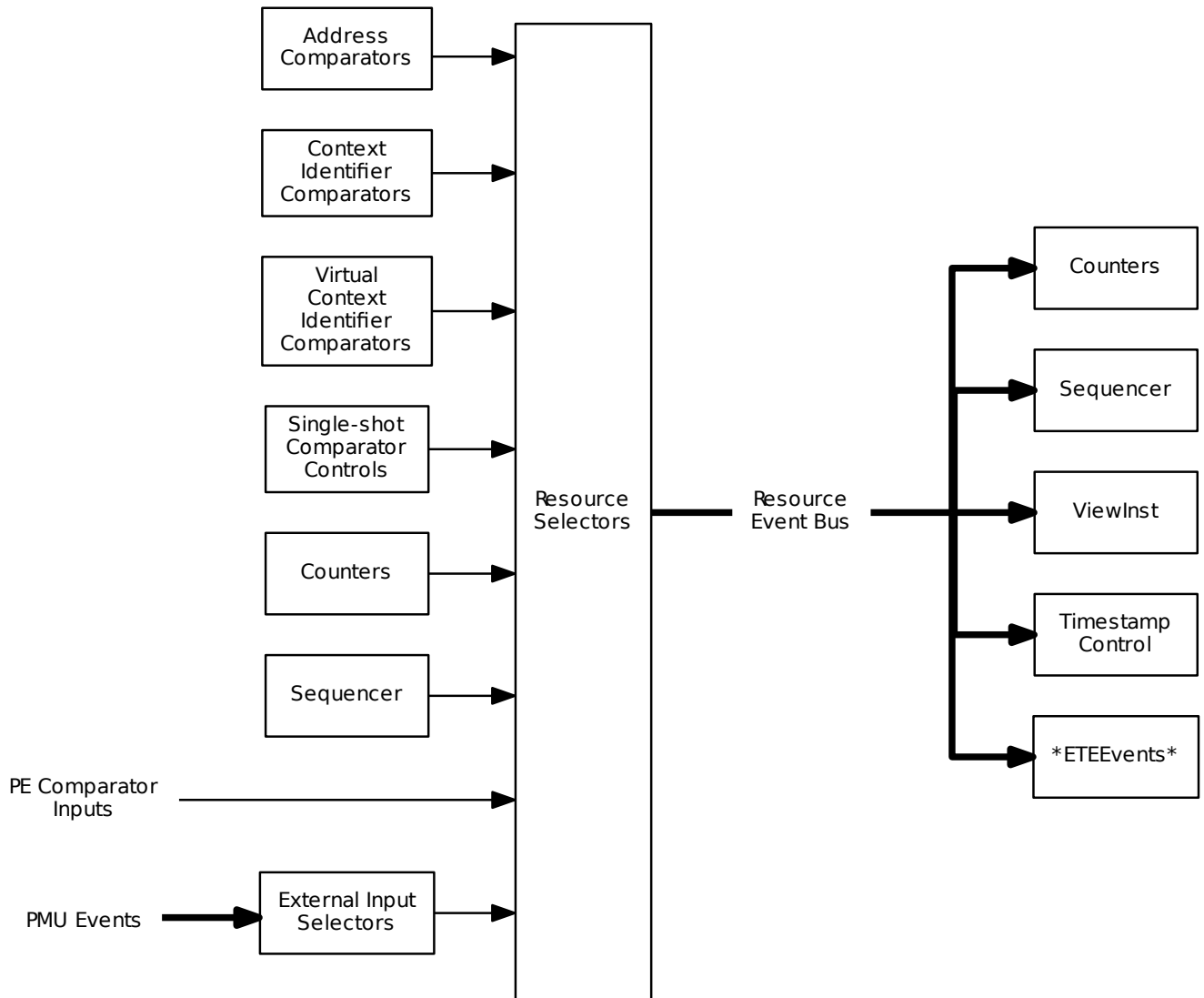
Figure D7.4: Resources organization

D7.3 Selecting a resource or a pair of resources

I_{BRQFW} A resource is selected by using a Resource Selector.

R_{QDJVV} Each Resource Selector uses one of the 30 TRCRSCTLR<n> registers. The trace unit implements Resource Selectors in pairs, so that a maximum of 15 programmable pairs can be implemented.

R_{NRSGN} Resource Selector 0 always provides a FALSE result.

R_{SXSQT} While the resources are in the Running state, Resource Selector 1 provides a TRUE result.

I_{TQVKS} TRCIDR4.NUMRSPAIR indicates how many pairs of Resource Selectors are implemented.

S_{MSHWC} Resource Selectors can be used in pairs or used individually. When a pair of Resource Selectors is used, a Boolean function can be applied to the outputs of the combination of selected resources. See Figure D7.6.

R_{WZVDQ}   While TRCRSCTLR<n>.SELECT[m] is `0b1`, the Resource Selector selects the Resource Number m of the group selected by TRCRSCTLR<n>.GROUP as described in Table D7.1.

Table D7.1: Resource grouping

Group	Resource Number	Resource
`0b0000`	0-3	External Input Selectors 0-3
	4-15	RESERVED
`0b0001`	0-7	PE Comparator Inputs 0-7
	8-15	RESERVED
`0b0010`	0	Counter 0 at zero
	1	Counter 1 at zero
	2	Counter 2 at zero
	3	Counter 3 at zero
	4	Sequencer state 0
	5	Sequencer state 1
	6	Sequencer state 2
	7	Sequencer state 3
	8-15	RESERVED
`0b0011`	0-7	Single-shot Comparator Control 0-7
	8-15	RESERVED
`0b0100`	0-15	Single Address Comparator 0-15
`0b0101`	0-7	Address Range Comparator 0-7
	8-15	RESERVED
`0b0110`	0-7	Context Identifier Comparator 0-7
	8-15	RESERVED
`0b0111`	0-7	Virtual Context Identifier Comparator 0-7
	8-15	RESERVED
`0b1xxx`	0-15	RESERVED

R_{HVNQG}   While TRCRSCTLR<n>.INV is set to `0b0` and one or more resources in a group are selected, when any of the outputs of the selected resources are high, the Resource Selector fires.

R_{WFGMY}   While TRCRSCTLR<n>.INV is set to `0b1`, when none of the outputs of the selected resources are high, the Resource Selector fires.
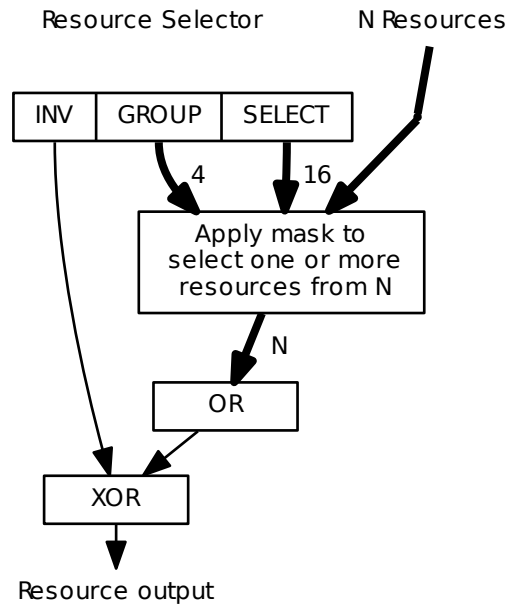
Figure D7.5: A Resource Selector

D7.3.1   A Resource Selector pair

I_{DLRMJ} The Resource Selectors are arranged in pairs, and the result of each of a pair of Resource Selectors can be combined using a boolean function and used to drive other resources and events in the trace unit.

R_{KTNJM} For each TRCRSCTLR<n> register which is the lower register for a pair of Resource Selectors, the TRCRSCTLR<n> register has the TRCRSCTLR<n>.PAIRINV field.

I_{QKTSJ} For example:

- TRCRSCTLR2 and TRCRSCTLR3 constitute a Resource Selector pair. In this case:
    - TRCRSCTLR2 is the lower register.
    - TRCRSCTLR2.PAIRINV optionally inverts the result of the Boolean function that is applied to the outputs of the combination of selected resources.
    - TRCRSCTLR3 is the upper register.
    - TRCRSCTLR3.PAIRINV is RES0.

This means that, when a Resource Selector pair is used, the following scenario is possible:

- One TRCRSCTLR<n> might select only one resource within the group.
- The other TRCRSCTLR<n> might select more than one resource from the group, so that the result is a logical OR of the selected resources.
- A Boolean function, for example a logical AND, might be applied to the outputs of the combination of selected resources.
- The result of that Boolean function might be inverted by using PAIRINV. Figure D7.6 shows this.

I_{LPJXK} In Figure D7.6, the Boolean function is selected by using the INV field for each Resource Selector, with the PAIRINV field for each Resource Selector pair, see Table D7.2.
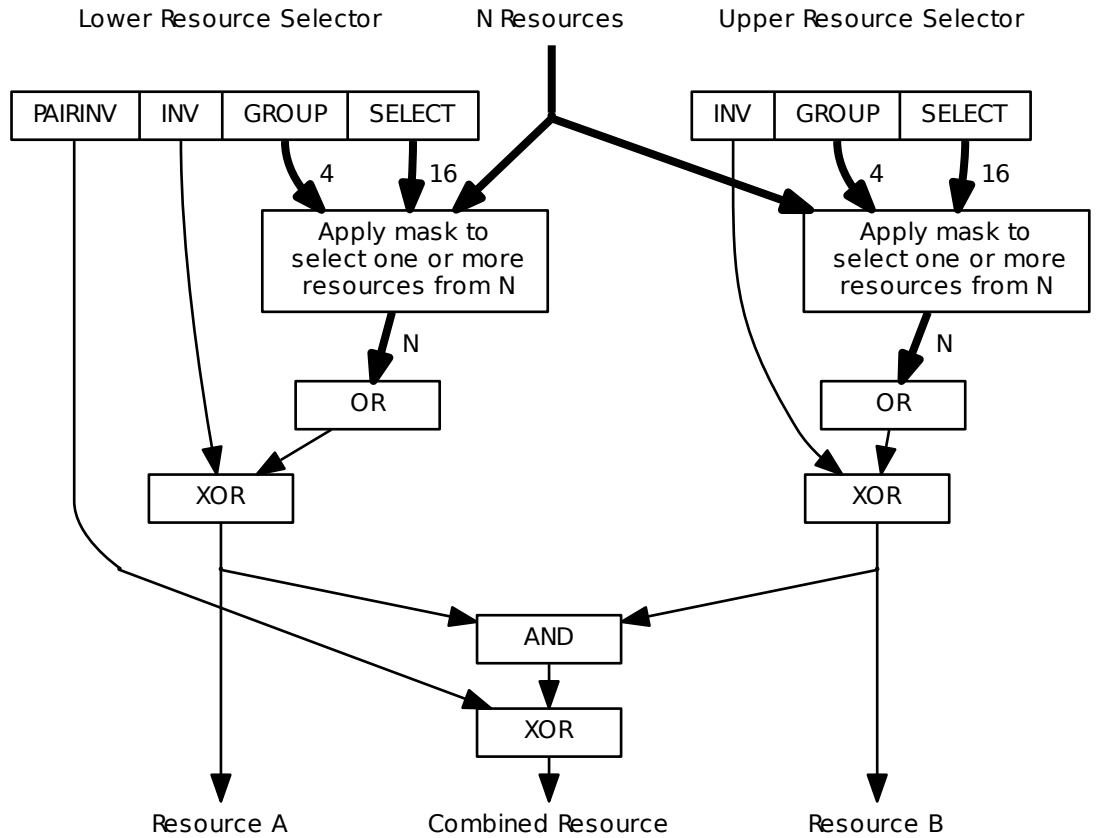
Figure D7.6: A Resource Selector pair

Table D7.2: Selecting a boolean function

Function	Resource A INV	Resource B INV	PAIRINV
$A \wedge B$	0b0	0b0	0b0
$\neg A \vee \neg B$	0b0	0b0	0b1
RESERVED	0b0	0b1	0b0
$\neg A \vee B$	0b0	0b1	0b1
$\neg A \wedge B$	0b1	0b0	0b0
RESERVED	0b1	0b0	0b1
$\neg A \wedge \neg B$	0b1	0b1	0b0
$A \vee B$	0b1	0b1	0b1

D7.4 Address comparators

I_{LGGVG} An ETE trace unit provides between 0 and 16 *Single Address Comparators* (SACs) that each compare the instruction address with a user-programmed value.

R_{YCRNP} The trace unit implements SACs in pairs, so that a trace unit implementation contains an even number of SACs.

I_{MNTCY} TRCIDR4.NUMACPAIRS indicates how many pairs of SACs are implemented.

R_{YMVDZ} When the PE executes instructions in Debug state, Address Comparators do not match.

R_{YPWLR} When the PE executes instructions in a Trace Prohibited region, Address Comparators do not match.

I_{RFTWJ} Address Comparators might match in failed transactions.

I_{WDJPG} Address Comparators might match on speculative execution.

D7.4.1 Single Address Comparators

I_{SSHHT} A *Single Address Comparator* (SAC) can be used in the following ways:

- As inputs to the ViewInst start/stop function in the ViewInst function (see D6.8.2 *ViewInst start/stop function filtering*).
- As an individual resource.
- The comparator can be programmed so that, whenever the PE is in Non-secure state, the comparator only matches in certain Exception levels.
- The comparator can be programmed so that, whenever the PE is in Secure state, the comparator only matches in certain Exception levels.

R_{DKCFF} An SAC only matches on Exception levels and Security states as indicated by TRCACATR<n>.

$$\text{SAC\_EL}_i[n] = \begin{cases} \neg\text{TRCACATRn.EXLEVEL\_S\_EL0} & \text{Secure EL0} \\ \neg\text{TRCACATRn.EXLEVEL\_S\_EL1} & \text{Secure EL1} \\ \neg\text{TRCACATRn.EXLEVEL\_S\_EL2} & \text{Secure EL2} \\ \neg\text{TRCACATRn.EXLEVEL\_S\_EL3} & \text{EL3} \\ \neg\text{TRCACATRn.EXLEVEL\_NS\_EL0} & \text{Non-Secure EL0} \\ \neg\text{TRCACATRn.EXLEVEL\_NS\_EL1} & \text{Non-Secure EL1} \\ \neg\text{TRCACATRn.EXLEVEL\_NS\_EL2} & \text{Non-Secure EL2} \end{cases} \tag{D7.1}$$

R_{QFNSK} An SAC only matches on the context indicated by TRCACATR<n>.CONTEXT and TRCACATR<n>.CONTEXTTYPE.

$$m = \text{TRCACATRn.CONTEXT} \tag{D7.2}$$

$$\text{type} = \text{TRCACATRn.CONTEXTTYPE} \tag{D7.3}$$

$$\text{SAC\_CONTEXT}_i[n] = \begin{cases} 1 & \text{type is 0} \\ \text{CIDCOMP}[m] & \text{type is 1} \\ \text{VMIDCOMP}[m] & \text{type is 2} \\ \text{CIDCOMP}[m]\wedge \\ \text{VMIDCOMP}[m] & \text{type is 3} \end{cases} \tag{D7.4}$$

R_{PYMZV} When an instruction is executed, and the address of the lowest byte of the instruction exactly matches the programmed address of an SAC, the SAC matches.

$$\text{SAC\_ADDR}_i[n] = (\text{ThisInstrAddr}()_i \equiv \text{TRCACVRn.ADDRESS}) \tag{D7.5}$$

I_{SPFXX} For example, for a 4-byte instruction at address `0x1000`:

- The lowest byte of the instruction is at `0x1000`.
- The second byte of the instruction is at `0x1001`.
- The third byte of the instruction is at `0x1002`.
- The highest byte of the instruction is at `0x1003`.

If an SAC is programmed with `0x1000`, then it always matches on that instruction at address `0x1000`.

I_{JZXFJ} It is IMPLEMENTATION DEFINED whether an SAC matches when its programmed address matches any byte of an instruction which is not the lowest byte of the instruction.

I_{VSFSS} The Arm architecture supports disabling `IT` instructions on more than one subsequent instruction, using the ITD bits in the SCTLR, HSCTLR, and SCTLR_EL1 System registers. If any of the ITD bits are set to `0b1` and are affecting `IT` operation, and a SAC is programmed to match on the address of the instruction that is immediately after an `IT` instruction, when the instruction immediately after the `IT` instruction is executed it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{TFYFT} If any of the ITD bits are set to `0b1`, Arm recommends that a SAC is programmed to match on the address of the `IT` instruction, instead of the instruction immediately after the `IT` instruction.

S_{MLDYK} To avoid unexpected behavior from an SAC, Arm recommends that the SAC is always programmed with an address that is for the lowest byte of an instruction.

I_{MCKFH} When the instruction immediately after a `MOVPRFX` instruction is executed, if a SAC is programmed to match on the address of this instruction, then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{FPTHL} Arm recommends that a SAC is programmed to match on the address of the `MOVPRFX` instruction, instead of the instruction immediately after the `MOVPRFX` instruction.

I_{TBNTJ} The operation of a SAC is as follows:

$$\text{SAC}_i[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ \text{SAC\_ADDR}_i[n] \wedge \text{SAC\_EL}_i[n] \wedge \text{SAC\_CONTEXT}_i[n] & \text{Otherwise} \end{cases} \tag{D7.6}$$

D7.4.2   Address Range Comparators

I_{HDFQM} Pairs of SACs are arranged to form one *Address Range Comparator* (ARC). An ARC is programmed with an address range, so that whenever any address in that range is accessed, the ARC matches. A trace unit contains between zero and eight *Address Range Comparators* (ARCs). ARCs can be used in the following ways:

- Selected for the ViewInst include/exclude function in the ViewInst function (see D6.8.3 *ViewInst include/exclude function filtering*).
- As individual resources.

An ARC is programmed by programming the SACs as follows:

- The first SAC is programmed with the start address of the instruction range.
- The second SAC is programmed with the end address of the instruction range.

S_{WFSPV} The address that the second SAC is programmed with must be greater than or equal to the address that the first SAC is programmed with, that is, the end address must be greater than or equal to the start address.

R_{MXCGD} While the start address of an ARC is greater than the end address, the behavior of the ARC is CONSTRAINED UNPREDICTABLE, that is, at any time the ARC might do either of the following:

- Match.
- Not match.

R_{XYJLC} While the TRCACATR<n> registers for the SACs in an ARC are programmed to different values, the behavior of the ARC is CONSTRAINED UNPREDICTABLE.

R_{LLQPL} While an ARC is programmed with an instruction address range, when the PE executes an instruction at an address in the following range, the ARC matches:

$$\text{start\_address} = \text{TRCACVRn.ADDRESS} \tag{D7.7}$$

$$\text{end\_address} = \text{TRCACVR(n+1).ADDRESS} \tag{D7.8}$$

$$\text{ARC\_ADDR}_i[n/2] = (\text{ThisInstrAddr}()_i \geqslant \text{start\_address}) \wedge (\text{ThisInstrAddr}()_i \leqslant \text{end\_address}) \tag{D7.9}$$

R_{YYXSQ} When an instruction is executed, and the address of the lowest byte of the instruction is within the programmed address range of an ARC, the ARC matches.

I_{RPFWZ} When an instruction is executed and the programmed address range of an ARC contains addresses for one or more bytes of the instruction, but does not contain the address for the lowest byte of the instruction, it is IMPLEMENTATION SPECIFIC whether the ARC matches.

I_{RZFPT} For example, for a 4-byte instruction at address `0x1000`:

- The lowest byte of the instruction is at `0x1000`.
- The second byte of the instruction is at `0x1001`.
- The third byte of the instruction is at `0x1002`.
- The highest byte of the instruction is at `0x1003`.

If the programmed address range contains `0x1000`, then the ARC always matches. However, if the programmed address range starts at either `0x1001`, `0x1002`, or `0x1003`, then it is IMPLEMENTATION SPECIFIC whether the ARC matches.

S_{HSFTQ} To avoid unexpected behavior from an ARC, Arm recommends that the ARC is always programmed with an address range that starts with an address for the lowest byte of an instruction.

I_{VRRHS} The Arm architecture supports disabling IT instructions on more than one subsequent instruction, using the ITD bits in the SCTLR, HSCTLR, and SCTLR_EL1 System registers. If any of the ITD bits are set to `0b1` and are affecting IT operation, and an ARC is programmed to include the address of the instruction that is immediately after an IT instruction but not include the IT instruction, when the instruction immediately after the IT instruction is executed then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{DMHQH} If any of the ITD bits are set to `0b1`, Arm recommends that an ARC is programmed to include both the IT instruction and the instruction immediately after the IT instruction.

I_{PBKPJ} When the instruction immediately after a MOVPRFX instruction is executed, if an ARC is programmed to include the address of the instruction that is after the MOVPRFX instruction but not the MOVPRFX instruction, then it is CONSTRAINED UNPREDICTABLE whether that comparator matches.

S_{HVTHL} Arm recommends that an ARC is programmed to include both the MOVPRFX instruction and the instruction immediately after the MOVPRFX instruction.

I_{HTXLT} It might be possible for multiple matches to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place with each address in the programmed range. In this case, either or both of the following might occur:

- An address in the range is matched more than once.
- More than one address in the range is matched simultaneously.

R_{HMYMX} When multiple ARC matches occur simultaneously for one ARC, both of the following are true:

- The ARC signals a match at least once.
- The ARC does not signal more matches than the number of instructions that are executed with an address that matches an address in the programmed range.

I_{CTBDN} Each ARC can be used with one, or a combination of, the following:

- A Context Identifier Comparator.
- A Virtual Context Identifier Comparator.

R_{TMCJX} An ARC only matches on Exception levels and Security states as indicated by TRCACATR<2n>.

$$ARC\_EL_i[n] = \begin{cases} \neg TRCACATR<2n>.EXLEVEL\_S\_EL0 & \text{Secure EL0} \\ \neg TRCACATR<2n>.EXLEVEL\_S\_EL1 & \text{Secure EL1} \\ \neg TRCACATR<2n>.EXLEVEL\_S\_EL2 & \text{Secure EL2} \\ \neg TRCACATR<2n>.EXLEVEL\_S\_EL3 & \text{EL3} \\ \neg TRCACATR<2n>.EXLEVEL\_NS\_EL0 & \text{Non-Secure EL0} \\ \neg TRCACATR<2n>.EXLEVEL\_NS\_EL1 & \text{Non-Secure EL1} \\ \neg TRCACATR<2n>.EXLEVEL\_NS\_EL2 & \text{Non-Secure EL2} \end{cases} \quad \text{(D7.10)}$$

R_{VSBJF} An ARC only matches on the context indicated by TRCACATR<2n>.CONTEXT and TRCACATR<2n>.CONTEXTTYPE.

$$m = TRCACATR<2n>.CONTEXT \quad \text{(D7.11)}$$
$$\text{type} = TRCACATR<2n>.CONTEXTTYPE \quad \text{(D7.12)}$$

$$ARC\_CONTEXT_i[n] = \begin{cases} 1 & \text{type is 0} \\ CIDCOMP[m] & \text{type is 1} \\ VMIDCOMP[m] & \text{type is 2} \\ CIDCOMP[m] \wedge \\ VMIDCOMP[m] & \text{type is 3} \end{cases} \quad \text{(D7.13)}$$

R_{RTXJN} The operation of an ARC is as follows:

$$ARC_i[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ ARC\_ADDR_i[n] \wedge ARC\_EL_i[n] \wedge ARC\_CONTEXT_i[n] & \text{Otherwise} \end{cases} \quad \text{(D7.14)}$$

D7.5 Context Identifier Comparator

I_{KDSNY} An ETE trace unit provides between zero and eight Context Identifier Comparators. Each Context Identifier Comparator can be used in any of the following ways:

- Associated with a SAC.
- Associated with an ARC.
- As an individual resource.

R_{DCCBY} While a Context Identifier Comparator is associated with either an SAC or an ARC, only while the PE is executing with the Context identifier that the Context Identifier Comparator is programmed with and when an address is accessed which the SAC or ARC is programmed to match on, the SAC or ARC signals a match.

R_{BKQKQ} While a Context Identifier Comparator is used as an individual resource, when an instruction block is executed with the Context identifier that the Context Identifier Comparator is programmed with, the Context Identifier Comparator matches.

I_{PBXRH} When using a Context Identifier Comparator as an independent resource to activate a resource event, the time that the resource event is activated relative to the time that the Context Identifier Comparator becomes active might be imprecise.

I_{RBLYL} It might be possible for multiple matches of a Context Identifier Comparator to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place.

R_{MPJBW} When multiple Context Identifier Comparator matches occur simultaneously for one Context Identifier Comparator, both of the following are true:

- The Context Identifier Comparator signals a match at least once.
- The Context Identifier Comparator does not signal more matches than the number of instructions that are executed with the Context identifier that the Context Identifier Comparator is programmed with.

I_{HDCJK} A Context Identifier Comparator might match on speculative execution, that is, a Context Identifier Comparator might match if the PE speculatively changes the Context identifier.

R_{MCYYC} When the PE executes instructions in Debug state, Context Identifier Comparators do not match.

R_{SRZGJ} When the PE executes instructions in a Trace Prohibited region, Context Identifier Comparators do not match.

I_{GKDRL} The Context identifier might change at points that are not Context synchronization events, for example when a system instruction is used to write to the current Context identifier register. In these scenarios, the Context Identifier Comparator might compare against the old or new Context identifier value for any instruction after the *P0 element* before the system instruction, up to the Context synchronization event after the system instruction.

$$m = \begin{cases} \text{TRCCIDCCTLR0.COMP0} & n \equiv 0 \\ \text{TRCCIDCCTLR0.COMP1} & n \equiv 1 \\ \text{TRCCIDCCTLR0.COMP2} & n \equiv 2 \\ \text{TRCCIDCCTLR0.COMP3} & n \equiv 3 \\ \text{TRCCIDCCTLR1.COMP4} & n \equiv 4 \\ \text{TRCCIDCCTLR1.COMP5} & n \equiv 5 \\ \text{TRCCIDCCTLR1.COMP6} & n \equiv 6 \\ \text{TRCCIDCCTLR1.COMP7} & n \equiv 7 \end{cases} \tag{D7.15}$$

$$v = \text{TRCCIDCVRn.VALUE} \tag{D7.16}$$

$$\text{cid} = \text{CONTEXTIDR\_EL1.PROCID} \tag{D7.17}$$

$$\text{CIDCOMP}[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ \prod_{j=0}^{7} \Big(v[8j+7:8j] \equiv \text{cid}[8j+7:8j]) \vee \text{m}[j] \Big) & \text{Otherwise} \end{cases} \tag{D7.18}$$

D7.6 Virtual Context Identifier Comparators

I_{BXVPG} An ETE trace unit provides between zero and eight Virtual Context Identifier Comparators. Each Virtual Context Identifier Comparator can be used in any of the following ways:

  - Associated with a SAC.
  - Associated with an ARC.
  - As an individual resource.

R_{RTRBM} While a Virtual Context Identifier Comparator is associated with either an SAC or an ARC, only while the PE is executing with the Virtual context identifier that the Virtual Context Identifier Comparator is programmed with and when an address is accessed which the SAC or ARC is programmed to match on, the SAC or ARC signals a match.

R_{VWYMY} While a Virtual Context Identifier Comparator is used as an individual resource, when an instruction block is executed with the Virtual context identifier that matches the Virtual Context Identifier Comparator value, the Virtual Context Identifier Comparator matches.

R_{FLXQL} While TRFCR_EL2.CX indicates that Virtual Context Identifier Comparators cannot match, Virtual Context Identifier Comparators do not match.

R_{LPKBR} When the PE executes instructions in Debug state, Virtual Context Identifier Comparators do not match.

R_{WZWLT} When the PE executes instructions in a Trace Prohibited region, Virtual Context Identifier Comparators do not match.

I_{SCPJP} When using a Virtual Context Identifier Comparator as an independent resource to activate a resource event, the time at which the resource event is activated relative to the time at which the Virtual Context Identifier Comparator becomes active might be imprecise.

R_{LJRPW} A Virtual Context Identifier Comparator is associated with an SAC by programming TRCACATR<n>.CONTEXT for the SAC.

I_{GJCRG} It might be possible for multiple matches of a Virtual context identifier to occur simultaneously. The definition of when matches occur simultaneously is IMPLEMENTATION SPECIFIC, and might vary because of runtime conditions. However, an example of when multiple matches might occur simultaneously is when multiple instructions are observed in the same processor clock cycle, so that multiple comparisons take place.

R_{JNNDL} When multiple Virtual Context Identifier Comparator matches occur simultaneously for one Virtual Context Identifier Comparator, both of the following are true:

  - The Virtual Context Identifier Comparator signals a match at least once.
  - The Virtual Context Identifier Comparator does not signal more matches than the number of instructions that are executed with the Virtual context identifier that the Virtual Context Identifier Comparator is programmed with.

I_{NPPCF} A Virtual Context Identifier Comparator might signal a match on speculative execution, that is, a Virtual Context Identifier Comparator might signal a match when the PE speculatively changes the Virtual context identifier.

I_{PPWXT} The Virtual context identifier might change at points which are not Context synchronization events, for example when a system instruction is used to write to CONTEXTIDR_EL2. In these scenarios, the Virtual Context Identifier Comparator might compare against the old or new Virtual context identifier value for any instruction after the *P0 element* before the system instruction, up to the Context synchronization event after the system instruction.

$$m = \begin{cases} \text{TRCVMIDCCTLR0.COMP0} & n \equiv 0 \\ \text{TRCVMIDCCTLR0.COMP1} & n \equiv 1 \\ \text{TRCVMIDCCTLR0.COMP2} & n \equiv 2 \\ \text{TRCVMIDCCTLR0.COMP3} & n \equiv 3 \\ \text{TRCVMIDCCTLR1.COMP4} & n \equiv 4 \\ \text{TRCVMIDCCTLR1.COMP5} & n \equiv 5 \\ \text{TRCVMIDCCTLR1.COMP6} & n \equiv 6 \\ \text{TRCVMIDCCTLR1.COMP7} & n \equiv 7 \end{cases} \tag{D7.19}$$

$$\text{VMID} = \text{CONTEXTIDR\_EL2.PROCID} \tag{D7.20}$$

$$v = \text{TRCVMIDCVRn.VALUE} \tag{D7.21}$$

$$\text{VMIDCOMP}[n] = \begin{cases} 0 & \text{When Prohibited} \\ 0 & \text{When in Debug State} \\ \prod_{j=0}^{7} \Big(v[8j+7:8j] \equiv \text{VMID}[8j+7:8j]) \vee \text{m}[j] \Big) & \text{Otherwise} \end{cases} \tag{D7.22}$$

D7.7 Counters

I_{NCCBM} The Counters that are employed by the ETE architecture are all decrement counters.

The ETE architecture enables a trace unit to connect Counter outputs to resource events, so that a Counter at zero state can be used as a resource to activate a resource event. For example, a Counter at zero state might be used to assert an External Output or to make ViewInst active.

An ETE trace unit provides up to four 16-bit Counters. TRCIDR5.NUMCNTR indicates how many Counters are implemented. For each Counter, the following can be specified:

- The initial counter value. This can be programmed using TRCCNTVR<n>.
- The reload value. This can be programmed using TRCCNTRLDVR<n>.
- The resource event that causes the Counter to reload with the reload value. This resource event is called RLDEVENT. In addition, if required, the Counter can be programmed so that it automatically reloads whenever it reaches zero.
- The resource event that enables the Counter to decrement. This resource event is called CNTEVENT. The Counter decrements whenever CNTEVENT is active.

R_{RBMQM} The processor clock clocks the Counters in the trace unit.

R_{PZQGV} While the PE is stalled, the Counters continue to count.

R_{FHFMP} While the resources are in the Paused state, the Counters do not count.

R_{LFVYH} When a Counter value is changed by anything other than a direct write to TRCCNTVR<n>, the trace unit considers the change to be an indirect write to TRCCNTVR<n>.VALUE.

I_{MLDXC} Each Counter operates in one of the two following possible modes:

- Normal mode.
- Self-reload mode.

R_{SBQPN} While the Counter is in Normal Mode, when the Counter reaches zero, the Counter remains at zero until the reload resource event, RLDEVENT, occurs.

R_{HYLGG} While the Counter is in Normal Mode, the Counter-at-zero resource is active for the whole of the time that the Counter is at zero.

R_{YLYPH} While the Counter is in Self-reload Mode, when the Counter reaches zero, when the decrement resource event is next active, the trace unit reloads the Counter with the reload value.

R_{VGJNL} While the Counter is in Self-reload Mode, when the Counter value is zero, the decrement resource event is active and the reload resource event is not active, the Counter-at-zero resource is active for one cycle.

I_{KTRXV} The following examples show various operating scenarios for a single Counter. Each Counter is programmed with a reload value of `0x3`.

Figure D7.7: Counter Example 1, Normal mode
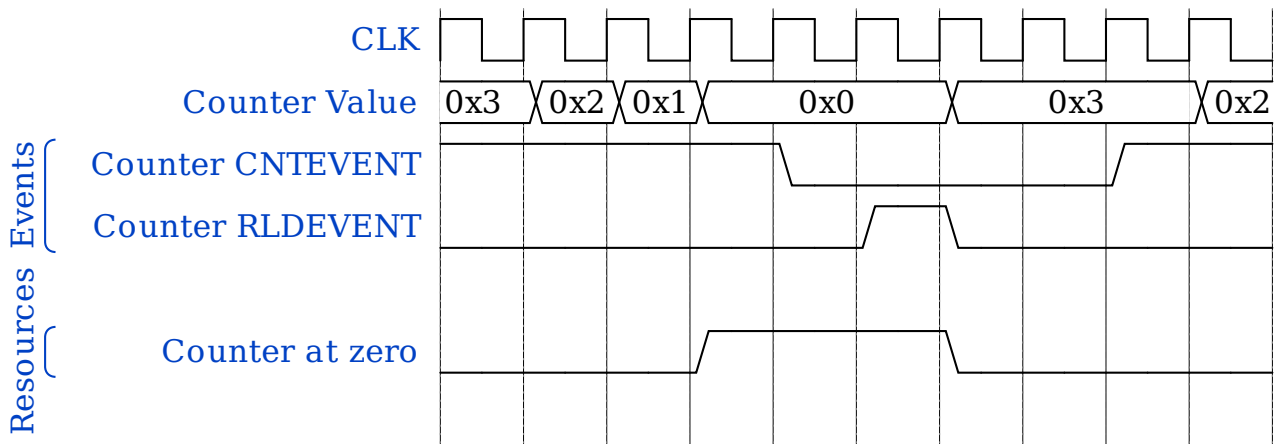
Figure D7.8: Counter Example 2, Normal mode
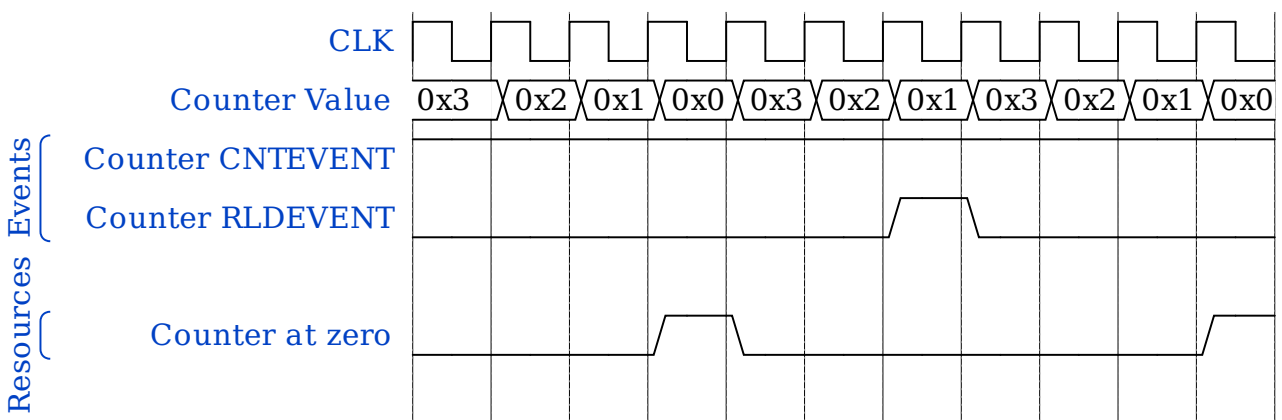
Figure D7.9: Counter Example 3, Self-reload mode
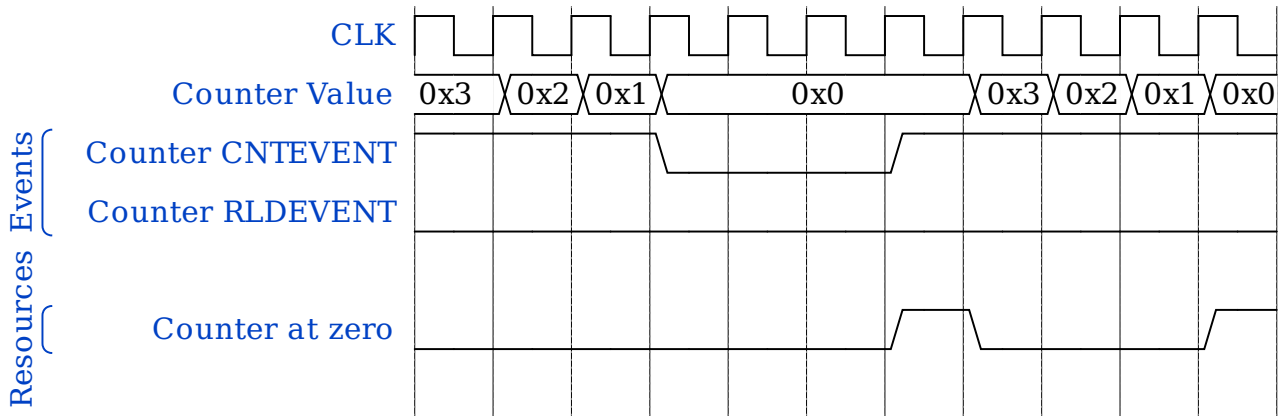
Figure D7.10: Counter Example 4, Self-reload mode
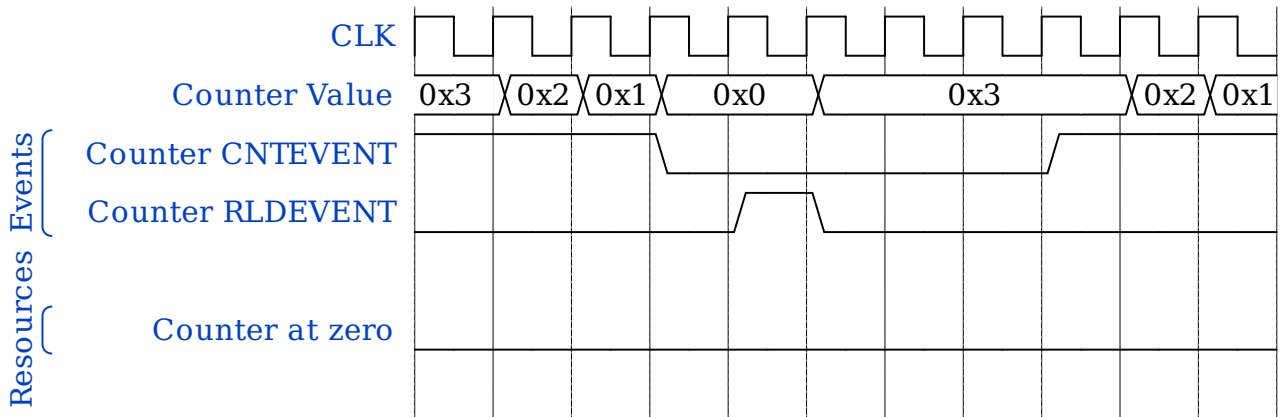
Figure D7.11: Counter Example 5, Self-reload mode

R_{KXLKC} While the decrement resource event is inactive, the Counters do not decrement.

R_{DDCDK} The trace unit prioritizes the reload resource event over the count decrement resource event.

D7.7.1 Forming a larger Counter from two separate Counters

I_{TYLSH} Some Counters can be chained together to form a larger counter, so that every time one Counter reloads, another Counter decrements.

I_{MMDRW} The following example shows an operating scenarios for 2 Counters chained together. Counter 0 is programmed with a reload value of 0x2.
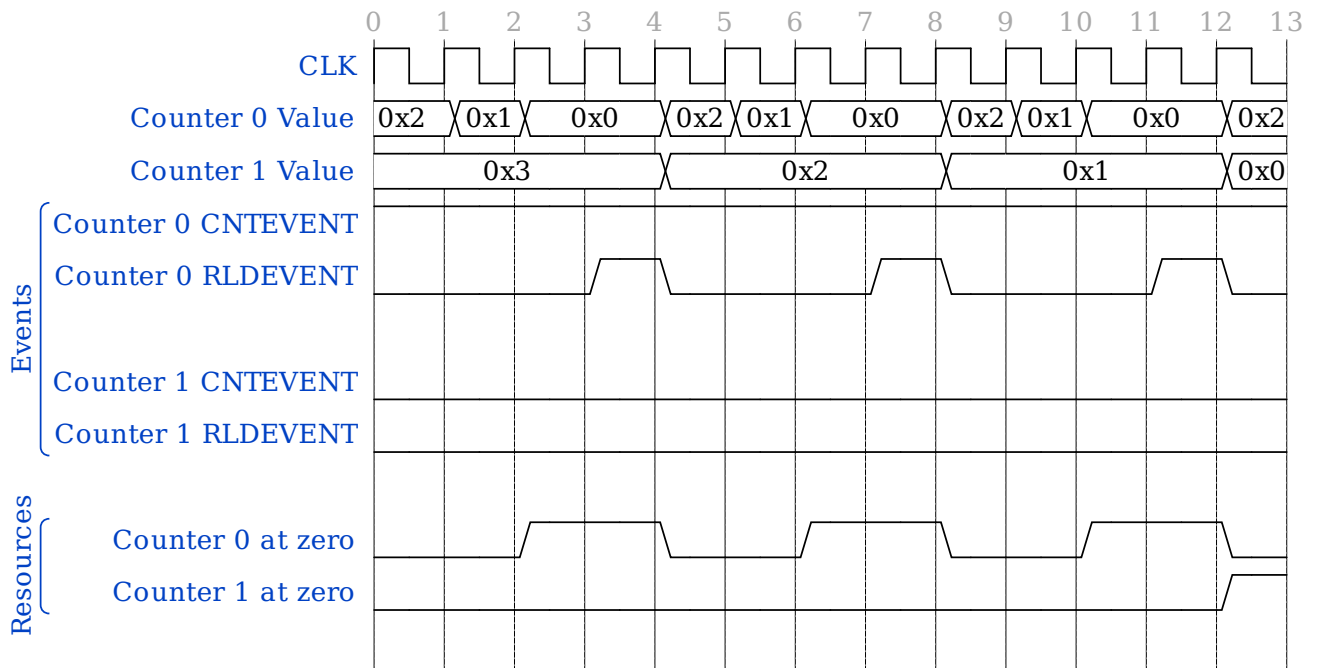
Figure D7.12: Chained Counter Example 1

Only certain Counters can be programmed to do this, as follows:

R_{WPWQD} Counter 1 can be programmed to decrement when Counter 0 reloads.

R_{GYKGZ} Counter 3 can be programmed to decrement when Counter 2 reloads.

R_{QHZFW} The decrement resource event for the higher Counter n is active when either of the following occurs:

- The lower Counter reloads due to one of the following:
  - The reload resource event that is selected by TRCCNTCTLR<n-1>.RLDEVENT.
  - The self-reload mechanism that is controlled by TRCCNTCTLR<n-1>.RLDSELF.
- The decrement resource event that is selected by TRCCNTCTLR<n>.CNTEVENT is active.

R_{BDPDN} While two Counters are chained together to form a larger counter, the larger counter appears as a 32-bit counter without any tearing of the values between the two Counters.

I_{FTDHL} For example, if Counter 0 is in Self-reload mode and has a value of 0x0000 and a reload value of 0xFFFF, and Counter 1 is in Normal mode and has a value of 0x1234, then when a decrement resource event occurs on Counter 0, Counter 0 reloads to 0xFFFF. The reload of Counter 0 causes Counter 1 to decrement, resulting in a value of 0x1233. Therefore the sequence on the Counters on consecutive cycles is 0x12340000 and 0x1233FFFF.

I_{BCMGM} For Counters 1 and 3, TRCCNTCTLR<n>.CNTCHAIN is a RW field that determines whether the Counter is chained. For Counters 0 and 2, TRCCNTCTLR<n>.CNTCHAIN is RES0.

Note

The CounterAtZero resource might not be asserted at the same time that the Counter is at zero. For example, this could happen if the trace unit implementation pipelines some logic.

RLDEVENT	*dec_action*	Counter value	Action	Resource Active	Notes
Inactive	X	0	Stable	Yes	Resource is active while Counter is at zero and remains at zero
Inactive	0	> 0	Stable	No	No activity
Inactive	1	> 0	Decrement	No	Decrement when not zero
Active	X	0	Reload	Yes	Reload, but resource is active because Counter is at zero
Active	X	> 0	Reload	No	Reload

D7.7.2 Counter Operation in Self-reload mode

RLDEVENT	*dec_action*	Counter value	Action	Resource Active	Notes
Inactive	0	X	Stable	No	No activity, resource is not active even if the Counter is at zero
Inactive	1	0	Reload	Yes	Reload because dec_action is active and the Counter is at zero, resource is active only in this cycle
Inactive	1	> 0	Decrement	No	Decrement when not zero
Active	X	X	Reload	No	Reload regardless of decrement action and the value of the Counter, resource is never active

```
// The counter-at-zero resources
array boolean CounterAtZero[0..3];


//
// EvalAllCounters() is called each clock cycle
//
EvalAllCounters()
    array boolean reload[0..3];
    reload[0] = EvalCounter(0, FALSE);
    reload[1] = EvalCounter(1, reload[0]);
    reload[2] = EvalCounter(2, FALSE);
    reload[3] = EvalCounter(3, reload[2]);


//
// EvalCounter() is called for each counter
//
```

```
boolean EvalCounter(integer index, boolean lower_reloads)
    boolean dec_action;
    boolean resource_active;
    bits(16) next_value;
    boolean reload;
    boolean decrement;


    // A dec_action signal is constructed which indicates whether the counter
    // decrements. This is based on TRCCNTCTLR[n].CNTEVENT and, for counters
    // which support chaining, on TRCCNTCTLR[n].CNTCHAIN and on whether or not
    // the lower counter is reloading.
    dec_action = IsEventActive(TRCCNTCTLR[index].CNTEVENT) ||
                            (TRCCNTCTLR[index].CNTCHAIN && lower_reloads);


    // The counter-at-zero resource is active if the counter is
    // currently at zero and is either in Normal mode or in
    // Self-Reload mode and dec_action is active and the reload
    // event is not active.
    resource_active = (TRCCNTVR[index] == 0) &&
                    (!TRCCNTCTLR[index].RLDSELF ||
                     (dec_action && !IsEventActive(TRCCNTCTLR[index].RLDEVENT)
                     )
                    );


    // The counter reloads if the reload event is active or the self-reload
    // mechanism causes a reload.
    reload = IsEventActive(TRCCNTCTLR[index].RLDEVENT) ||
            (TRCCNTCTLR[index].RLDSELF && dec_action && TRCCNTVR[index] == 0);


    // The counter only decrements if it is non-zero and does not reload and
    // dec_action is active.
    decrement = !reload && (TRCCNTVR[index] != 0) && dec_action;


    // Determine the next value of the counter
    if reload then
        TRCCNTVR[index] = TRCCNTRLDVR[index].VALUE;
    else if decrement then
        TRCCNTVR[index] = TRCCNTVR[index] - 1;
    else
        TRCCNTVR[index] = TRCCNTVR[index];


    CounterAtZero[index] = resource_active;
    return reload;
```

D7.8 Sequencer

I_{BGGRG} An ETE trace unit can contain a Sequencer state machine that has four states.
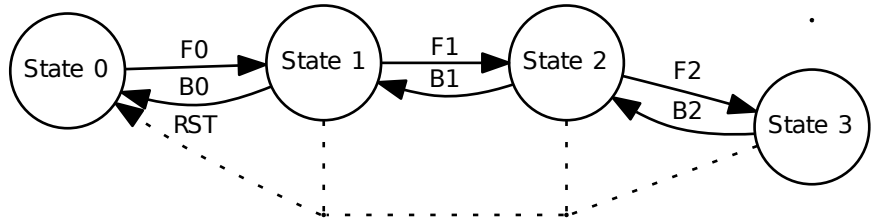
Figure D7.13: Sequencer state machine

I_{PTVBH} TRCIDR5.NUMSEQSTATE indicates whether the state machine is implemented.

R_{QYNJH} If the Sequencer state machine is implemented, it has 4 states, numbered 0 to 3.

R_{TQWHX} When the trace unit leaves the Disabled state, the Sequencer state machine starts in the state programmed in TRCSEQSTR.STATE.

I_{SYCBV} The Sequencer can be connected to resource events, so that the Sequencer transitions from one state to another when certain resource events occur. The TRCSEQEVR<n> registers can be used to specify which resource events cause the state machine to transition.

Each register can be used to specify the following:

- A resource event that causes the state machine to progress to the next state.
- A resource event that causes the state machine to transition backwards to the previous state.

Different resource events can be chosen to cause the Sequencer to transition between different states. For example, a particular resource event might cause an F0 transition from state 0 to state 1 on one processor clock cycle, whereas a different resource event might cause an F1 transition from state 1 to state 2 on the next processor clock cycle. A third independent resource event might cause a B1 transition backwards from state 2 to state 1 on the third clock cycle.

R_{NPVRQ} The trace unit prioritizes forward transitions over backward transitions in the Sequencer state machine. That is, when two resource events occur that result in a forward transition conflicting with a backward transition in the same processor clock cycle, the trace unit gives priority to the forward transition and ignores the backward transition.

I_{QNFJZ} The Sequencer can progress through multiple states in a single processor clock cycle. For example, if the Sequencer is in state 0 and the resource events that cause an F0 and F1 transition to take place both become active in one clock cycle, then the Sequencer progresses from state 0 to state 2.

I_{DMZGJ} The Sequencer can be reset to state 0 from any other state. TRCSEQRSTEVR can be used to specify a resource event to reset the Sequencer.

R_{HQBBF} When a resource event that causes an RST transition occurs, the Sequencer finishes the clock cycle in state 0 and does not progress to another state until the next clock cycle.

R_{KVSXC} The trace unit prioritizes RST transitions over all other transitions. That is, when a resource event that causes an RST transition is active in the same clock cycle as resource events that cause other transitions, the trace unit gives priority to the RST transition and ignores all other transitions.

R_{JDPYL} The table below defines all of the possible state transitions.

	To			
From	**0**	**1**	**2**	**3**
0	RST | !F0	F0 & !F1	F0 & F1 & !F2	F0 & F1 & F2
1	RST | (B0 & !F1 & !F0)	(!B0 | F0) & !F1	F1 & !F2	F1 & F2
2	RST | (B0 & B1 & !F2 & !F1 & !F0)	B1 & (!B0 | F0) & !F1 & !F2	(!B1 | F1) & !F2	F2
3	RST | (B0 & B1 & B2 & !F2 & !F1 & !F0)	B2 & B1 & (!B0 | F0) & !F2 & !F1	B2 & (!B1 | F1) & !F2	!B2 | F2

I_{YQZGV} If multiple resource events that cause transitions become active in one processor clock cycle, there is no guarantee that the order of these resource events becoming active is observed. For example, you might program:

- F0 to be active on an instruction Address Comparator at address `0x1000`.
- F1 to be active on an instruction Address Comparator at address `0x1004`.

If the instruction at `0x1000` and the instruction at `0x1004` are executed in the same processor clock cycle, then the transition from state 0 to state 2 occurs regardless of the program order of the two instructions.

R_{VDTDP} When the Sequencer state is changed by anything other than a direct write to TRCSEQSTR, the trace unit considers the change to be an indirect write to TRCSEQSTR.STATE.

I_{WYFZH} The ETE architecture provides each Sequencer state as a resource, so that states can be used to trigger other resource events in the trace unit.

```
┌─────────────────────┐
│  A resource event   │
└─────────────────────┘
          │
     results in
          ▼
┌─────────────────────┐
│    A transition     │
└─────────────────────┘
          │
     results in
          ▼
┌─────────────────────┐
│    A change of      │
│   Sequencer state   │
└─────────────────────┘
          │
     results in
          ▼
┌─────────────────────┐
│  A resource event   │
└─────────────────────┘
```
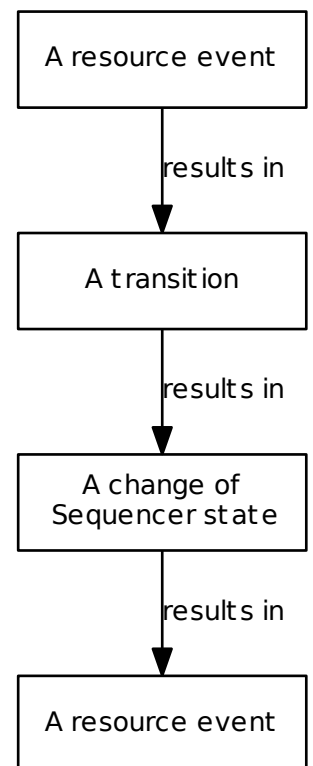
Figure D7.14: Sequencer operation

R_{HQHFT} When the Sequencer progresses through multiple states in a single processor clock cycle, for each state that it passes through, the resource state that the Sequencer triggers is active for that cycle.

I_{DCFMF} For example, if the Sequencer is in state 0, and in one processor clock cycle it moves to state 3, then the resource events that state 1 and state 2 are connected to must be active for that clock cycle. The same rule applies if the Sequencer is transitioning backwards, so that if it is in state 3, and in one processor clock cycle B2 and B1 cause it move to state 1, then the resource event that state 2 is connected to must be active for that clock cycle.

The exception to this is when a RST transition causes the Sequencer to return to state 0. For example, if the Sequencer is in state 3, and in one processor clock cycle it moves to state 0, then the resource events that state 2 and 1 are connected to must not become active.
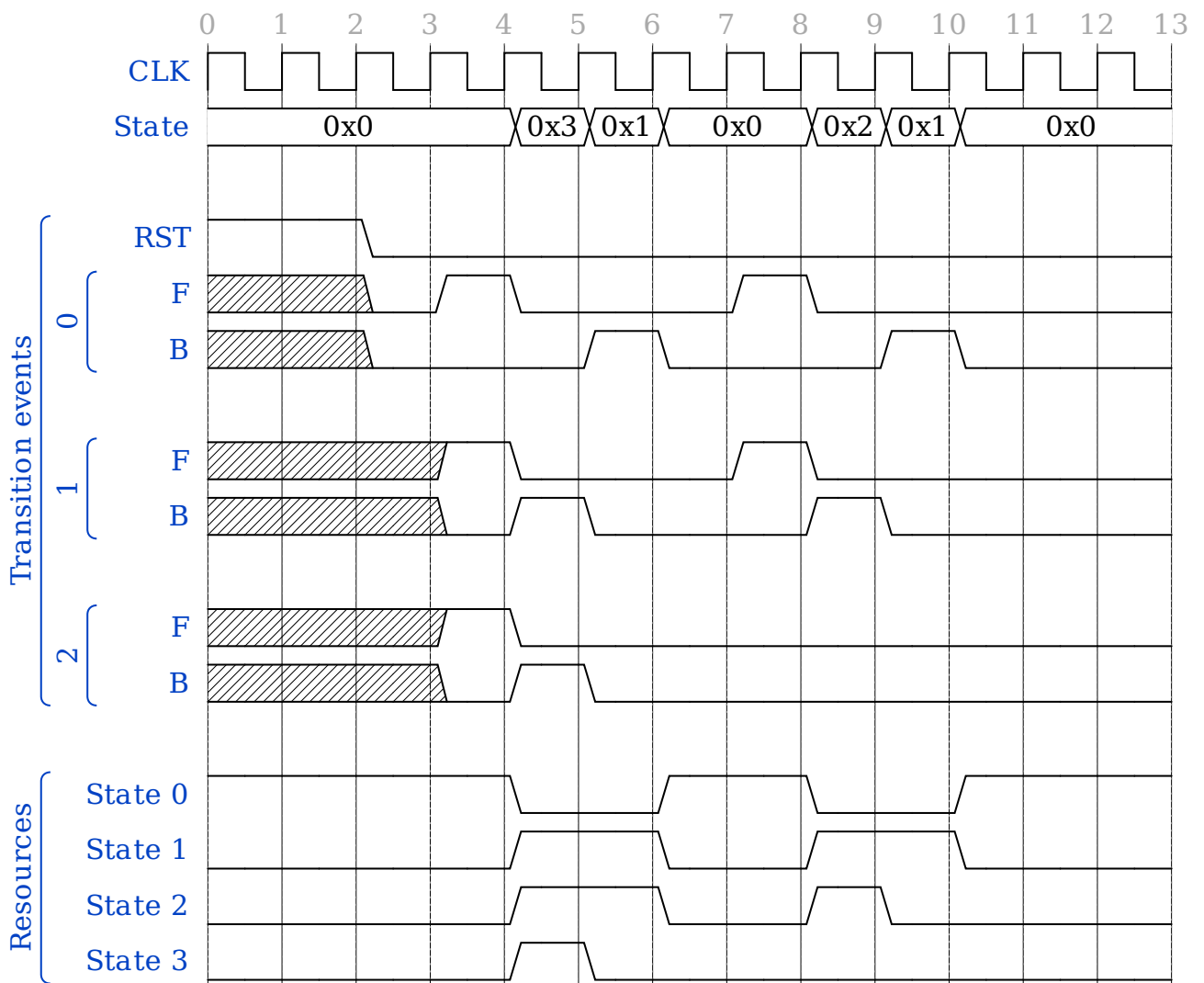
Figure D7.15: Example of State Transitions

D7.8.1 Pseudocode

D7.8.1.1 Sequencer Operation

```
// The sequencer state resources
array boolean SequencerState[0..3];


// EvalSequencer()
// ===============


EvalSequencer()
    (rst, txn0, txn1, txn2, txn3) = SequencerTransitions();
    // Sequencer State resources
    SequencerState[0] = FALSE;
    SequencerState[1] = FALSE;
    SequencerState[2] = FALSE;
    SequencerState[3] = FALSE;


    SequencerResource(rst, txn0, txn1, txn2, txn3);
    SequencerNextState(rst, txn0, txn1, txn2, txn3);
```

D7.8.1.2 Sequencer Transitions

```
// SequencerTransitions()
// ======================


( boolean rst,
  array boolean txn0[0..3],
  array boolean txn1[0..3],
  array boolean txn2[0..3],
  array boolean txn3[0..3]) SequencerTransitions()
    boolean F0 = IsEventActive(TRCSEQEVR0.F);
    boolean B0 = IsEventActive(TRCSEQEVR0.B);


    boolean F1 = IsEventActive(TRCSEQEVR1.F);
    boolean B1 = IsEventActive(TRCSEQEVR1.B);


    boolean F2 = IsEventActive(TRCSEQEVR2.F);
    boolean B2 = IsEventActive(TRCSEQEVR2.B);


    boolean rst = IsEventActive(TRCSEQRSTEVR);


    array boolean txn0[0..3];
    array boolean txn1[0..3];
    array boolean txn2[0..3];
    array boolean txn3[0..3];


    txn0[1] = F0 && !F1;
    txn0[2] = F0 &&  F1 && !F2;
    txn0[3] = F0 &&  F1 &&  F2;


    txn1[0] =   B0 && !F0  && !F1;
    txn1[1] = (!B0 ||  F0) && !F1;
    txn1[2] =                 F1 && !F2;
    txn1[3] =                 F1 &&  F2;


    txn2[0] =   B0 && !F0  &&  B1 && !F1  && !F2;
    txn2[1] = (!B0 ||  F0) &&  B1 && !F1  && !F2;
    txn2[2] =                (!B1 ||  F1) && !F2;
    txn2[3] =                                 F2;


    txn3[0] =   B0 && !F0  &&  B1 && !F1  && B2 && !F2;
    txn3[1] = (!B0 ||  F0) &&  B1 && !F1  && B2 && !F2;
    txn3[2] =                (!B1 ||  F1) && B2 && !F2;
    txn3[3] =                                (!B2 ||  F2);


    return (rst, txn0, txn1, txn2, txn3)
```

D7.8.1.3 Sequencer Resource

```
// SequencerResource()
// ===================


SequencerResource(boolean rst,
                  array boolean txn0[0..3],
                  array boolean txn1[0..3],
                  array boolean txn2[0..3],
                  array boolean txn3[0..3])


    case TRCSEQSTR.STATE of
        0 then SequencerState[0] = TRUE;
        1 then SequencerState[1] = TRUE;
        2 then SequencerState[2] = TRUE;
        3 then SequencerState[3] = TRUE;


    // If the statemachine passes through
    // several states in one iteration then
    // set the SequencerState as appropriate.
    if !rst then
        case TRCSEQSTR.STATE of
            0 then
                if txn0[2] then
                    SequencerState[1] = TRUE;
                if txn0[3] then
                    SequencerState[1] = TRUE;
                    SequencerState[2] = TRUE;
            1 then
                if txn1[3] then
                    SequencerState[1] = TRUE;
                    SequencerState[2] = TRUE;
            2 then
                if txn2[0] then
                    SequencerState[1] = TRUE;
            3 then
                if txn3[0] then
                    SequencerState[1] = TRUE;
                    SequencerState[2] = TRUE;
                if txn3[1] then
                    SequencerState[2] = TRUE;
```

D7.8.1.4 Sequencer Next State

```
// SequencerNextState()
// ====================


SequencerNextState(boolean rst,
                   array boolean txn0[0..3],
                   array boolean txn1[0..3],
                   array boolean txn2[0..3],
                   array boolean txn3[0..3])


    if rst then
        TRCSEQSTR.STATE = 0;
    else
        case TRCSEQSTR.STATE of
            0 then
                if txn0[1] then
                    TRCSEQSTR.STATE = 1;
                if txn0[2] then
                    TRCSEQSTR.STATE = 2;
                if txn0[3] then
                    TRCSEQSTR.STATE = 3;
            1 then
                if txn1[0] then
                    TRCSEQSTR.STATE = 0;
                if txn1[1] then
                    TRCSEQSTR.STATE = 1;
                if txn1[2] then
                    TRCSEQSTR.STATE = 2;
                if txn1[3] then
                    TRCSEQSTR.STATE = 3;
            2 then
                if txn2[0] then
                    TRCSEQSTR.STATE = 0;
                if txn2[1] then
                    TRCSEQSTR.STATE = 1;
                if txn2[2] then
                    TRCSEQSTR.STATE = 2;
                if txn2[3] then
                    TRCSEQSTR.STATE = 3;
            3 then
                if txn3[0] then
                    TRCSEQSTR.STATE = 0;
                if txn3[1] then
                    TRCSEQSTR.STATE = 1;
                if txn3[2] then
                    TRCSEQSTR.STATE = 2;
                if txn3[3] then
                    TRCSEQSTR.STATE = 3;
```

D7.9 Single-shot Comparator Controls

I_{QSKXC} If a trace unit is exposed to speculative execution or execution in Transactional state, when Address Comparators are used to activate resource events in the trace unit, then those resource events might be activated when speculative execution occurs:

- A Single Address Comparator might signal a match on speculative execution or within a transaction.
- An Address Range Comparator might signal a match on speculative execution or within a transaction.

For example, this means that if an Address Comparator is used to activate a Counter or assert an External Output, then that Counter might decrement, or that External Output might become asserted, as a result of speculative execution. The Single-shot Comparator Controls for Address Comparators make it possible for resource events in the trace unit to be activated based only on non-speculative execution, that is, only on architectural execution.

A trace unit can provide up to eight Single-shot Comparator Controls. Each Single-shot Comparator Control can be used with one or more Address Comparators.

I_{TLFLF} Single-shot Comparator Controls can be used as a trace unit resource, to activate trace unit resource events. For example, a Single-shot Comparator Control can be selected to:

- Enable or reload a trace unit Counter.
- Initiate a transition in the trace unit Sequencer state machine.
- Assert an External Output.

A Single-shot Comparator Control can therefore, if programmed to assert an External Output, be used to indicate to a trace analyzer that a particular instruction has been resolved for execution. This means that a trace analyzer can start or stop trace capture that is based on the architectural execution of that instruction.

I_{RBMXW} If a trace unit contains one or more Address Comparators, Arm recommends that at least one Single-shot Comparator Control is implemented.

I_{VNBPG} A Single-shot Comparator Control works in the following way:

1. One or more Address Comparators are selected by using the TRCSSCCR<n> for the Single-shot Comparator Control. The selected Address Comparators can be all Single Address Comparators, all Address Range Comparators, or a combination of both.

2. When one of the selected Address Comparators matches, then when the instruction is confirmed to have architecturally executed, the Single-shot Comparator Control fires.

When a selected Address Comparator matches, but the instruction is confirmed to have not architecturally executed, the Single-shot Comparator Control does not fire.

R_{XVVYX} When an instruction which matches an Address Comparator is confirmed to have architecturally executed, and the Address Comparator is selected by TRCSSCCR<n>, and TRCSSCSR<n>.STATUS is `0b0` or TRCSSCCR<n>.RST is `0b1`, the Single-shot Comparator Control <n> fires.

R_{XFJGB} When a `TSB CSYNC` instruction is executed while a Single-shot Comparator Control is programmed to fire due to the `TSB CSYNC` instruction, only when the related Trace synchronization event has completed, the Single-shot Comparator fires.

R_{SWNFV} When a Single-shot Comparator Control fires, the trace unit considers this an indirect write to set TRCSSCSR<n>.STATUS to `0b1`.

R_{GBDCK} While the resources are in the Paused state, when the conditions for a Single-shot Comparator Control to fire are met:

- If TRCSSCCR<n>.RST == `0b1` or TRCSSCSR<n>.STATUS == `0b0` then TRCSSCSR<n>.PENDING is indirectly written to `0b1`.
- If TRCSSCCR<n>.RST == `0b0` and TRCSSCSR<n>.STATUS == `0b1` then TRCSSCSR<n>.PENDING is either indirectly written to `0b1` or is unchanged.

R_{SDDWY} When one of the Address Comparators selected for a Single-shot Comparator Control matches, when the instruction that it matches on is in a Transaction which fails or is canceled, the Single-shot Comparator Control does not fire.

R_{NKKSN} When the trace unit becomes disabled and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction that is still speculative, the Single-shot Comparator Control does not fire.

R_{KFMKS} While the PE is executing in Transactional state, when the trace unit becomes disabled and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction in Transactional state, the Single-shot Comparator Control does not fire.

R_{DQZSD} When tracing becomes prohibited and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction that is still speculative, the Single-shot Comparator Control waits until the instruction speculation is resolved and fires if the instruction is architecturally executed.

R_{XRSYH} While the PE is executing in Transactional state, when tracing becomes prohibited and an Address Comparator selected by a Single-shot Comparator Control has matched on an instruction in Transactional state, the behavior of the Single-shot Comparator Control is IMPLEMENTATION DEFINED and is one of the following:

- The Single-shot Comparator Control does not fire.
- The Single-shot Comparator Control waits for the transaction to be resolved and fires if the transaction completes successfully.

R_{VTWXJ} While a Single-shot Comparator Control is used for instruction address comparisons, when the conditions for the Single-shot Comparator Control to fire are met, the Single-shot Comparator Control fires, regardless of whether either of the following are true:

- The instruction fails its condition code check.
- The instruction is a failed store-exclusive operation.

I_{XZKFW} When a Single-shot Comparator Control is used to activate a resource event, the resource event might not become activated until some time after the trace unit has traced the instruction. This is because although the trace unit traces the instruction it is executed, the PE might not confirm whether the instruction was architecturally executed or canceled because of mis-speculation until some time later, and therefore the Single-shot Comparator Control might not fire until some time later.

D7.9.1 Single-shot Comparator Control modes

I_{XZJSV} Each Single-shot Comparator Control operates in one of the following modes:

- Single-shot mode: The Single-shot Comparator Control only fires once. That is, after it has fired, it never fires again.
- Multi-shot mode: The Single-shot Comparator Control resets after each time it fires. That is, it can fire again when a selected Address Comparator next signals an address match for an instruction is architecturally executed.

TRCSSCCR<n>.RST selects the mode.

R_{KJBCH} While a Single-shot Comparator Control is in multi-shot mode, when the Single-shot Comparator Control fires, it fires for a maximum of one processor clock cycle.

R_{SNDBZ} While a Single-shot Comparator Control is in multi-shot mode, when multiple of the comparators selected for the Single-shot Comparator Control match in close succession, only the first match is guaranteed to cause the Single-shot Comparator Control to fire.

I_{HSGTY} Examples of multiple comparator matches in close succession include:

- More than one of the Address Comparators that are selected signal an address match simultaneously.
- One Address Comparator matches multiple times while a first match is still waiting to be resolved.

D7.9.2 Operation while in Paused state

I_{SPQLS} The resolution of a speculative instruction might occur after the PE has entered a Trace Prohibited region and the resources have entered the Paused state. If the conditions for the Single-shot Comparator Control to fire are met while the resources are in the Paused state then the Single-shot Comparator Control resource event is delayed to ensure that the Single-shot Comparator Control resource event is seen.

R_{TQHNK} While the resources are in the Paused state, the Single-shot Comparator Controls do not fire.

R_{PVRGR} When the resources enter the Running state while TRCSSCSR<n>.PENDING is `0b1`, the following occur:

- If TRCSSCCR<n>.RST == `0b1` or TRCSSCSR<n>.STATUS == `0b0`, the Single-shot Comparator Control fires.
- TRCSSCSR<n>.PENDING is indirectly written to `0b0`.

D7.10 External Outputs

I_{BZHDF} The ETE architecture supports between one and four External Outputs. The number of outputs that a trace unit has is IMPLEMENTATION DEFINED, but at least one output is always implemented.

I_{QPQFJ} External Outputs are used to indicate ETEEvents to a trace analyzer.

ETEEvents are controlled by resources events. For example, an instruction Address Comparator can be used to drive one of the resource events.

If an External Output is programmed to be asserted based on program execution, such as an Address Comparator, the External Output might not be asserted at the same time as any trace generated by that program execution is output by the trace unit.

I_{PNVWQ} Typically, the generated trace might be buffered in a trace unit which means that the External Output would be asserted before the trace is output.

S_{MFTNW} To program an External Output, use TRCEVENTCTL0R to select a resource.

S_{RBKWB} The TRCIDR0.NUMEVENT field shows how many ETEEvents are supported for the particular implementation.

I_{RFLGF} The External Outputs are connected to the Cross Trigger Interface (CTI) for the PE, as defined in *Arm® Architecture Reference Manual, for A-profile architecture* [1].

R_{RFGJD} In a PE where the Trace Unit reset is independent of the PE Warm reset and the *Cross-trigger Interface* (CTI) reset is independent of the PE Warm reset, transmission of External Outputs to the CTI is unaffected by a PE Warm reset.

D7.10.1 Operation while in Paused state

X_{NXCSB} While the resources are in the Paused state an ETEEvent might occur, but any associated trace packets might not be generated. TRCRSR.EVENT provides a mechanism for recording this occurrence so that the trace unit state can be saved and restored.

R_{BCMYM} While the resources are in the Paused state, the ETEEvent selector retains that one or more ETEEvents have been generated but not traced.

R_{SNKFL} When an ETEEvent has been generated and the associated External Output has been asserted, any associated Event packets are generated.

R_{FVCMB} When an ETEEvent has been generated but the associated External Output has not been asserted, any associated Event packets are not generated.

R_{GYWLS} When an ETEEvent occurs while the resources are in the Paused state and the Event packet is not output, the trace unit sets the associated TRCRSR.EVENT[n] to 0b1.

R_{DCLHJ} When an ETEEvent occurs while the resources are in the Paused state, this is considered an indirect write to TRCRSR.

R_{SWBRL} When the trace unit enters the Running state while TRCRSR.EVENT[n] is 0b1, the associated ETEEvent resource is active for a single PE clock cycle, and the trace unit clears TRCRSR.EVENT[n] to 0b0 and considers the action an indirect write to TRCRSR.

I_{KZYKM} When the trace unit enters the Running state while TRCRSR.EVENT[n] is 0b1, the resource event selected by TRCEVENTCTL0R.EVENT<n> might also be active on the same PE clock cycle. If this happens, the associated ETEEvent resource is active for the single PE clock cycle and might not generate 2 separate ETEEvents for these 2 causes of the ETEEvent.

D7.11 External Inputs

I_{TPPSC} The trace unit uses the PMU events as External Inputs.

R_{MTGKB} If a PMU event number that is selected is not implemented, then the External Input resource event is inactive.

R_{YCNCR} Unless otherwise specified by the PMU event, the following PMU events are selectable by the trace unit:

- All PMU events required by the Performance Monitors Extension.
- If FEAT_PMUv3 is implemented, all Common architectural and microarchitectural events implemented by the Performance Monitors Extension.

Note

This includes Common events defined by other extensions, such as SVE and *Statistical Profiling Extension* (SPE).

I_{XJBHV} Additional ASIC-specific events can be selected by using a number in the IMPLEMENTATION DEFINED region.

I_{VWHTZ} There is no requirement that all IMPLEMENTATION DEFINED events are visible by the trace unit, PMU counters and the PMUEVENT bus.

R_{PHDQT} For ETE, export of PMU events to the trace unit is not affected by PMCR.X or PMCR_EL0.X.

R_{RFWZB} When SelfHostedTraceEnabled() == TRUE and tracing is prohibited, only the PMU events defined by rules VBCBZ and KRSMY are exported to the trace unit.

R_{WSXTC} When SelfHostedTraceEnabled() == FALSE and the PE is in Secure state and counting in Secure state is prohibited, only the PMU events defined by rules VBCBZ and KRSMY are exported to the trace unit.

R_{VBCBZ} The following PMU events are always exported to the trace unit:

- CTI_TRIGOUT4.
- CTI_TRIGOUT5.
- CTI_TRIGOUT6.
- CTI_TRIGOUT7.
- PMU_OVFS.
- TRB_WRAP.
- TRB_TRIG.

R_{KRSMY} The following PMU events are always exported to the trace unit, unless SelfHostedTraceEnabled() == TRUE and TRFCR_EL2.E2TRE is 0b0:

- PMU_HOVFS.

R_{QPDHK} When multiple occurrences of the same PMU event occur during the same cycle, the trace unit only observes a single occurrence of the PMU event.

I_{MHHNV} The operation of the PMU events and the generation of trace within the trace unit are not tightly coupled, and there is no guarantee that enabling ViewInst due to a PMU event will cause the instruction that caused the PMU event to be traced.

R_{XGMPN} When the PMU event SW_INCR is selected as an External Input and PMSWINC_EL0 is written from EL2 or EL3, the External Input is asserted if any bit [n] written has the value 0b1 and the relevant PMU counter <n> is implemented.

R_{BXPZK} When the PMU event SW_INCR is selected as an External Input and PMSWINC_EL0 is written from EL1 or EL0, the External Input is asserted if any bit [n] written has the value 0b1 and the relevant PMU counter <n> is implemented and any of the following are true:

- EL2 is implemented and enabled in the current Security state and <n> is less than MDCR_EL2.HPMN.

- EL2 is implemented and disabled in the current Security state.
- EL2 is not implemented.

R_{TTPPY} In a PE where the Trace Unit reset is independent of the PE Warm reset and the CTI reset is independent of the PE Warm reset, transmission of the CTI_TRIGOUTn events from the CTI to the trace unit is unaffected by a PE Warm reset.

D7.11.1 Operation while in Paused state

X_{HZLDV} The External Input Selectors are guaranteed to be active while in the Paused state. This is so that while the resources are Paused any cross trigger event is not lost but will occur when the resources resume running.

TRCRSR.EXTIN provides a mechanism to capture the sticky state of the External Input Selectors while in the Paused state so that the ETE state can be saved and restored.

I_{ZQFND} When one or more selected External Inputs have been asserted, while the resources are in the Paused state, the trace unit retains the knowledge that one or more selected External Inputs have been asserted.

R_{KCXLF} While the resources are in the Pausing or Paused states and the trace unit is not disabled and is not in the low-power state, when an External Input Selector n detects the selected External Input is asserted, the trace unit performs an indirect write to set TRCRSR.EXTIN[n] to $0b1$.

R_{QWYSK} When the resources enter the Running state while TRCRSR.EXTIN[n] is $0b1$, the External Input Selector resource is active for a single PE clock cycle, and the trace unit clears TRCRSR.EXTIN[n] and considers the action an indirect write to TRCRSR.

D7.11.2 Operation while in the Low-power state

R_{KVFVS} While the trace unit is in the low-power state, the External Input Selectors are inactive.

D7.12 PE Comparator Inputs

I_{CXBPR} The ETE architecture provides up to eight PE Comparator Inputs, that is, inputs that can be driven from comparators within the PE. For example, a PE might contain IMPLEMENTATION DEFINED comparators.

R_{CNVSS} The number of PE Comparator Inputs is indicated by TRCIDR4.NUMPC.

R_{BDWHM} While the PE is executing in a Trace Prohibited region, the PE Comparator Inputs are inactive.

R_{TNHHY} The PE Comparator Inputs are only used in IMPLEMENTATION SPECIFIC code.

I_{DDXFB} Each PE Comparator Input can be used in any of the following ways:

- To control the ViewInst start/stop function.
- To control the Single-shot Comparator Controls.
- As an independent resource.

I_{SKDCW} The behavior of the PE Comparator Inputs on the resources and the filtering of the trace unit is IMPLEMENTATION DEFINED.

Chapter D8
Register Description

D8.1 Accessing ETE registers

I_{VYNRB} The ETE architecture provides registers for programming the trace unit and reading back the programmed settings. These registers can be accessed by using one or more of the following access mechanisms:

- System instructions, for use by self-hosted software running on the *Processing Element* (PE) being traced.
- An external debugger interface, for use by an external debugger.

R_{NBPML} When register accesses occur simultaneously from multiple access mechanisms, the trace unit behaves as if all accesses occur atomically in any order.

D8.1.1 External debugger interface

I_{KPYGY} The external debugger interface defines an address-mapped peripheral that occupies 4KB of address space.

> **Note**
>
> The PE does not have to be in Debug state to program the trace unit registers.

I_{KFRVP} The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral.

R_{VQWLY} The trace unit supports the following access sizes:

- Word-aligned 32-bit accesses to access 32-bit registers or either half of a 64-bit register mapped to a doubleword-aligned pair of adjacent 32-bit locations.

- Doubleword-aligned 64-bit accesses to access 64-bit registers mapped to a doubleword-aligned pair of adjacent 32-bit locations. The order in which the two halves are accessed is not specified.

Note

This means that a system implementing the debug registers using a 32-bit bus, such as AMBA APB in CoreSight systems, with a wider system interconnect must implement a bridge between the system and the debug bus that can split 64-bit accesses.

R_{VNNPF} All registers are only single-copy atomic at word granularity.

R_{KYDTQ} The trace unit does not support the following accesses:

- Byte.
- Halfword.
- Unaligned word. These accesses are not single-copy atomic at word granularity.
- Unaligned doubleword. These accesses are not single-copy atomic at doubleword granularity.
- Doubleword accesses to a pair of 32-bit locations that are not a doubleword-aligned pair forming a 64-bit register.
- Quadword or higher.
- Exclusives.

R_{YFRMG} The behavior of the accesses that are unsupported by the trace unit is CONSTRAINED UNPREDICTABLE and is one of the following:

- Accesses generate an External Abort, and writes set the accessed register or registers to an UNKNOWN value or values.
- Reads return UNKNOWN data and writes are ignored.
- Reads return UNKNOWN data and writes set the accessed register or registers to an UNKNOWN value. This is the Arm preferred behavior.

Note

For accesses from the external debugger interface, the size of an access is determined by the interface. In an Arm Debug Interface compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.

Note

The CoreSight APB-AP supports only word accesses.

R_{YSHRS} For accesses from the external debugger interface which return an error response when `AllowExternalTraceAccess()` returns FALSE, EDPRSR.STAD is only set to 0b1 when this is the highest priority cause of the error. The following causes are higher priority than `AllowExternalTraceAccess()`:

- The trace unit core power domain is powered down.
- The OS Lock is locked and the register is defined to return an error response due to the OS Lock being locked.

R_{KQMKX} Accesses from the external debugger interface to Unimplemented or Reserved registers behave as follows:

- For accesses in the range of offsets `0xF00` to `0xFFC`, the access behaves as RES0H.
- For accesses in the range of offsets `0x000` to `0xEFC` when the OS Lock is locked, the access behaves as RES0H or returns an error.

- For accesses in the range of offsets `0x000` to `0xEFC` when the OS Lock is unlocked and MDCR_EL3.ETAD == 0b0, the access behaves as RES0H.
- For Secure accesses in the range of offsets `0x000` to `0xEFC` when the OS Lock is unlocked and MDCR_EL3.ETAD == 0b1, the access behaves as RES0H.
- For Non-secure accesses in the range of offsets `0x000` to `0xEFC` when the OS Lock is unlocked and MDCR_EL3.ETAD == 0b1, the access behaves as RES0H or returns an error.

R_{WXKDP} Reads of write-only registers are Reserved.

R_{SVSNR} Writes to read-only registers are Reserved.

I_{WTJFD} For accesses that return an error, see *Arm® Architecture Reference Manual, for A-profile architecture* [1] for more details on how this error is handled.

D8.1.2   System instructions

D8.1.2.1   AArch64

R_{JVLBN} MRS instructions with `op0 == 0b10` and `op1 == 0b001` read from trace unit registers.

R_{FYRRX} MSR instructions with `op0 == 0b10` and `op1 == 0b001` write to trace unit registers.

R_{VGVTS} Instructions with `CRn >= 0b1000` are UNDEFINED.

R_{SGPQB} While the PE is in EL0, all accesses are UNDEFINED.

I_{WCXDT} For consistency with the Arm architecture, system instruction accesses to registers which are not implemented generate an Undefined instruction exception. These accesses include:

- Writes to read-only registers.
- Reads from write-only registers.
- Accesses to registers which are not present due to the configuration of the trace unit.

D8.2 Synchronization of register updates

D8.2.1 AArch64 system registers

I_{KWCGH} As defined in *Arm® Architecture Reference Manual, for A-profile architecture* [1], direct writes to trace unit registers are only guaranteed to be visible to execution after a subsequent Context synchronization event, which consists of one of the following:

- Taking an exception.
- Returning from an exception.
- Performing an Instruction Synchronization Barrier operation.
- Exit from Debug state.
- Executing a DCPS instruction in Debug state.
- Executing a DRPS instruction in Debug state.

I_{PNZZH} Direct reads of trace unit registers while the trace unit is not in the Stable or Idle states are not guaranteed to contain the results of the trace operation of execution previous to the direct read operation.

I_{QPVJQ} As defined in *Arm® Architecture Reference Manual, for A-profile architecture* [1], a direct write to a register using the same register number that was used by a previous system instruction to write it, the final result is the value of the second write, without requiring any context synchronization between the two write instructions.

I_{WLGQK} As defined in *Arm® Architecture Reference Manual, for A-profile architecture* [1], a direct read of a register using the same register number that was used by an earlier direct write is guaranteed to observe the value that was written, without requiring any context synchronization between the write and read instructions.

S_{HSXGZ} Context synchronization events are important when changing the value of TRCPRGCTLR.EN or when changing the OS Lock. After writing to TRCPRGCTLR to change the value of TRCPRGCTLR.EN, one read of TRCSTATR is required before programming any other registers. A Context synchronization event is required between writing to TRCPRGCTLR and reading TRCSTATR. If multiple reads of TRCSTATR are required, a Context synchronization event is required between each read of TRCSTATR to ensure any change to the trace unit state is observed.

R_{WPWWS} When indirect writes or external writes to the registers in Table D8.1 occur, both of the following can observe the writes:

- Direct reads in finite time without explicit synchronization.
- Subsequent indirect reads without explicit synchronization.

Table D8.1: Registers with a guarantee of observability

Register	Notes
TRCCLAIMSET	Claim Tag Set Register
TRCCLAIMCLR	Claim Tag Clear Register
TRCCNTVR\<n\>	Counter Value Register \<n\>
TRCSEQSTR	Sequencer State Register
TRCSSCSR\<n\>	Single-shot Comparator Control Status Register

R_{JQTMC} When the trace unit becomes enabled or disabled as a result of a direct write, for any instruction in program order before the direct write, the new state of the trace unit does not affect trace operations.

R_{KNQWS} When the trace unit becomes enabled or disabled as a result of a direct write, for any instruction after a Context synchronization event in program order after the direct write, the new state of the trace unit takes effect for any trace operations.

Note

The registers which control whether the trace unit is enabled or disabled are:

- TRCPRGCTLR.
- OSLAR_EL1.

S_{YKGMP} Arm recommends that a Context synchronization event is executed after programming the trace unit registers, to ensure that all updates are synchronized to the trace unit before normal code execution resumes.

R_{WZQWC} When a Context synchronization event occurs while the trace unit is in the Idle or Stable states, and at no other time, indirect writes to the trace unit registers are guaranteed to be visible to direct reads.

R_{GQKGX} When either of the following events occurs, and at no other time, indirect writes to the trace unit registers are guaranteed to be visible to indirect reads or external reads:

- The trace unit transitions into the Stable state.
- The trace unit transitions into the Idle state.

R_{XLXQL} The trace unit functions perform indirect writes to the registers and indirect reads from the registers in architectural order.

See D8.3 *Trace unit programming states* for more details on programming the trace unit.

D8.2.2   External Debugger registers

I_{KNWDX} As defined in the "Synchronization of changes to the external debug registers" chapter of *Arm® Architecture Reference Manual, for A-profile architecture* [1], this section refers to accesses from the external debug interface as *external reads* and *external writes*.

I_{HMNWB} As defined in *Arm® Architecture Reference Manual, for A-profile architecture* [1], explicit synchronization is not required for an external read or an external write by an external agent to be observable to a following external read or external write by that agent to the same register using the same address, so explicit synchronization is never required for registers that are accessible only in the external debug interface.

I_{YXWFD} As defined in *Arm® Architecture Reference Manual, for A-profile architecture* [1], when an external write to a register using the same register number that was used by a previous external write is performed, the final result is the value of the second write, without requiring any context synchronization between the two write accesses.

R_{PGTLX} The trace unit does not require explicit synchronization for an external write to be visible to indirect reads.

R_{DYRZC} The trace unit does not require explicit synchronization for an external write to be visible to subsequent external reads.

I_{RDFSX} As defined in *Arm® Architecture Reference Manual, for A-profile architecture* [1], explicit synchronization is required for an external write to be visible to direct reads.

R_{MMYRJ} While the trace unit is in the Stable or Idle states, the trace unit does not require explicit synchronization for indirect writes to be visible to external reads.

D8.2.3   Synchronization and the authentication interface

R_{WYWMJ} Changes to the authentication interface are indirect writes to TRCAUTHSTATUS by the controller of the authentication interface. It is IMPLEMENTATION DEFINED whether a change on the authentication interface is guaranteed to be observable to an external read of the register only after a Context synchronization event or in finite time.

D8.3 Trace unit programming states

R_{WMGGP} The trace unit is always in one of the states shown in Figure D8.1 and Table D8.2.

R_{DXFFN} When the trace unit becomes enabled, the trace unit transitions from the Idle state to the Enabling state.

R_{ZFPHC} The trace unit transitions from the Enabling state to the Running state in a finite amount of time.

R_{TZSRP} When the trace unit becomes disabled, the trace unit transitions from the Running state to the Unstable state.

R_{CZHWF} The trace unit transitions from the Unstable state to the Stable state in a finite amount of time.

R_{BPTKT} While the trace unit is in the Stable and Idle states, the states of the following fields do not change other than via direct writes or external writes:

- TRCVICTLR.SSSTATUS.
- TRCSEQSTR.STATE.
- TRCCNTVR<n>.VALUE.
- TRCSSCSR<n>.STATUS.
- TRCRSR.EVENT.
- TRCRSR.EXTIN.
- TRCRSR.TA.

I_{TDLZL} The trace unit programmers' model state can be safely read when in any of the Stable or Idle states.

R_{TRZCP} When the trace unit becomes fully idle and both of the following are true, the trace unit transitions from the Stable state to the Idle state:

- The trace unit is drained of any trace.
- With the exception of the programming interfaces, all external interfaces on the trace unit are quiescent.
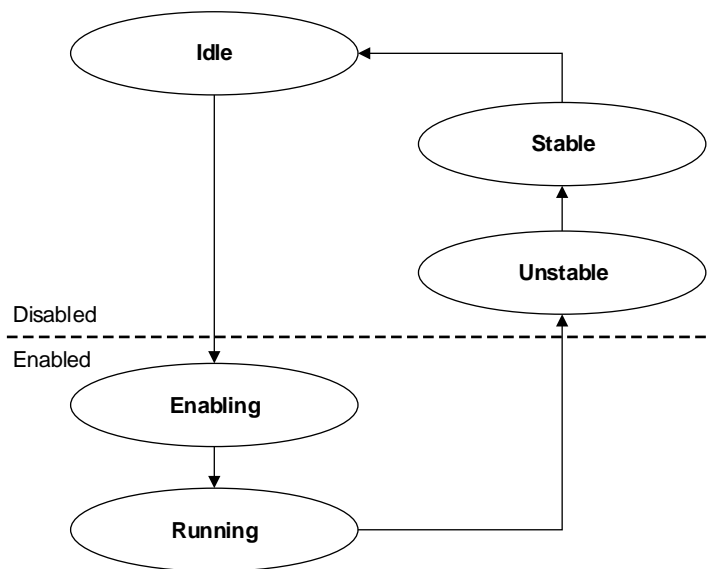
Figure D8.1: Trace unit programming states

Table D8.2: Trace unit programming states

State	TRCSTATR.IDLE	TRCSTATR.PMSTABLE	Trace unit enabled
Idle	0b1	0b1	No
Enabling	0b1	UNKNOWN	Yes
Running	0b0	UNKNOWN	Yes
Unstable	0b0	0b0	No
Stable	0b0	0b1	No

R_{RWYQD} While the trace unit is not in the Idle state, direct writes and external writes to the trace unit registers are CONSTRAINED UNPREDICTABLE, except for the following registers:

- TRCPRGCTLR.
- TRCCLAIMSET.
- TRCCLAIMCLR.

This CONSTRAINED UNPREDICTABLE behavior is one of the following:

- The write is ignored.
- The register takes an UNKNOWN value.

The trace byte stream might also be corrupted and analysis of the byte stream might be impossible.

R_{MDZDN} While the trace unit is not in the Idle or Running states, changing the value of TRCPRGCTLR.EN is CONSTRAINED UNPREDICTABLE.

This CONSTRAINED UNPREDICTABLE behavior is one of the following:

- The write is ignored.
- The register takes an UNKNOWN value.

I_{PRQRD} For more information, see:

- D6.2.1 *Behavior on enabling*.
- D6.2.2 *Behavior on disabling*.
- Access permissions on the corresponding register page.

I_{FJPCN} Figure D8.2 shows the procedure that must be used when programming the trace unit registers using the External Debugger interface.
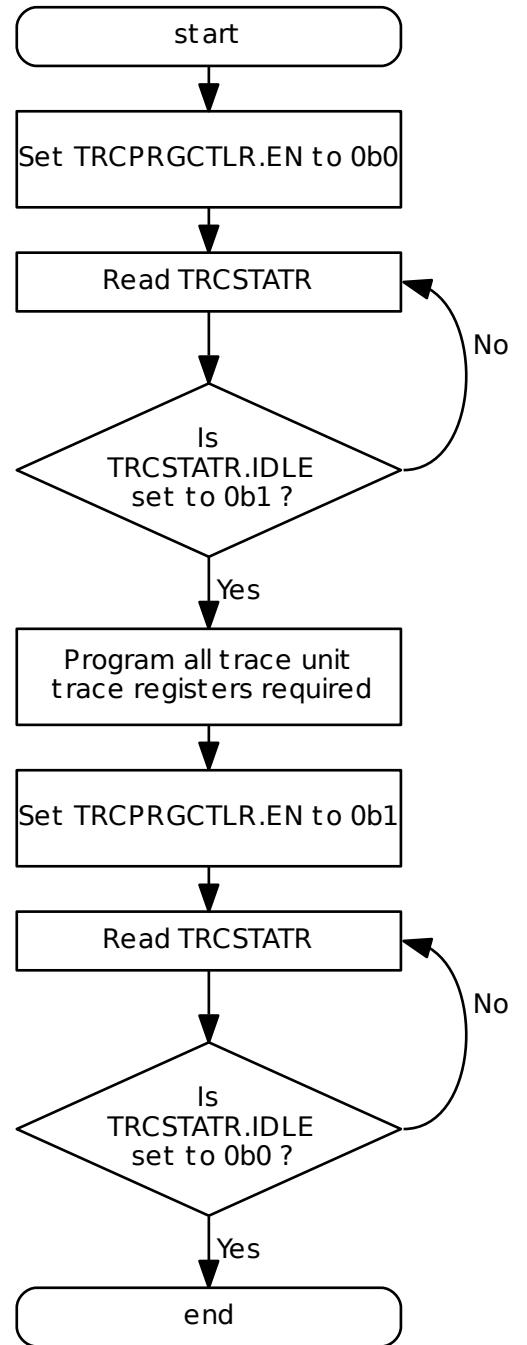
Figure D8.2: External Debugger Interface programming procedure

I$_{ZGJRF}$ Figure D8.3 shows the procedure that is used when programming the trace unit registers using the System instruction interface.
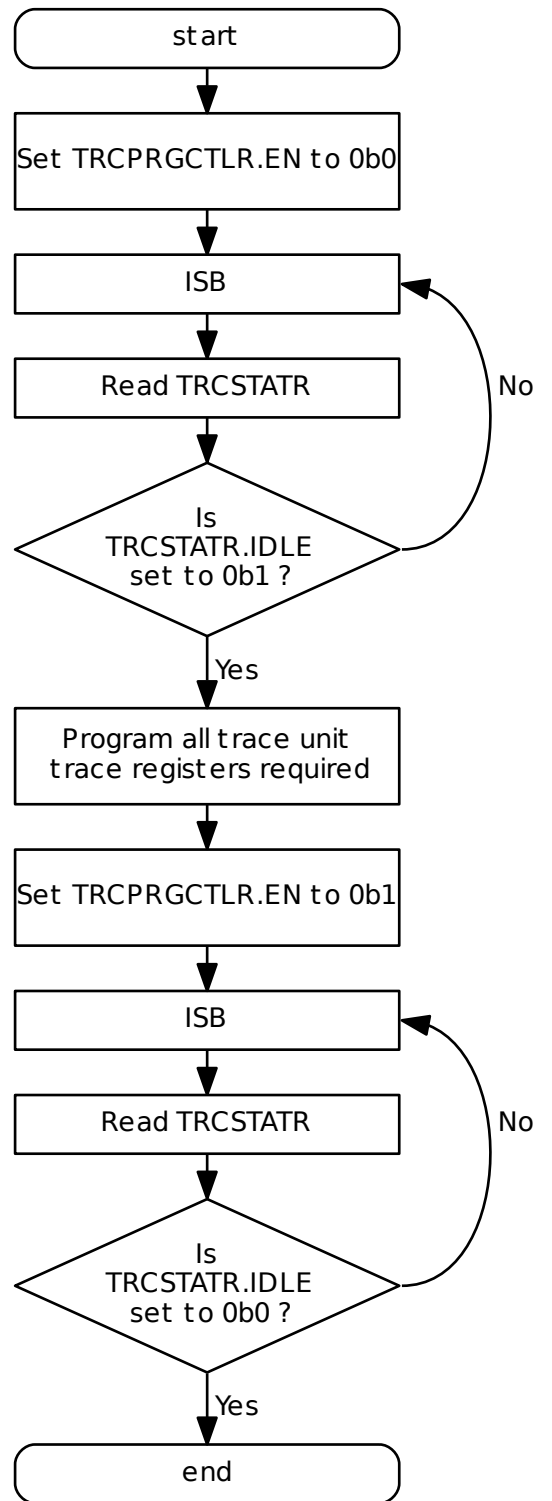
Figure D8.3: System instruction programming procedure

D8.4 External debug registers

D8.4.1 Trace registers, external debug register map

Offset	Access	Register	Description
0x004	R/W	TRCPRGCTLR	Programming Control Register
0x00C	RO	TRCSTATR	Trace Status Register
0x010	R/W	TRCCONFIGR	Trace Configuration Register
0x018	R/W	TRCAUXCTLR	Auxiliary Control Register
0x020	R/W	TRCEVENTCTL0R	Event Control 0 Register
0x024	R/W	TRCEVENTCTL1R	Event Control 1 Register
0x028	R/W	TRCRSR	Resources Status Register
0x02C	R/W	TRCSTALLCTLR	Stall Control Register
0x030	R/W	TRCTSCTLR	Timestamp Control Register
0x034	R/W	TRCSYNCPR	Synchronization Period Register
0x038	R/W	TRCCCCTLR	Cycle Count Control Register
0x03C	R/W	TRCBBCTLR	Branch Broadcast Control Register
0x040	R/W	TRCTRACEIDR	Trace ID Register
0x044	R/W	TRCQCTLR	Q Element Control Register
0x080	R/W	TRCVICTLR	ViewInst Main Control Register
0x084	R/W	TRCVIIECTLR	ViewInst Include/Exclude Control Register
0x088	R/W	TRCVISSCTLR	ViewInst Start/Stop Control Register
0x08C	R/W	TRCVIPCSSCTLR	ViewInst Start/Stop PE Comparator Control Register
0x100+4×n	R/W	TRCSEQEVR	Sequencer State Transition Control Register
0x118	R/W	TRCSEQRSTEVR	Sequencer Reset Control Register
0x11C	R/W	TRCSEQSTR	Sequencer State Register
0x120+4×n	R/W	TRCEXTINSELR	External Input Select Register
0x140+4×n	R/W	TRCCNTRLDVR	Counter Reload Value Register
0x150+4×n	R/W	TRCCNTCTLR	Counter Control Register
0x160+4×n	R/W	TRCCNTVR	Counter Value Register
0x180	RO	TRCIDR8	ID Register 8
0x184	RO	TRCIDR9	ID Register 9
0x188	RO	TRCIDR10	ID Register 10
0x18C	RO	TRCIDR11	ID Register 11
0x190	RO	TRCIDR12	ID Register 12
0x194	RO	TRCIDR13	ID Register 13

Offset	Access	Register	Description
0x1C0	R/W	TRCIMSPEC0	IMP DEF Register 0
0x1C0+4×n	R/W	TRCIMSPEC	IMP DEF Register
0x1E0	RO	TRCIDR0	ID Register 0
0x1E4	RO	TRCIDR1	ID Register 1
0x1E8	RO	TRCIDR2	ID Register 2
0x1EC	RO	TRCIDR3	ID Register 3
0x1F0	RO	TRCIDR4	ID Register 4
0x1F4	RO	TRCIDR5	ID Register 5
0x1F8	RO	TRCIDR6	ID Register 6
0x1FC	RO	TRCIDR7	ID Register 7
0x200+4×n	R/W	TRCRSCTLR	Resource Selection Control Register
0x280+4×n	R/W	TRCSSCCR	Single-shot Comparator Control Register
0x2A0+4×n	R/W	TRCSSCSR	Single-shot Comparator Control Status Register
0x2C0+4×n	R/W	TRCSSPCICR	Single-shot Processing Element Comparator Input Control Register
0x400+8×n	R/W	TRCACVR[31:0]	Address Comparator Value Register , bits[31:0]
0x404+8×n	R/W	TRCACVR[63:32]	Address Comparator Value Register , bits[63:32]
0x480+8×n	R/W	TRCACATR[31:0]	Address Comparator Access Type Register , bits[31:0]
0x484+8×n	R/W	TRCACATR[63:32]	Address Comparator Access Type Register , bits[63:32]
0x600+8×n	R/W	TRCCIDCVR[31:0]	Context Identifier Comparator Value Registers , bits[31:0]
0x604+8×n	R/W	TRCCIDCVR[63:32]	Context Identifier Comparator Value Registers , bits[63:32]
0x640+8×n	R/W	TRCVMIDCVR[31:0]	Virtual Context Identifier Comparator Value Register , bits[31:0]
0x644+8×n	R/W	TRCVMIDCVR[63:32]	Virtual Context Identifier Comparator Value Register , bits[63:32]
0x680	R/W	TRCCIDCCTLR0	Context Identifier Comparator Control Register 0
0x684	R/W	TRCCIDCCTLR1	Context Identifier Comparator Control Register 1
0x688	R/W	TRCVMIDCCTLR0	Virtual Context Identifier Comparator Control Register 0
0x68C	R/W	TRCVMIDCCTLR1	Virtual Context Identifier Comparator Control Register 1
0xFA0	R/W	TRCCLAIMSET	Claim Tag Set Register
0xFA4	R/W	TRCCLAIMCLR	Claim Tag Clear Register

D8.4.2 Management registers, external debug register map

Offset	Access	Register	Description
0x304	RO	TRCOSLSR	Trace OS Lock Status Register
0x310	R/W	TRCPDCR	PowerDown Control Register
0x314	RO	TRCPDSR	PowerDown Status Register

Offset	Access	Register	Description
0xF00	R/W	TRCITCTRL	Integration Mode Control Register
0xFA8	RO	TRCDEVAFF[31:0]	Device Affinity Register, bits[31:0]
0xFAC	RO	TRCDEVAFF[63:32]	Device Affinity Register, bits[63:32]
0xFB0	WO	TRCLAR	Lock Access Register
0xFB4	RO	TRCLSR	Lock Status Register
0xFB8	RO	TRCAUTHSTATUS	Authentication Status Register
0xFBC	RO	TRCDEVARCH	Device Architecture Register
0xFC0	RO	TRCDEVID2	Device Configuration Register 2
0xFC4	RO	TRCDEVID1	Device Configuration Register 1
0xFC8	RO	TRCDEVID	Device Configuration Register
0xFCC	RO	TRCDEVTYPE	Device Type Register
0xFD0	RO	TRCPIDR4	Peripheral Identification Register 4
0xFD4	RO	TRCPIDR5	Peripheral Identification Register 5
0xFD8	RO	TRCPIDR6	Peripheral Identification Register 6
0xFDC	RO	TRCPIDR7	Peripheral Identification Register 7
0xFE0	RO	TRCPIDR0	Peripheral Identification Register 0
0xFE4	RO	TRCPIDR1	Peripheral Identification Register 1
0xFE8	RO	TRCPIDR2	Peripheral Identification Register 2
0xFEC	RO	TRCPIDR3	Peripheral Identification Register 3
0xFF0	RO	TRCCIDR0	Component Identification Register 0
0xFF4	RO	TRCCIDR1	Component Identification Register 1
0xFF8	RO	TRCCIDR2	Component Identification Register 2
0xFFC	RO	TRCCIDR3	Component Identification Register 3

D8.4.3 Integration registers

Table D8.5: Integration registers, external debug register map

Offset	Access	Register	Description
0xE80 to 0xEFC	R/W	-	Reserved for IMPLEMENTATION DEFINED integration and topology detection registers.

Chapter D9
Trace Analyzer

This chapter describes a simple trace decompressor.
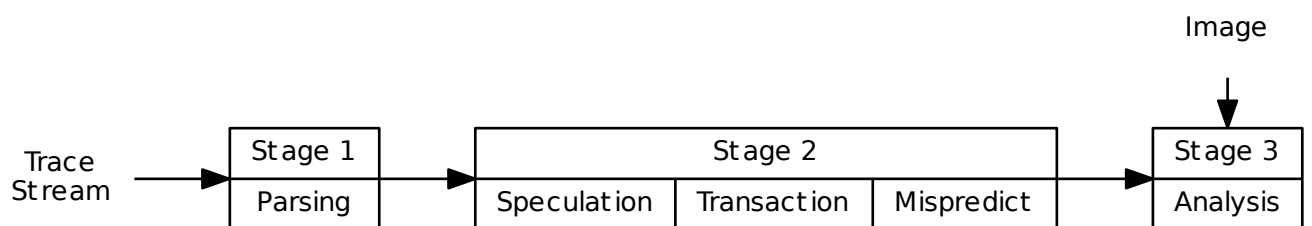
Figure D9.1: Stages of trace decompression

Rules-based writing

Rules within this section refer only to the trace analyzer and are not requirements for the trace unit. The rules in this section ensure that trace the analyzer works with all possible ETE trace units, but trace analyzers can be tailored for specific implementations.

D9.1 Introduction

D9.1.1 Using *Trace Info elements* to start trace analysis

After the trace analyzer has located an Alignment Synchronization packet and synchronized with the trace byte stream, it must search for the following elements to begin to analyze the trace byte stream:

1. A *Trace Info element*.
2. A *Context element* and a *Target Address element*.

I_{FCGKX} The trace unit might not generate a *Context element* and *Target Address element* immediately after it generates a *Trace Info element*.

I_{BWQGD} If a *Cancel element* cancels a *Trace Info element* then the trace analyzer can still use the information from the discarded *Trace Info element*, but if the *Context element* and *Target Address element* are also discarded, then the trace analyzer must wait for the trace unit to generate a new *Context element* and *Target Address element*.

D9.1.2 Encountering *Trace Info elements* after trace analysis has started

The trace unit might generate *Trace Info elements* periodically, as a result of trace protocol synchronization requests. This is useful if trace is stored in a circular buffer, because it provides multiple points where trace analysis can start.

After a trace analyzer observes the first *Trace Info element*, it can ignore subsequent *Trace Info elements* in the same trace session because the static trace programming cannot change and the speculation depth is updated by other element types during the trace session.

D9.1.3 Decompression information

To decompress a trace byte stream, the trace analyzer requires a number of values which differ between implementations. All the information required by the decompressor to analyze the trace byte stream is provided in the TRCIDR0 to TRCIDR13 registers.

Table D9.1 lists the static variables required by the decompressing stages and the fields that provide this information.

Table D9.1: Static trace unit information

Variable	ID Field	Stage
Commit Mode	TRCIDR0.COMMOPT	1
Virtual context identifier size	TRCIDR2.VMIDSIZE	1
Maximum speculation depth	TRCIDR8.MAXSPEC	1
Transactional Memory support	TRCIDR0.COMMTRANS	1 & 2
WFx Instructions	TRCIDR2.WFXMODE	3

D9.2 Stage 1 - Parsing the byte stream

The first stage of analyzing the trace is to convert from the bits of the trace byte stream to the elements that are encoded in that trace byte stream.

The ETE architecture enables a trace unit to use techniques that can reduce the trace bandwidth and trace storage requirements. Some of these techniques require the trace analyzer to retain some information between packets so that it can successfully analyze future packets.

D9.2.1 Retained state

R_{ZPQMY} The trace analyzer maintains an independent copy of the address history buffer of the last three *Target Address elements*.

I_{HDKHN} The address history buffer in the trace analyzer is required to reconstruct the *Target Address elements* from the trace byte stream.

R_{JBGCR} The trace analyzer must maintain the current speculation depth of the parsed trace byte stream.

R_{HDJJV} The trace analyzer must have the maximum speculation depth supported by the trace unit.

R_{RZNPD} The trace analyzer maintains a copy of the last *Timestamp element* value decompressed.

I_{NPCCX} The last *Timestamp element* value in the trace analyzer is required to reconstruct the full timestamp value for a Timestamp Packet.

R_{ZPNYS} The trace analyzer must maintain a copy of the context:

- Context identifier.
- Virtual context identifier.
- AArch64 or AArch32 state
- Exception level
- Security state

R_{MWFSN} The trace analyzer must maintain a copy of the cycle count threshold.

D9.2.1.1 InstructionParserState

```
// InstructionParserState
// =====================
// State of the instruction parser.


type InstructionParserState is (
    bits(64) timestamp,          // The most recently broadcast timestamp value.
    // The Address History Buffer.
    array [0..2] of AddressHistoryBufferEntry address_history_buffer,


    // Context parameters.
    bits(32) context_id,         // Most recently broadcast Context ID.
    bits(32) vmid,               // Most recently broadcast VMID.
    bits(2) exception_level,     // Most recently broadcast Exception level.
    SecurityLevel security,      // Most recently broadcast Security state.
    boolean sixty_four_bit,      // Most recently broadcast AArch state
                                 // (32 or 64?).
```

```
        // Speculation
        integer current_spec_depth,   // The current speculation depth.
        boolean T,                    // The current transactional state.


        // Trace Session static
        integer cc_threshold,         // Cycle count threshold value.


        // Static state
        integer max_spec_depth,       // The maximum speculation depth.
        boolean commit_mode,          // Commit mode.
        boolean comm_trans            // How transactions traced.
    )
```

D9.2.2 Parsing

The first stage of the decompressor is to convert from the trace byte stream to trace element stream. The trace byte stream can start at the an Alignment Sync packet boundary.

D9.2.2.1 Parse_Trace()

```
// Parse_Trace()
// ============
// Parses a trace bytestream generated by an ETE trace unit.


Parse_Trace(bits(S) stream)
    repeat
        header = ReadAndConsume(8, stream);
        LogDecompressor(PARSE, DSTATE.stream_ptr ++ " Header " ++ header);
        case header of
            when '00000000'  Parse_ExtensionPacket(header, stream);
            when '00000001'  Parse_TraceInfoPacket(header, stream);
            when '0000001x'  Parse_TimestampPacket(header, stream);
            when '00000100'  TraceOnPacket();
            when '00000110'  Parse_ExceptionPacket(header, stream);
            when '00001010'  TransactionStartPacket();
            when '00001011'  TransactionCommitPacket();
            when '0000110x'  Parse_CycleCountPackets(header, stream);
            when '0000111x'  Parse_CycleCountPackets(header, stream);
            when '0001xxxx'  Parse_CycleCountPackets(header, stream);
            when '00101101'  Parse_CommitPacket(header, stream);
            when '0010111x'  Parse_CancelPackets(header, stream);
            when '001100xx'  Parse_MispredictPacket(header, stream);
            when '001101xx'  Parse_CancelPackets(header, stream);
            when '00111xxx'  Parse_CancelPackets(header, stream);
            when '01110000'  // Ignore packet
            when '0111xxxx'  Parse_EventTracingPacket(header, stream);
            when '1000000x'  Parse_ContextPacket(header, stream);
            when '1000001x'  Parse_AddressWithContextPacket(header, stream);
            when '10000101'  Parse_AddressWithContextPacket(header, stream);
            when '10000110'  Parse_AddressWithContextPacket(header, stream);
            when '1001000x'  Parse_TargetAddressPacket(header, stream);
```

```
                  when '10010010'  Parse_TargetAddressPacket(header, stream);
                  when '100101xx'  Parse_TargetAddressPacket(header, stream);
                  when '1001100x'  Parse_TargetAddressPacket(header, stream);
                  when '1001101x'  Parse_TargetAddressPacket(header, stream);
                  when '10011101'  Parse_TargetAddressPacket(header, stream);
                  when '10011110'  Parse_TargetAddressPacket(header, stream);
                  when '1010xxxx'  Parse_QPacket(header, stream);
                  when '101100xx'  Parse_SourceAddressPacket(header,stream);
                  when '1011010x'  Parse_SourceAddressPacket(header,stream);
                  when '1011011x'  Parse_SourceAddressPacket(header,stream);
                  when '1011100x'  Parse_SourceAddressPacket(header,stream);
                  when '11xxxxxx'  Parse_AtomPackets(header, stream);
                  otherwise        ReservedEncoding();
          until EndOfStream(stream);


          return;
```

D9.2.3 Alignment Sync packet

The Alignment Sync packet is a unique sequence of bits that identifies the boundary of another packet. The unique sequence is a header byte, 0b00000000, followed by a minimum of ten payload bytes of 0b00000000 and one final payload byte of 0b10000000.

D9.2.3.1 Parse_ExtensionPacket()

```
// Parse_ExtensionPacket()
// ======================
// Parses Alignment Synchronization, Discard and Overflow packets.


Parse_ExtensionPacket(bits(8) header, bits(S) stream)
    extension = ReadAndConsume(8, stream);
    case extension of
        when '00000000'              // A-sync
            while extension == '00000000' do
                extension = ReadAndConsume(8, stream);
            if extension != '10000000' then
                ReservedEncoding();
            LogDecompressor(PARSE, "ASYNC");


        when '00000011'              // Discard
            LogDecompressor(PARSE, "DISCARD");
            DiscardPacket();


        when '00000101'              // Overflow
            LogDecompressor(PARSE, "OVERFLOW");
            OverflowPacket();


        otherwise
            ReservedEncoding();
```

```
                    return;
```

D9.2.4 Discard

R_{RVFVY}   The current speculation depth must be reset to 0.

D9.2.4.1 DiscardPacket()

```
// DiscardPacket()
// ===============
// Processes a Discard packet.


DiscardPacket()
    Emit(DiscardElement());
    if DSTATE.IA.T then
        Emit(TransactionFailureElement());


    DSTATE.IA.current_spec_depth = 0;
    DSTATE.IA.T = FALSE;


    return;
```

D9.2.5 Overflow

An Overflow instruction trace packet is output whenever a trace unit buffer overflow occurs, which means that some of the trace might be lost, and that tracing is inactive until the overflow condition clears.

D9.2.5.1 OverflowPacket()

```
// OverflowPacket()
// ================
// Processes an Overflow packet.


OverflowPacket()
    Emit(DiscardElement());
    Emit(OverflowElement());


    if DSTATE.IA.T then
        Emit(TransactionFailureElement());


    DSTATE.IA.T = FALSE;
    DSTATE.IA.current_spec_depth = 0;


    return;
```

D9.2.6 Trace Info

I_{JMXMP} A Trace Info packet indicates where the compression algorithms used by the trace unit have been set to a known architectural state. As the architectural state of the compression algorithms is known a trace analyzer can start decompression of the trace byte stream at this point.

If the trace unit exposes some trace speculation to the trace analyzer then the trace info packet indicates the trace speculation depth at this point in the trace element stream.

If the trace analyzer starts analysis where the trace speculation depth is nonzero then the analyzer should ignore speculation depth of *Commit elements*.

R_{PKZKZ} A Trace Info Packet sets all entries of the address history buffer to have an address of `0x0` and to sub_isa of IS0.

R_{LPYLR} The current_spec_depth is set to the speculation depth indicated in the trace info element.

D9.2.6.1 Parse_TraceInfoPacket()

```
// Parse_TraceInfoPacket()
// =======================
// Parses a Trace Info packet.


Parse_TraceInfoPacket(bits(8) header, bits(S) stream)
    bits(8)  INFO = Zeros();
    bits(96) SPEC = Zeros();
    bits(96) CYCT = Zeros();


    bits(8) PLCTL = ReadAndConsume(8, stream);


    //  Extract the INFO section if present
    if PLCTL<0> == '1' then
        INFO = ReadAndConsume(8, stream);


    //  Extract the SPEC section if present
    if PLCTL<2> == '1' then
        SPEC = ULEB128(stream);


    //  Extract the CYCT section if present
    if PLCTL<3> == '1' then
        CYCT = ULEB128(stream);


    TraceInfoPacket(PLCTL, INFO, SPEC, CYCT);


    return;
```

D9.2.6.2 TraceInfoPacket()

```
// TraceInfoPacket()
// =================
// Processes a Trace Info packet.
```

```
TraceInfoPacket(bits(8) PLCTL,
                bits(8) INFO,
                bits(SN) SPEC,
                bits(CN) CYCT)
    DSTATE.IA.timestamp = Zeros();
    DSTATE.IA.context_id = Zeros();
    DSTATE.IA.vmid = Zeros();
    DSTATE.IA.exception_level = EL0;
    DSTATE.IA.security = SecurityLevel_SECURE;
    DSTATE.IA.sixty_four_bit = FALSE;
    AddressHistoryBuffer.Reset();


    cc_threshold = if INFO<0> == '1' then UInt(CYCT) else 0;
    DSTATE.IA.current_spec_depth = UInt(SPEC);


    Emit(TraceInfoElement(INFO<0> == '1',     // cc_enabled
                          cc_threshold,
                          DSTATE.IA.current_spec_depth,
                          INFO<6> == '1'));


    return;
```

D9.2.7 Trace On

The Trace On packet indicates that there has been a discontinuity in the instruction trace element stream. It is output whenever a gap occurs, after the gap occurs. This means that a Trace On packet is output:

- When trace generation becomes operative, after the first A-Sync and Trace Info packets but before any packet types that indicate any *P0 elements*.

- After a trace unit buffer overflow. Again, the Trace On packet is output after the A-Sync and Trace Info packets but before any packet types that indicate any *P0 elements*.

- After gaps caused by filtering. For example, if filtering is applied to the generation of the trace element stream, so that the trace unit only generates trace for a particular program code sequence, the trace unit might spend much of its time in an inactive state, only generating trace periodically. In this case, a Trace On packet is output after each discontinuity in the trace element stream. The Trace On packet must be output before any packet types that indicate any *P0 elements*.

D9.2.7.1 TraceOnPacket()

```
// TraceOnPacket()
// ===============
// Processes a Trace On packet.


TraceOnPacket()
    Emit(TraceOnElement());
    return;
```

D9.2.8 Speculation

R_{PSHDJ} The *Commit element* must modify the current speculation depth.

D9.2.8.1 Parse_CommitPacket()

```
// Parse_CommitPacket()
// ===================
// Parses a Commit packet.


Parse_CommitPacket(bits(8) header, bits(S) stream)
    bits(32) COMMIT = ULEB128(stream);
    LogDecompressor(PARSE, "COMMIT");
    CommitPacket(COMMIT);
    return;
```

D9.2.8.2 CommitPacket()

```
// CommitPacket()
// ==============
// Processes a Commit packet.


CommitPacket(bits(N) COMMIT)
    Emit(CommitElement(UInt(COMMIT)));
    UpdateSpecDepth(-UInt(COMMIT));
    return;
```

R_{NQHYR} The *Cancel element* must modify the current speculation depth.

D9.2.8.3 Parse_CancelPackets()

```
// Parse_CancelFormatPackets()
// ===========================
// Parses all the various Cancel packets.


Parse_CancelPackets(bits(8) header, bits(S) stream)
    LogDecompressor(PARSE, "CANCEL");
    case header of
        when '0010111x'        // Cancel Format 1
            M = header<0>;
            bits(32) CANCEL = ULEB128(stream);
            CancelFormat1Packet(M, CANCEL);


        when '001101xx'        // Cancel Format 2
            A2 = header<1:0>;
            CancelFormat2Packet(A2);


        when '00111xxx'        // Cancel Format 3
            A = header<0>;
            CC = header<2:1>;
            CancelFormat3Packet(CC, A);
```

```
        return;
```

D9.2.8.4 CancelFormat1Packet()

```
// CancelFormat1Packet()
// =====================
// Processes a Cancel packet, format 1.


CancelFormat1Packet(bit M, bits(N) CANCEL)
    count = UInt(CANCEL);
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    if M == '1' then
        Emit(MispredictElement());


    return;
```

D9.2.8.5 CancelFormat2Packet()

```
// CancelFormat2Packet()
// =====================
// Processes a Cancel packet, format 2.


CancelFormat2Packet(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_E);
        when '10'
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_N);


    count = 1;
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    Emit(MispredictElement());


    return;
```

D9.2.8.6 CancelFormat3Packet()

```
// CancelFormat3Packet()
// =====================
// Processes a Cancel packet, format 3.


CancelFormat3Packet(bits(2) CC, bit A)
    if A == '1' then
```

```
        HandleAtom(Atom_E);
    count = UInt(CC) + 2;
    Emit(CancelElement(count));
    UpdateSpecDepth(-count);
    Emit(MispredictElement());

    return;
```

D9.2.9 Mispredict

D9.2.9.1 Parse_MispredictPacket()

```
// Parse_MispredictPacket()
// =======================
// Parses a Mispredict packet.

Parse_MispredictPacket(bits(8) header, bits(S) stream)
    A = header<1:0>;
    LogDecompressor(PARSE, "MISPREDICT");
    MispredictPacket(A);
    return;
```

D9.2.9.2 MispredictPacket()

```
// MispredictPacket()
// ==================
// Processes a Mispredict packet.

MispredictPacket(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_E);
        when '10'
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '11'
            HandleAtom(Atom_N);
        otherwise

    Emit(MispredictElement());
    return;
```

D9.2.10 Atom Packets

R_{RVVFW} Each *Atom element* decoded from an Atom packet must increment the current speculation depth of this stage of the decompressor

D9.2.10.1 Parse_AtomPackets()

```
// Parse_AtomPackets()
// ===================
// Parses all the various Atom packets.


Parse_AtomPackets(bits(8) header, bits(S) stream)
    LogDecompressor(PARSE, "ATOM");
    case header of
        when '1111011x'              // Atom Format 1
            bit A = header<0>;
            AtomFormat1Packet(A);


        when '110110xx'              // Atom Format 2
            bits(2) A = header<1:0>;
            AtomFormat2Packet(A);


        when '11111xxx'              // Atom Format 3
            bits(3) A = header<2:0>;
            AtomFormat3Packet(A);


        when '110111xx'              // Atom Format 4
            bits(2) A = header<1:0>;
            AtomFormat4Packet(A);


        when '11110101'              // Atom Format 5.1
            AtomFormat5_1Packet();


        when '11010101', '11010110', '11010111' // Atom Format 5.2
            bits(2) A = header<1:0>;
            AtomFormat5_2Packet(A);


        when '11xxxxxx'              // Atom Format 6
            bit A = header<5>;
            bits(5) COUNT = header<4:0>;
            AtomFormat6Packet(A, COUNT);


    return;
```

D9.2.10.2 AtomFormat1Packet()

```
// AtomFormat1Packet()
// ===================
// Processes an Atom packet, format 1.


AtomFormat1Packet(bit A)
    if A == '1' then
        HandleAtom(Atom_E);
    else
```

```
                HandleAtom(Atom_N);

        return;
```

D9.2.10.3 AtomFormat2Packet()

```
// AtomFormat2Packet()
// ===================
// Processes an Atom packet, format 2.

AtomFormat2Packet(bits(2) A)
    for I = 0 to 1
        if A<I> == '1' then
            HandleAtom(Atom_E);
        else
            HandleAtom(Atom_N);

        return;
```

D9.2.10.4 AtomFormat3Packet()

```
// AtomFormat3Packet()
// ===================
// Processes an Atom packet, format 3.

AtomFormat3Packet(bits(3) A)
    for I = 0 to 2
        if A<I> == '1' then
            HandleAtom(Atom_E);
        else
            HandleAtom(Atom_N);

        return;
```

D9.2.10.5 AtomFormat4Packet()

```
// AtomFormat4Packet()
// ===================
// Processes an Atom packet, format 4.

AtomFormat4Packet(bits(2) A)
    case A of
        when '00'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
            HandleAtom(Atom_E);
        when '01'
            HandleAtom(Atom_N);
```

```
                HandleAtom(Atom_N);
                HandleAtom(Atom_N);
                HandleAtom(Atom_N);
            when '10'
                HandleAtom(Atom_N);
                HandleAtom(Atom_E);
                HandleAtom(Atom_N);
                HandleAtom(Atom_E);
            when '11'
                HandleAtom(Atom_E);
                HandleAtom(Atom_N);
                HandleAtom(Atom_E);
                HandleAtom(Atom_N);


        return;
```

D9.2.10.6 AtomFormat5_1Packet()

```
// AtomFormat5_1Packet()
// =====================
// Processes an Atom packet, format 5.1.


AtomFormat5_1Packet()
    HandleAtom(Atom_N);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);
    HandleAtom(Atom_E);


    return;
```

D9.2.10.7 AtomFormat5_2Packet()

```
// AtomFormat5_2Packet()
// =====================
// Processes an Atom packet, format 5.2.


AtomFormat5_2Packet(bits(2) A)
    case A of
        when '01'
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
            HandleAtom(Atom_N);
        when '10'
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
```

```
        when '11'
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);
            HandleAtom(Atom_N);
            HandleAtom(Atom_E);

    return;
```

D9.2.10.8 AtomFormat6Packet()

```
// AtomFormat6Packet()
// ===================
// Processes an Atom packet, format 6.

AtomFormat6Packet(bit A, bits(5) COUNT)
    for I = 0 to UInt(COUNT) + 2
        HandleAtom(Atom_E);
    if A == '1' then
        HandleAtom(Atom_N);
    else
        HandleAtom(Atom_E);

    return;
```

D9.2.11 Q Packets

R_{RZFZW} The *Q element* must increment the current speculation depth at this stage of the decompressor.

D9.2.11.1 Parse_QPacket()

```
// Parse_QPacket()
// ===============
// Parses a Q packet.

Parse_QPacket(bits(8) header, bits(S) stream)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    bits(32) count;

    TYPE = header<3:0>;
    LogDecompressor(PARSE, "Q");
    case TYPE of
        when '0000', '0001', '0010'
            ExactMatchBytes(header, stream);
            count = ULEB128(stream);
            QPacket(TYPE, entry, count);

        when '0101'
            ShortAddressBytes(IS0, stream);
```

```
                    count = ULEB128(stream);
                    QPacket(TYPE, entry, count);


            when '0110'
                    ShortAddressBytes(IS1, stream);
                    count = ULEB128(stream);
                    QPacket(TYPE, entry, count);


            when '1010', '1011'
                    LongAddressBytes(header, stream);
                    count = ULEB128(stream);
                    QPacket(TYPE, entry, count);


            when '1100'
                    count = ULEB128(stream);
                    QPacket(TYPE, UNKNOWN_ADDRESS, count);


            when '1111'
                    QPacket(TYPE, UNKNOWN_ADDRESS, UNKNOWN_COUNT);


            otherwise
                    ReservedEncoding();


        return;
```

D9.2.11.2 QPacket()

```
// QPacket()
// =========
// Processes a Q packet.

QPacket(bits(4) TYPE, AddressHistoryBufferEntry A, bits(CN) COUNT)
    Emit(QElement(UInt(COUNT)));
    UpdateSpecDepth(1);

    // The decoding of the Address field is done by the AddressPacket function,
    // but this did not Emit the address element.
    if (TYPE != '11xx' && TYPE != '00xx') then
        Emit(TargetAddressElement(A));


    return;
```

D9.2.12 Source Address Packets

D9.2.12.1 Parse_SourceAddressPacket()

```
// Parse_SourceAddressPacket()
// ==========================
// Parses a Source Address packet.


Parse_SourceAddressPacket(bits(8) header, bits(S) stream)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);


    case header of
        when '10111000'
            data1 = ReadAndConsume(8, stream);
            data2 = ReadAndConsume(8, stream);
            data3 = ReadAndConsume(48, stream);
            entry.sub_isa = IS0;
            entry.address<63:0> = data3:data2<6:0>:data1<6:0>:'00';
            AddressHistoryBuffer.Add(entry);
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));


        when '10111001'
            data11 = ReadAndConsume(8, stream);
            data21 = ReadAndConsume(56, stream);
            a = entry;
            entry.sub_isa = IS1;
            entry.address<63:0> = data21:data11<6:0>:'0';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));


        when '10110110'
            data12 = ReadAndConsume(8, stream);
            data22 = ReadAndConsume(8, stream);
            data32 = ReadAndConsume(16, stream);
            a = entry;
            entry.sub_isa = IS0;
            entry.address<31:0> = data32:data22<6:0>:data12<6:0>:'00';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
            Emit(SourceAddressElement(entry));


        when '10110111'
            data13 = ReadAndConsume(8, stream);
            data23 = ReadAndConsume(24, stream);
            a = entry;
            entry.sub_isa = IS1;
            entry.address<31:0> = data23:data13<6:0>:'0';
            AddressHistoryBuffer.Add(entry);
            UpdateSpecDepth(1);
```

```
                        Emit(SourceAddressElement(entry));


            when '10110100'
                data14 = ReadAndConsume(8, stream);
                a = entry;
                entry.sub_isa = IS0;
                if data14<7> == '1' then
                    data24 = ReadAndConsume(8, stream);
                    entry.address<16:0> = data24<7:0>:data14<6:0>:'00';
                else
                    entry.address<8:0> = data14<6:0>:'00';


                AddressHistoryBuffer.Add(entry);
                UpdateSpecDepth(1);
                Emit(SourceAddressElement(entry));


            when '10011101'
                data15 = ReadAndConsume(8, stream);
                a = entry;
                entry.sub_isa = IS0;
                if data15<7> == '1' then
                    data25 = ReadAndConsume(8, stream);
                    entry.address<15:0> = data25<7:0>:data15<6:0>:'0';
                else
                    entry.address<7:0> = data15<6:0>:'0';


                AddressHistoryBuffer.Add(entry);
                UpdateSpecDepth(1);
                Emit(SourceAddressElement(entry));


            when '101100xx'                  // Exact match
                q = UInt(header<1:0>);
                entry = AddressHistoryBuffer.Get(q);
                AddressHistoryBuffer.Add(entry);
                UpdateSpecDepth(1);
                Emit(SourceAddressElement(entry));


        return;
```

D9.2.13 Exceptions

R_{SRPFR} The *Exception element* must increment the current speculation depth at this stage of the decompressor.

D9.2.13.1 Parse_ExceptionPacket()

```
// Parse_ExceptionPacket()
// ======================
// Parses an exception packet.
```

```
Parse_ExceptionPacket(bits(8) header, bits(S) stream)
    payload = ReadAndConsume(8, stream);
    bits(2) E;
    E<0> = payload<0>;
    E<1> = payload<6>;
    bits(5) TYPE = payload<5:1>;
    bits(8) AH;
    LogDecompressor(PARSE, "EXCEPTION");


    if E == '01' || E == '10' then
        // Treat the ADDRESS bytes as an Address packet, including
        // updating the address registers.
        AH = ReadAndConsume(8, stream);


        case AH of
            when '1001000x', '10010010'             // Exact Match.
                ExactMatchBytes(AH, stream);


            when '10010101'                         // Short Address IS0.
                ShortAddressBytes(IS0, stream);


            when '10010110'                         // Short Address IS1.
                ShortAddressBytes(IS1, stream);


            when '1001101x', '10011101', '10011110'   // Long Address.
                LongAddressBytes(AH, stream);


            when '1000001x', '10000101', '10000110'   // Long Address with
                LongAddressBytes(AH, stream);       // Context.
                ContextBytes(stream);


            when '01110000'                         // Unknown address
                UnknownAddressHistoryBuffer();

        ExceptionPacket(E, TYPE, AH);
    else
        ReservedEncoding();


    return;
```


D9.2.13.2 ExceptionPacket()

```
// ExceptionPacket()
// =================
// Processes an Exception packet.


ExceptionPacket(bits(2) E, bits(5) TYPE, bits(8) AH)
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    case E of
```

```
                    when '01'
                        if TYPE == '11000' then
                            Emit(TransactionFailureElement());
                        else
                            Emit(ExceptionElement(UInt(TYPE), entry.address));
                    when '10'
                        // The new context and address must now be Emitted.
                        if AH<7:4> == '1000' then   // Long Address with Context
                            Emit(ContextElement(DSTATE.IA.context_id,
                                                DSTATE.IA.vmid,
                                                DSTATE.IA.exception_level,
                                                DSTATE.IA.security,
                                                DSTATE.IA.sixty_four_bit));
                        Emit(TargetAddressElement(entry));

                        if TYPE == '00000' && DSTATE.IA.T then
                            Emit(TransactionFailureElement());
                            Emit(ExceptionElement(UInt(TYPE), entry.address));
                        elsif TYPE == '11000' then
                            Emit(TransactionFailureElement());
                        else
                            Emit(ExceptionElement(UInt(TYPE), entry.address));

                    otherwise
                        ReservedEncoding();

                UpdateSpecDepth(1);

                return;
```

D9.2.14 Address and context

D9.2.14.1 Parse_TargetAddressPacket()

```
// Parse_TargetAddressPacket()
// ===========================
// Parses a Target Address packet.

Parse_TargetAddressPacket(bits(8) header, bits(S) stream)
    case header of
        when '1001101x', '10011101', '10011110' // Long Address
            LogDecompressor(PARSE, "LONG_ADDRESS");
            LongAddressBytes(header, stream);

        when '10010101'                         // Short Address IS0
            LogDecompressor(PARSE, "SHORT_ADDRESS");
            ShortAddressBytes(IS0, stream);

        when '10010110'                         // Short Address IS1
```

```
                LogDecompressor(PARSE, "SHORT_ADDRESS");
                ShortAddressBytes(IS1, stream);


        when '1001000x', '10010010'              // Exact match
                LogDecompressor(PARSE, "EXACT_MATCH");
                ExactMatchBytes(header, stream);

    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);
    Emit(TargetAddressElement(entry));
    return;
```

D9.2.14.2 LongAddressBytes()

```
// LongAddressBytes()
// ==================
// Reads and parses the long address form used in some packets.


LongAddressBytes(bits(8) header, bits(S) stream)
    case header<2:0> of
        when '010'      // 32-bit IS0
                data32_is0 = ReadAndConsume(16, stream);
                A8_2 = data32_is0<6:0>;  SBZ(data32_is0<7>);
                A15_9 = data32_is0<14:8>; SBZ(data32_is0<15>);
                bits(16) A31_16;
                A31_16 = POD(16, stream);
                LongAddressPacket(header, A31_16:A15_9:A8_2);


        when '011'      // 32-bit IS1
                data32_is1 = ReadAndConsume(8, stream);
                A7_1 = data32_is1<6:0>;  SBZ(data32_is1<7>);
                A31_8 = POD(24, stream);
                LongAddressPacket(header, A31_8:A7_1);


        when '101'      // 64-bit IS0
                data64_is0 = ReadAndConsume(16, stream);
                A8_2 = data64_is0<6:0>;
                SBZ(data64_is0<7>);
                A15_9 = data64_is0<14:8>;
                SBZ(data64_is0<15>);
                A63_16 = POD(48, stream);
                LongAddressPacket(header, A63_16:A15_9:A8_2);


        when '110'      // 64-bit IS1
                data64_is1 = ReadAndConsume(8, stream);
                A7_1 = data64_is1<6:0>;  SBZ(data64_is1<7>);
                A63_8 = POD(56, stream);
                LongAddressPacket(header, A63_8:A7_1);


        return;
```

D9.2.14.3 LongAddressPacket()

```
// LongAddressPacket()
// ==================
// Parses the long form address used in some packets.


LongAddressPacket(bits(8) header, bits(AN) A)

    a = AddressHistoryBuffer.Get(0);


    // Called from a variety of packet types, so only look at bits <2:0> of
    // the header.
    case header<2:0> of
        when '010'            // 32-bit address, IS0
            assert(AN == 30);
            a.sub_isa = IS0;
            // address<63:32> unchanged
            a.address<31:2> = A<29:0>;
            a.address<1:0> = '00';


        when '011'            // 32-bit address, IS1
            assert(AN == 31);
            a.sub_isa = IS1;
            // address<63:32> unchanged
            a.address<31:1> = A<30:0>;
            a.address<0> = '0';


        when '101'            // 64-bit address, IS0
            assert(AN == 62);
            a.sub_isa = IS0;
            a.address<63:2> = A<61:0>;
            a.address<1:0> = '00';


        when '110'            // 64-bit address, IS1
            assert(AN == 63);
            a.sub_isa = IS1;
            a.address<63:1> = A<62:0>;
            a.address<0> = '0';


    UpdateAddressHistoryBuffer(a.address, a.sub_isa);
    return;
```

D9.2.14.4 ShortAddressBytes()

```
// ShortAddressBytes()
// ==================
// Reads and parses the short form address used in some packets.
```

```
ShortAddressBytes(SubISA sub_isa, bits(S) stream)
    bits(15) data;

    if sub_isa == IS0 then
        data = BitReplacement(stream,
                            AddressHistoryBuffer.Get(0).address<16:2>);
    else
        data = BitReplacement(stream,
                            AddressHistoryBuffer.Get(0).address<15:1>);


    ShortAddressPacket(sub_isa, data);
    return;
```

D9.2.14.5 ShortAddressPacket()

```
// ShortAddressPacket()
// ===================
// Parses the short form address used in some packets.


ShortAddressPacket(SubISA sub_isa, bits(AN) A)
    a = AddressHistoryBuffer.Get(0);

    assert (AN == 7 || AN == 15);


    case sub_isa of
        when IS0                              // IS0
            a.sub_isa = IS0;
            // address<63:AN+2> unchanged
            a.address<AN+1:0> = A:'00';

        when IS1                              // IS1
            a.sub_isa = IS1;
            // address<63:AN+1> unchanged
            a.address<AN:0> = A:'0';


    UpdateAddressHistoryBuffer(a.address, a.sub_isa);
    return;
```

D9.2.14.6 ExactMatchBytes()

```
// ExactMatchBytes()
// ================
// Reads and parses an exact address match used in some packets.


ExactMatchBytes(bits(8) header, bits(S) stream)
    QE = header<1:0>;
    ExactMatchPacket(QE);
    return;
```

D9.2.14.7 ExactMatchPacket()

```
// ExactMatchPacket()
// ==================
// Parses an exact address match.


ExactMatchPacket(bits(2) QE)
    q = UInt(QE);
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(q);
    AddressHistoryBuffer.Add(entry);
    return;
```

R_{TKBWR}   The *Context element* value is created by combining the value encoded in a Context Packet and the last *Context element* value.

D9.2.14.8 Parse_AddressWithContextPacket()

```
// Parse_AddressWithContextPacket()
// ================================
// Parses a Target Address with Context packet.


Parse_AddressWithContextPacket(bits(8) header, bits(S) stream)
    LongAddressBytes(header, stream);
    ContextBytes(stream);
    LogDecompressor(PARSE, "ADDR_W_CONTEXT");
    AddressHistoryBufferEntry entry = AddressHistoryBuffer.Get(0);


    Emit(ContextElement(DSTATE.IA.context_id,
                        DSTATE.IA.vmid,
                        DSTATE.IA.exception_level,
                        DSTATE.IA.security,
                        DSTATE.IA.sixty_four_bit));
    Emit(TargetAddressElement(entry));
    return;
```

D9.2.14.9 ContextBytes()

```
// ContextBytes()
// ==============
// Generates a Context packet from the stream.


ContextBytes(bits(S) stream)
    payload = ReadAndConsume(8, stream);
    EL = payload<1:0>;
    SBZ(payload<3:2>);
    SF = payload<4>;
    NS = payload<5>;
    V = payload<6>;
    C = payload<7>;
    case C:V of
        when '00'
```

```
                    ContextPacket('1', C, V, NS, SF, EL,
                                  DSTATE.IA.vmid,
                                  DSTATE.IA.context_id);
            when '01'
                bits(32) VMID;
                VMID = POD(32, stream);
                ContextPacket('1', C, V, NS, SF, EL,
                                  VMID,
                                  DSTATE.IA.context_id);
            when '10'
                context_id = POD(32, stream);
                ContextPacket('1', C, V, NS, SF, EL,
                                  DSTATE.IA.vmid,
                                  context_id);
            when '11'
                bits(32) VMID;
                VMID = POD(32, stream);
                context_id = POD(32, stream);
                ContextPacket('1', C, V, NS, SF, EL,
                                  VMID,
                                  context_id);


    return;


// Parse_ContextPacket()
// =====================
// Parses a Context packet.


Parse_ContextPacket(bits(8) header, bits(S) stream)
    LogDecompressor(PARSE, "CONTEXT");
    P = header<0>;
    if P == '0' then
        ContextPacket(P, '0', '0', '0', '0', '00',
                      DSTATE.IA.vmid,
                      DSTATE.IA.context_id);
    else
        ContextBytes(stream);


    Emit(ContextElement(DSTATE.IA.context_id,
                        DSTATE.IA.vmid,
                        DSTATE.IA.exception_level,
                        DSTATE.IA.security,
                        DSTATE.IA.sixty_four_bit));
    return;
```

D9.2.14.10 ContextPacket()

```
// ContextPacket()
// ===============
// Processes a Context packet.
```

```
ContextPacket(bit P,
              bit C,
              bit V,
              bit NS,
              bit SF,
              bits(2) EL,
              bits(32) VMID,
              bits(32) context_id)
    if P == '1' then
        if C == '1' then
            DSTATE.IA.context_id = context_id;
        if V == '1' then
            DSTATE.IA.vmid = VMID;
        LogDecompressor(PACKET,
                        "set context_id " ++ DSTATE.IA.context_id ++
                        ", vmid " ++ DSTATE.IA.vmid);
        DSTATE.IA.exception_level<1:0> = EL<1:0>;
        if NS == '1' then
            DSTATE.IA.security = SecurityLevel_NONSECURE;
        else
            DSTATE.IA.security = SecurityLevel_SECURE;
        DSTATE.IA.sixty_four_bit = (SF == '1');


    return;
```

D9.2.15 Transactions

D9.2.15.1 TransactionStartPacket()

```
// Transaction Start Packet()
// ==========================
// Processes a Transaction Start packet.


TransactionStartPacket()
    Emit(TransactionStartElement());
    DSTATE.IA.T = TRUE;
    if DSTATE.IA.comm_trans then
        UpdateSpecDepth(1);


    return;
```

D9.2.15.2 TransactionCommitPacket()

```
// TransactionCommitPacket()
// =========================
// Processes a Transacition Commit packet.


TransactionCommitPacket()
    Emit(TransactionCommitElement());
    DSTATE.IA.T = FALSE;
```

```
                return;
```

D9.2.16 Timestamps

R_{KJNMB}   The *Timestamp element* value is created by combining the value encoded in a Timestamp Packet and the last *Timestamp element* value.

D9.2.16.1 Parse_TimestampPacket()

```
// Parse_TimestampPacket()
// =======================
// Parses a Timestamp packet.


Parse_TimestampPacket(bits(8) header, bits(S) stream)
    bit N;
    bits(64) TS = BitReplacement(stream, DSTATE.IA.timestamp);
    bits(20) COUNT = Zeros();
    N = header<0>;
    LogDecompressor(PARSE, "TIMESTAMP");
    if N == '1' then
        COUNT = ULEB128(stream);
        TimestampPacket(N, TS, COUNT);
    else
        TimestampPacket(N, TS, COUNT);


    return;
```

D9.2.16.2 Parse_CycleCountPackets()

```
// Parse_CycleCountPackets()
// =========================
// Parses all the various Cycle Count packets.


Parse_CycleCountPackets(bits(8) header, bits(S) stream)
    LogDecompressor(PARSE, "CYCLE_COUNT");
    case header of
        when '0000111x'                // Cycle Count format 1
            bit U = header<0>;
            bits(32) COMMIT = Zeros();
            bits(20) COUNT = Zeros();
            if TRCIDR0.COMMOPT == '0' then
                COMMIT = ULEB128(stream);
            if U == '0' then
                COUNT = ULEB128(stream);
            CycleCountFormat1Packet(U, COMMIT, COUNT);


        when '0000110x'                // Cycle Count format 2
            bits(8) payload = POD(8, stream);
            bits(4) BBBB = payload<3:0>;
            if TRCIDR0.COMMOPT == '0' then
```

```
                    bit F = header<0>;
                    bits(4) AAAA = payload<7:4>;
                    CycleCountFormat2Packet(F, AAAA, BBBB);
                else
                    CycleCountFormat2Packet('1', '1111', BBBB);


            when '0001xxxx'                 // Cycle Count format 3
                bits(2) BB = header<1:0>;
                if TRCIDR0.COMMOPT == '0' then
                    bits(2) AA = header<3:2>;
                    CycleCountFormat3Packet(AA, BB);
                else
                    CycleCountFormat3Packet('00', BB);


        return;
```

D9.2.16.3 TimestampPacket()

```
// TimestampPacket()
// =================
// Processes a Timestamp packet.


TimestampPacket(bit N, bits(64) TS, bits(CN) COUNT)
    DSTATE.IA.timestamp = TS;
    if N == '1' then
        Emit(TimestampElement(UInt(DSTATE.IA.timestamp), UInt(COUNT)));
    else
        Emit(TimestampElement(UInt(DSTATE.IA.timestamp), integer UNKNOWN));


    return;
```

D9.2.16.4 CycleCountFormat1Packet()

```
// CycleCountFormat1Packet()
// =========================
// Processes a Cycle Count packet, format 1.


CycleCountFormat1Packet(bit U, bits(N) COMMIT, bits(20) COUNT)
    if UInt(COMMIT) > 0 then
        Emit(CommitElement(UInt(COMMIT)));
    UpdateSpecDepth(-UInt(COMMIT));
    if U == '1' then
        Emit(CycleCountElement(integer UNKNOWN));
    else
        Emit(CycleCountElement(UInt(COUNT) + DSTATE.IA.cc_threshold));


    return;
```

D9.2.16.5 CycleCountFormat2Packet()

```
// CycleCountFormat2Packet()
// =========================
// Processes a Cycle Count packet, format 2.


CycleCountFormat2Packet(bit F, bits(4) AAAA, bits(4) BBBB)
    if F == '1' then
        commit_count = DSTATE.IA.max_spec_depth + UInt(AAAA) - 15;
    else
        commit_count = UInt(AAAA) + 1;


    if commit_count > 0 then
        Emit(CommitElement(commit_count));
        UpdateSpecDepth(-commit_count);
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BBBB)));


    return;
```

D9.2.16.6 CycleCountFormat3Packet()

```
// CycleCountFormat3Packet()
// =========================
// Processes a Cycle Count packet, format 3.


CycleCountFormat3Packet(bits(2) AA, bits(2) BB)
    if !DSTATE.IA.commit_mode then
        Emit(CommitElement(UInt(AA) + 1));
        UpdateSpecDepth(-(UInt(AA) + 1));
    Emit(CycleCountElement(DSTATE.IA.cc_threshold + UInt(BB)));


    return;
```

D9.2.17 Event Tracing

D9.2.17.1 Parse_EventTracingPacket()

```
// Parse_EventTracingPacket()
// ==========================
// Parses an Event packet.


Parse_EventTracingPacket(bits(8) header, bits(S) stream)
    LogDecompressor(PARSE, "EVENT_TRACE");
    EventTracingPacket(header<3:0>);
    return;
```

D9.2.17.2 EventTracingPacket()

```
// EventTracingPacket()
// ====================
// Processes an Event packet.


EventTracingPacket(bits(4) EVENT)
    for I = 0 to 3
        if EVENT<I> == '1' then
            Emit(EventElement(I));


    return;
```

D9.2.18 Functions

D9.2.18.1 ReadAndConsume()

```
// ReadAndConsume()
// ================
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.
bits(N) ReadAndConsume(integer N, bits(S) stream);
```

D9.2.18.2 HandleAtom()

```
// HandleAtom()
// ============
// Logs and emits an atom, and updates the speculation depth.


HandleAtom(Atom t)
    LogDecompressor(PACKET, if t == Atom_E then "E" else "N");
    Emit(AtomElement(t));
    UpdateSpecDepth(1);
    return;
```

D9.2.18.3 UpdateSpecDepth()

```
// UpdateSpecDepth()
// =================
// Update the speculation depth by a number of elements.


UpdateSpecDepth(integer count)
    DSTATE.IA.current_spec_depth = DSTATE.IA.current_spec_depth + count;
    if DSTATE.IA.current_spec_depth > DSTATE.IA.max_spec_depth then
        commit_number = DSTATE.IA.current_spec_depth - DSTATE.IA.max_spec_depth;
        Emit(CommitElement(commit_number));
        DSTATE.IA.current_spec_depth = DSTATE.IA.max_spec_depth;
```

```
        return;
```

D9.2.18.4 UpdateAddressHistoryBuffer()

```
// UpdateAddressHistoryBuffer()
// ===========================
// Adds the given address and sub_isa to the AHB.

UpdateAddressHistoryBuffer(bits(64) address, SubISA sub_isa)
    AddressHistoryBuffer.Add(address, sub_isa);
    return;
```

D9.2.18.5 UnknownAddressHistoryBuffer()

```
// UnknownAddressHistoryBuffer()
// ============================
// Adds an unknown address and sub_isa to the AHB.

UnknownAddressHistoryBuffer()
    AddressHistoryBuffer.Add(Zeros(), IS0);
    return;
```

D9.3 Stage 2 - Speculation Resolution

The resolution stage operates on the Elements in turn. The Elements are buffered until their resolution is determined.

D9.3.1 Emit()

```
// Emit()
// ======
Emit(Element e)
    e.committed = FALSE;
    case e.kind of
        when ELEM_TRACE_INFO     ProcessTraceInfo(e);


        // Speculation Support
        when ELEM_CANCEL            ProcessCancel(e);
        when ELEM_COMMIT            ProcessCommit(e);
        when ELEM_DISCARD           ProcessDiscard(e);


        // Transactional Support
        when ELEM_TRANS_START       ProcessTransactionStart(e);
        when ELEM_TRANS_COMMIT      ProcessTransactionCommit(e);
        when ELEM_TRANS_FAILURE     ProcessTransactionFailure(e);


        // Others
        otherwise           Stack(e);
```

D9.3.2 Trace Info element

I_{RJDHD} The *Trace Info element* can be used as a point to start decompression of the trace element stream. When the *Trace Info element* is generated there might still be some speculative *P0 elements*. The number of speculative *P0 elements* is indicated by the current speculation depth member of the *Trace Info element*.

R_{JKLZY} If the analysis of the trace starts with a *Trace Info element* with a non-zero current speculation depth the decompressor must ignore the *Commit element* or *Cancel elements* for these *P0 elements* as they will not have been observed by the decompressor.

D9.3.2.1 ProcessTraceInfo()

```
// ProcessTraceInfo()
// ==================
// Processes a Trace Info element, resetting the analyzer to a known state.


ProcessTraceInfo(Element e)
    if ResolutionQueue.Uninitialized() then
        ResolutionQueue.Initialize(e.payload.current_spec_depth);

        if e.payload.in_transaction then
            TransactionQueue.StartTransaction();
```

```
        Stack(e);
        return;
```

D9.3.3 Commit element

R_{WGSCJ} The *Commit element* marks a number of *P0 elements* as resolved, and if at the head of the queue pass these elements onto the next stage of the decompressor.

D9.3.3.1 ProcessCommit()

```
// ProcessCommit()
// ==============
// Processes a Commit element, committing the given number of speculative
// elements.


ProcessCommit(Element e)
    integer I = 0;
    LogDecompressor(SPEC, "committing " ++ e.payload.count ++ " elements");


    repeat
        if !ResolutionQueue.Aligned() then
            I = I + 1;
            ResolutionQueue.Align();
        else
            case ResolutionQueue.Front().kind of


                when ELEM_EXCEPTION, ELEM_ATOM, ELEM_Q, ELEM_SOURCE_ADDRESS
                    if !ResolutionQueue.Front().committed then
                        I = I + 1;


                    ProcessTransaction(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();


                when ELEM_TRANS_START
                    if (TRCIDR0.COMMTRANS == '0' &&
                        !ResolutionQueue.Front().committed) then
                        I = I + 1;
                    ProcessTransaction(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();


                otherwise
                    ProcessTransaction(ResolutionQueue.Front());
                    ResolutionQueue.PopFront();


        LogDecompressor(SPEC, "new spec depth " ++ ResolutionQueue.Length());
    until I == e.payload.count;


    return;
```

D9.3.4 Cancel element

I_{NYYFS} For example, if a *Cancel element* indicates that the three most recent *P0 elements* are canceled, then the trace analyzer must discard:

- The *Cancel element*.
- All elements back to, and including, the third most recent *P0 element*.
- Any *Trace On elements* encountered in that section of the element stream.

R_{SPBCG} When discarding *P0 elements* that have been canceled, a trace analyzer must also discard many other element types that occur in the element stream between the *Cancel element* and the oldest *P0 element* that the *Cancel element* cancels. Table D9.2 shows which elements must be discarded.

Table D9.2: Cancel Element Operation

Element	Behavior on cancelation
Atom	Remove
Commit	**Illegal**
Context	Remove
Cycle Count	Process
Discard	**Illegal**
Exception	Remove
Event	Keep
Mispredict	Remove
Overflow	**Illegal**
Target Address	Remove
Source Address	Remove
Timestamp	Process
Trace Info	Keep
Trace On	Remove
Transaction Start	Remove
Transaction Failure	Remove

R_{CBYHC} When a *Cancel element* occurs, a trace analyzer must not discard *Cycle Count elements*.

R_{GZWHY} When a *Cancel element* occurs, a trace analyzer must not discard ETEEvents.

R_{ZLVXG} When a *Cancel element* occurs, a trace analyzer must discard *Mispredict elements*.

D9.3.4.1 ProcessCancel()

```
// ProcessCancel()
// ===============
// Processes a Cancel element, canceling the given number of speculative
// elements.
```

```
ProcessCancel(Element e)
    integer I = 0;
    LogDecompressor(SPEC, "canceling " ++ e.payload.count ++ " elements");


    repeat
        if !ResolutionQueue.Aligned() then
            I = I + 1;
            ResolutionQueue.Align();
        else
            case ResolutionQueue.Back().kind of

                when ELEM_ATOM, ELEM_EXCEPTION, ELEM_Q, ELEM_SOURCE_ADDRESS
                    if !ResolutionQueue.Back().committed then
                        I = I + 1;
                when ELEM_TRANS_START
                    if (TRCIDR0.COMMTRANS == '0' &&
                        !ResolutionQueue.Back().committed) then
                        I = I + 1;
                        TransactionQueue.EndTransaction();
                when ELEM_CYCLE_COUNT, ELEM_EVENT, ELEM_TRACE_INFO
                    AnalyzeElement(ResolutionQueue.Back());


                when ELEM_TIMESTAMP
                    AnalyzeElement(ResolutionQueue.Back());


            ResolutionQueue.PopBack();
    until I == e.payload.count;


    return;
```

D9.3.5 Discard element

I_{LYPKS} A *Discard element* indicates that tracing has become inactive while uncommitted *P0 elements* remain.

R_{YHRPG} The trace analyzer must cancel all speculative *P0 elements*.

D9.3.5.1 ProcessDiscard()

```
// ProcessDiscard()
// ================
// Processes a Discard element, discarding all speculative elements.


ProcessDiscard(Element e)
    for i = 0 to ResolutionQueue.Length()
        print(i);
        PrintElement(DSTATE.resolution_queue.queue[i]);
        print("
n");
```

```
            while !ResolutionQueue.Aligned() do
                ResolutionQueue.Align();


            while ResolutionQueue.Length() > 0 do
                case ResolutionQueue.Back().kind of

                    when ELEM_EVENT, ELEM_TRACE_INFO, ELEM_TIMESTAMP
                        AnalyzeElement(ResolutionQueue.Back());
                        ResolutionQueue.PopBack();


                    otherwise
                        ResolutionQueue.PopBack();


            TransactionQueue.EndTransaction();
            return;
```

D9.3.6 Stack

The Elements processed by this stage of the decompressor must be stored temporarily until the speculation has
been resolved.

D9.3.6.1   Stack()

```
// Stack()
// =======
// Pushes an element onto the resolution queue.
// TODO: Move to ResolutionQueue.asl and rename to Enqueue


Stack(Element e)
    LogElem(SPEC, e,
            "stacked element, new current depth " ++
            ResolutionQueue.Length());
    ResolutionQueue.Push(e);
    return;
```

D9.4 Stage 2 - Transaction Resolution

D9.4.1 ProcessTransaction()

```
// ProcessTransaction()
// ===================
// Push the element into the transaction queue if we are in a transaction,
// else analyze it immediately.

ProcessTransaction(Element e)
    if TransactionQueue.InTransaction() then
        TransactionQueue.Push(e);
    else
        AnalyzeElement(e);
```

D9.4.2 Transaction Start element

D9.4.2.1 ProcessTransactionStart()

```
// ProcessTransactionStart()
// ========================
// Processes a Transaction Start element, marking we are now in a transaction.

ProcessTransactionStart(Element e)
    TransactionQueue.StartTransaction();
    Stack(e);
    return;
```

D9.4.3 Transaction Commit element

D9.4.3.1 ProcessTransactionCommit()

```
// ProcessTransactionCommit()
// =========================
// Processes a Transaction Commit element, committing all elements in the
// transaction queue and ending the current transaction.

ProcessTransactionCommit(Element e)
    while TransactionQueue.Length() > 0 do
        AnalyzeElement(TransactionQueue.Front());
        TransactionQueue.FrontPop();
    TransactionQueue.EndTransaction();
```

D9.4.4 Transaction Failure element

R_{XZBFV} When a *Transaction Failure element* occurs, the trace analyzer must process the *Cycle Count elements* if it is maintaining a cumulative cycle count. Otherwise it must discard the *Cycle Count elements* that are associated with *P0 elements* within the transaction.

R_{KPKFL} When a *Transaction Failure element* occurs, a trace analyzer must not discard the ETEEvents.

D9.4.4.1   ProcessTransactionFailure()

```
// ProcessTransactionFailure()
// ===========================
// Processes a Transaction Failure element, discarding all elements in
// the transaction queue and ending the transaction.


ProcessTransactionFailure(Element e)
    TransactionQueue.EndTransaction();
    return;
```

D9.5 Stage 3 - Analysis

D9.5.1 AnalyzeElement()

```
// AnalyzeElement()
// ================
// Analyzes any element.


AnalyzeElement(Element e)
    case e.kind of
        when ELEM_TARGET_ADDRESS AnalyzeTargetAddress(e);
        when ELEM_CONTEXT        AnalyzeContext(e);
        when ELEM_MISPREDICT     AnalyzeMispredict(e);
        when ELEM_TRACE_ON       AnalyzeTraceOn(e);
        when ELEM_ATOM           AnalyzeAtom(e);
        when ELEM_EXCEPTION      AnalyzeException(e);
        when ELEM_Q              AnalyzeQ(e);
        when ELEM_CANCEL         ERROR("cancel element reached analysis stage");
        when ELEM_COMMIT         ERROR("commit element reached analysis stage");
        when ELEM_DISCARD        AnalyzeDiscard(e);
        when ELEM_OVERFLOW       AnalyzeOverflow(e);
        when ELEM_EVENT          AnalyzeEvent(e);
        when ELEM_TRACE_INFO     AnalyzeTraceInfo(e);
        when ELEM_TIMESTAMP      AnalyzeTimestamp(e);
        when ELEM_CYCLE_COUNT    AnalyzeCycleCount(e);
        when ELEM_SOURCE_ADDRESS AnalyzeSourceAddress(e);
        otherwise
            ERROR("Unrecognised element kind in analysis stage");


    return;
```

D9.5.2 Retained state

R_{GWFMZ} The trace analyzer must maintain a copy of the context:

- Context identifier.
- Virtual context identifier.
- AArch64 or AArch32 state
- Exception level
- Security state

D9.5.2.1 ReconstructState

```
// ReconstructState
// ================
// Temporary storage of reconstructor state, can change after resolution.


type ReconstructState is (
    bits(64) address,           // Current address
    bits(32) context_id,        // Current context ID
    bits(32) vmid,              // Current VMID
```

```
        bits(2) exception_level,    // Current Exception level
        SecurityLevel security,     // Current Security state
        boolean sixty_four_bit,     // Are we in AArch64?
        SubISA sub_isa              // Current sub_isa
)
```

D9.5.3 Operation of the return stack

I_{SZYZF} The trace analyzer maintains an independent copy of the return stack which is used to determine when *Target Address elements* have been removed and then infer the target of indirect *P0 instructions*.

I_{LGKYD} The trace analyzer return stack only operates after a certain point in the tracing flow, that is after the trace analyzer has decoded the trace packets and after all the elements that indicate speculative execution, except for *Mispredict elements*, have been removed from the trace element stream.

R_{HYMQH} Whenever a Branch with Link instruction is initially traced with an E *Atom element*, the link return address and sub_isa are pushed onto the trace analyzer return stack. This means that the trace unit return stack grows with each new entry, until its maximum depth is reached and the oldest entries start being discarded.

R_{MGHSB} Whenever an indirect *P0 instruction* is traced with a final E *Atom element*, and no *Target Address element* is traced before the next *P0 element*, the top entry of the trace analyzer return stack is removed and the value of that entry is the target of the indirect *P0 instruction*.

R_{XSVZC} A trace analyzer is not required to be aware of the depth of the trace unit return stack, and implements a return stack depth of 15 entries.

I_{GBVND} A trace analyzer return stack push always occurs whenever a Branch with Link instruction is traced with an E *Atom element*, even if the status of the E *Atom element* later changes to be an N *Atom element* as a result of a subsequent *Mispredict element*.

For example, the following sequence might occur:

1. The *Processing Element* (PE) speculatively executes a Branch with Link instruction that the trace unit traces with an E *Atom element*. The trace unit pushes the target address of the Branch with Link instruction onto the trace unit return stack.

2. The trace analyzer receives the E *Atom element* and pushes the target address of the Branch with Link instruction onto the trace analyzer return stack.

3. The PE then cancels the speculative execution. The trace unit generates a *Mispredict element*.

4. The trace analyzer receives the *Mispredict element* and changes the status of the E *Atom element* so that it becomes an N *Atom element*. The trace analyzer then knows which direction the program flow has taken, and also knows that the target address stored at the top of the trace analyzer return stack is mispredicted.

Note

Whenever the trace unit generates a *Mispredict element* to correct a Branch with Link instruction to an N *Atom element*, the mispredicted address remains in the return stack because there is no reason to remove it. There are no adverse consequences of leaving such a mispredicted address in the stack.

R_{GMTBD} If more than one *Mispredict element* is output corresponding to a particular *Atom element*, the status of the *Atom element* alternates between E and N until it settles in its final E or N state. If the final state of the *Atom element* is E, then when the PE executes an indirect *P0 instruction* and the trace unit compares the target address with the top entry in its return stack, an address match might occur. An address match can only occur if the final status of the *Atom element* is E.

I_{MPGDD} The trace analyzer never needs to discard the entries in its copy of the return stack. If the trace unit discards the entries in its return stack then the entries in the trace analyzer return stack remain. As more entries are pushed on to the return stack, the old entries are discarded when they are pushed off the end of the stack.

I_{WJLDR} The trace analyzer does not need to prevent the return stack from being modified while in a branch broadcasting region. The fact that the trace unit discards the entries in its return stack when entering the branch broadcasting region ensures that the return stack in the trace unit and the return stack in the trace analyzer remain synchronized.

D9.5.3.1 UpdateReturnStack()

```
// UpdateReturnStack()
// ==================


// Push the given instruction to the return stack if necessary.


UpdateReturnStack(DecodedInst inst)
    if inst.is_link then
        LogDecompressor(ANALYZE,
                        "pushing to return stack inst " ++
                        DSTATE.current_analyzer_state.address ++
                        ":  " ++ inst.instruction);


        nxt_state = DSTATE.current_analyzer_state;
        nxt_state.address = nxt_state.address + inst.size;
        if !nxt_state.sixty_four_bit then
            nxt_state.address<63:32> = Zeros();


        ReturnStack.Push(nxt_state.address, nxt_state.sub_isa);
        LogReturnStack();


    return;
```

D9.5.4 Atom element

R_{SKGMZ} An *Atom element* implies that one or more instructions have been traced, up to and including the next *P0 instruction*.

I_{VHHGW} A trace analyzer must analyze each instruction in the program image from the current address until it observes a *P0 instruction*. This indicates that the PE has executed each instruction between the current address and the *P0 instruction*.

D9.5.4.1 AnalyzeAtom()

```
// AnalyzeAtom()
// =============
// Analyzes an atom element.


AnalyzeAtom(Element e)
    if e.payload.atom_type == Atom_E then
        LogElem(ANALYZE, e, "ATOM E");
        DSTATE.most_recent_branch_was_taken = TRUE;
    else
```

```
                LogElem(ANALYZE, e, "ATOM N");
                DSTATE.most_recent_branch_was_taken = FALSE;


        CheckForReturnStackMatch();


        if DSTATE.sync_state == ADDRESS_STATE then
            DSTATE.sync_state = NOT_SYNC_STATE;
        if DSTATE.sync_state != FULL_SYNC_STATE then
            // If we are unsure of context or address then we cannot meaningfully
            // analyze the atom.
            return;


        boolean cur_inst_is_branch = FALSE;


        // Continue logging instructions until we hit a P0 instruction.
        while !cur_inst_is_branch do
            if !ProgramImage.DecodeAvailable() then
                DSTATE.sync_state = CONTEXT_STATE;
                LogDecompressor(FINAL_OUTPUT,
                                "unprocessed_execution " ++
                                "due to no program image available");
                if DSTATE.return_stack_clear_pending then
                    ReturnStack.Reset();
                return;


            decoded_inst = ProgramImage.DecodeNextInst();


            case decoded_inst.branchtype of
                when InstType_BRANCH_DIR, InstType_BRANCH_INDIR
                    ProcessBranchInstruction(decoded_inst,
                                             DSTATE.most_recent_branch_was_taken);
                    cur_inst_is_branch = TRUE;
                    UpdateReturnStack(decoded_inst);
                when InstType_WFX, InstType_ISB
                    ProcessBranchInstruction(decoded_inst,
                                             DSTATE.most_recent_branch_was_taken);
                    cur_inst_is_branch = TRUE;
                when InstType_OTHER
                    ReconstructState nxt_state = DSTATE.current_analyzer_state;
                    nxt_state.address = nxt_state.address + decoded_inst.size;
                    if !nxt_state.sixty_four_bit then
                        // mask off the left-most bits
                        nxt_state.address<63:32> = Replicate('0', 32);
                    DSTATE.next_analyzer_state = nxt_state;


            OutputInstruction(decoded_inst);
            DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;


        if DSTATE.return_stack_clear_pending then
            ReturnStack.Reset();
```

```
                return;
```

D9.5.5 Context element

D9.5.5.1 AnalyzeContext()

```
// AnalyzeContext()
// ===============
// Analyzes a context element.


AnalyzeContext(Element e)
    DSTATE.current_analyzer_state.context_id = e.payload.context_id;
    DSTATE.current_analyzer_state.vmid = e.payload.vmid;
    DSTATE.current_analyzer_state.security = e.payload.security;
    DSTATE.current_analyzer_state.exception_level = e.payload.exception_level;
    DSTATE.current_analyzer_state.sixty_four_bit = e.payload.sixty_four_bit;


    case DSTATE.sync_state of
        when NOT_SYNC_STATE
            DSTATE.sync_state = CONTEXT_STATE;
        when ADDRESS_STATE
            DSTATE.sync_state = FULL_SYNC_STATE;
        otherwise


    LogElem(ANALYZE, e, "CONTEXT");
    return;
```

D9.5.6 Exception element

R$_{PJFBL}$ For an *Exception element*, a trace analyzer must analyze each instruction from the current address, up to but not including the exception return address that the element provides. The PE has executed each instruction in that address range. The number of instructions that are executed can be zero.

Note

Trace analysis tools must be aware, that if PE execution is at the top of memory space, the address that the *Exception element* provides might be lower than the target address of the most recent *P0 element*.

D9.5.6.1 AnalyzeException()

```
// AnalyzeException()
// ==================
// Analyzes an exception element.


AnalyzeException(Element e)
    CheckForReturnStackMatch();
```

```
if DSTATE.sync_state == CONTEXT_STATE then
    DSTATE.sync_state = NOT_SYNC_STATE;
if DSTATE.sync_state != FULL_SYNC_STATE then
    LogDecompressor(ANALYZE,
                    "Entered exception while expecting address or context");
    return;


integer PER = UInt(e.payload.address);
if (ExceptionWithUnknownAddress(e)) then
    continue_forward = FALSE;
elsif UInt(DSTATE.current_analyzer_state.address) < PER then
    continue_forward = TRUE;
else
    continue_forward = FALSE;


// Continue logging instructions until we reach the specified address.
while continue_forward do
    if !ProgramImage.DecodeAvailable() then
        LogDecompressor(FINAL_OUTPUT,
                        "unprocessed_execution due to no decode available");
        DSTATE.sync_state = CONTEXT_STATE;
        if DSTATE.return_stack_clear_pending then
            ReturnStack.Reset();
        return;


    decoded_inst = ProgramImage.DecodeNextInst();


    if decoded_inst.branchtype == InstType_OTHER then
        ReconstructState nxt_state = DSTATE.current_analyzer_state;
        nxt_state.address = nxt_state.address + decoded_inst.size;
        DSTATE.next_analyzer_state = nxt_state;
        if !DSTATE.next_analyzer_state.sixty_four_bit then
            // mask off the left-most bits
            DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
    else
        ProcessBranchInstruction(decoded_inst, FALSE);
        UpdateReturnStack(decoded_inst);


    OutputInstruction(decoded_inst);


    bits(64) next_addr = DSTATE.current_analyzer_state.address +
decoded_inst.size;
    DSTATE.current_analyzer_state.address = next_addr;
    if !DSTATE.current_analyzer_state.sixty_four_bit then
        // mask off the left-most bits
        DSTATE.current_analyzer_state.address<63:32> = Replicate('0', 32);


    if UInt(DSTATE.current_analyzer_state.address) >= PER then
        continue_forward = FALSE;


    if DSTATE.return_stack_clear_pending then
```

```
                    ReturnStack.Reset();


            LogElem(ANALYZE, e,
                    "EXC, type:  " ++ ExcepTypeToStr(e.payload.exception_type) ++
                    " traced upto address " ++ DSTATE.current_analyzer_state.address);
            return;
```

D9.5.7 Source Address element

R_{VDRPP} A *Source Address element* indicates that one or more instructions have been traced, up to and including the instruction at the address associated with the element.

R_{YJHMV} A *Source Address element* indicates that the instruction at the address associated with the element was taken.

I_{GWZMF} A trace analyzer must analyze each instruction in the program image from the current address until it analyzes the instruction at the address associated with the *Source Address element*. This indicates that the PE has executed each instruction between the current address and that instruction, and each *P0 instruction* except the final instruction was not taken.

D9.5.7.1 AnalyzeSourceAddress()

```
// AnalyzeSourceAddress()
// =====================
// Analyzes a source address element.


AnalyzeSourceAddress(Element e)
    LogElem(ANALYZE, e, "SOURCE ADDR " ++ e.payload.address);
    CheckForReturnStackMatch();


    if DSTATE.sync_state == ADDRESS_STATE then
        DSTATE.sync_state = NOT_SYNC_STATE;
    if DSTATE.sync_state != FULL_SYNC_STATE then
        // If we are unsure of context or address then we cannot meaningfully
        // analyze the source address.
        return;


    DSTATE.most_recent_branch_was_taken = FALSE;
    integer address = UInt(e.payload.address);


    // Continue logging instructions until we hit the specified address.
    while (UInt(DSTATE.current_analyzer_state.address) <= address) do
        if !ProgramImage.DecodeAvailable() then
            DSTATE.sync_state = CONTEXT_STATE;
            LogDecompressor(FINAL_OUTPUT,
                            "unprocessed_execution due to no decode available");
            if DSTATE.return_stack_clear_pending then
                ReturnStack.Reset();
            return;


        decoded_inst = ProgramImage.DecodeNextInst();
```

```
                    if decoded_inst.branchtype == InstType_OTHER then
                        ReconstructState nxt_state = DSTATE.current_analyzer_state;
                        nxt_state.address = nxt_state.address + decoded_inst.size;
                        DSTATE.next_analyzer_state = nxt_state;
                        if !DSTATE.next_analyzer_state.sixty_four_bit then
                            DSTATE.next_analyzer_state.address<63:32> = Replicate('0', 32);
                    else
                        if DSTATE.current_analyzer_state.address == e.payload.address then
                            DSTATE.most_recent_branch_was_taken = TRUE;
                        ProcessBranchInstruction(decoded_inst,
                                                 DSTATE.most_recent_branch_was_taken);


                    cur_inst_is_branch = TRUE;
                    UpdateReturnStack(decoded_inst);


                OutputInstruction(decoded_inst);


                DSTATE.current_analyzer_state = DSTATE.next_analyzer_state;


        if DSTATE.return_stack_clear_pending then
            ReturnStack.Reset();
        return;
```

D9.5.8 Target Address element

D9.5.8.1 AnalyzeTargetAddress()

```
// AnalyzeTargetAddress()
// ======================
// Analyzes a target address element.

AnalyzeTargetAddress(Element e)
    DSTATE.current_analyzer_state.address = e.payload.address;
    DSTATE.current_analyzer_state.sub_isa = e.payload.sub_isa;


    case DSTATE.sync_state of
        when NOT_SYNC_STATE
            DSTATE.sync_state = ADDRESS_STATE;
        when CONTEXT_STATE
            DSTATE.sync_state = FULL_SYNC_STATE;
        otherwise


    LogElem(ANALYZE, e, "ADDR " ++ e.payload.address);
    return;
```

D9.5.9   Trace Info element

D9.5.9.1   AnalyzeTraceInfo()

```
// AnalyzeTraceInfo()
// ==================
// Analyzes a trace info element.


AnalyzeTraceInfo(Element e)
    CheckForReturnStackMatch();
    return_stack_clear_pending = TRUE;
    LogElem(ANALYZE, e, "TRACE_INFO");
    return;
```

D9.5.10   Trace On element

D9.5.10.1   AnalyzeTraceOn()

```
// AnalyzeTraceOn()
// ================
// Analyzes a trace on element.


AnalyzeTraceOn(Element e)
    return_stack_clear_pending = TRUE;
    DSTATE.sync_state = NOT_SYNC_STATE;
    LogElem(ANALYZE, e, "TRACE_ON");
    return;
```

D9.5.11   Mispredict element

R_{ZRVMZ} When a *Mispredict element* corresponds to an *Atom element* for a direct *P0 instruction*, before the trace analyzer can calculate the target of the direct *P0 instruction*, it must apply any applicable *Mispredict elements* so that it can determine whether it is an E *Atom element* or an N *Atom element*.

D9.5.11.1   AnalyzeMispredict()

```
// AnalyzeMispredict()
// ===================
// Analyzes a mispredict element.


AnalyzeMispredict(Element e)
    DSTATE.most_recent_branch_was_taken = !DSTATE.most_recent_branch_was_taken;


    ReconstructState nxt_state;
    nxt_state = UpdateBranchState(DSTATE.most_recent_branch_decoded_inst,
                                  DSTATE.most_recent_branch_state,
                                  DSTATE.most_recent_branch_was_taken);
```

```
        DSTATE.current_analyzer_state = nxt_state;


        LogDecompressor(ANALYZE, "MISPREDICT");
        return;
```

D9.5.12 ETEEvent element

D9.5.12.1 AnalyzeEvent()

```
// AnalyzeEvent()
// ==============
// Analyzes an event element.


AnalyzeEvent(Element e)
    LogElem(ANALYZE, e, "EVENT, id:  " ++ e.payload.event_id);
    LogDecompressor(FINAL_OUTPUT,
                    "Event " ++ e.payload.event_id ++ " occurred");
    return;
```

D9.5.13 Discard element

I_{JYPQC} When a trace analyzer encounters a *Discard element* it must be aware that if the last committed *P0 element* is a conditional *P0 instruction*, the E or N status of that *Atom element* might not be correct. This is because the trace unit might be unable to generate any *Mispredict elements* that the conditional *P0 instruction* might require.

I_{HPQFK} If the last *P0 instruction* is an indirect *P0 instruction* then the target address indicated in the trace stream might be incorrect. This is because the trace unit might be unable to generate any *Target Address elements* that the indirect *P0 instruction* might require.

D9.5.13.1 AnalyzeDiscard()

```
// AnalyzeDiscard()
// ================
// Analyzes a discard element.


AnalyzeDiscard(Element e)
    DSTATE.sync_state = NOT_SYNC_STATE;
    LogElem(ANALYZE, e, "Discard has occurred");
    return;
```

D9.5.14 Overflow element

An *Overflow element* indicates that some of the trace might be lost.

D9.5.14.1 AnalyzeOverflow()

```
// AnalyzeOverflow()
// =================
```

```
        // Analyzes an overflow element.


AnalyzeOverflow(Element e)
    DSTATE.sync_state = NOT_SYNC_STATE;
    LogElem(ANALYZE, e, "OVERFLOW has occurred");
    return;
```

D9.5.15   Q element

When a trace analyzer encounters a *Q element* which has a count of M executed instructions, it must proceed through the program image, analyzing each instruction until it has analyzed M instructions. If it encounters a conditional *P0 instruction*, the status of the condition code check for that instruction is UNKNOWN. The status of these *P0 instructions* is not explicitly given in the trace element stream but it might be possible to infer the status of a given *P0 instruction* that is based on other trace that is generated. After the trace analyzer has analyzed M instructions, the following *Target Address element* indicates where PE execution continues.

D9.5.15.1   AnalyzeQ()

```
// AnalyzeQ()
// =========
// Analyzes a Q element.


AnalyzeQ(Element e)
    CheckForReturnStackMatch();


    q_with_count = e.payload.count > 0;
    if q_with_count then
        further_analysis_possible = TRUE;
    else
        LogDecompressor(FINAL_OUTPUT, "Unprocessed execution due to Q element");
        further_analysis_possible = FALSE;
        DSTATE.sync_state = CONTEXT_STATE;
        // If we have no count then just wait to resync, it is not safe to guess


    i = 0;
    while further_analysis_possible do
        if ProgramImage.DecodeAvailable() then
            decoded_inst = ProgramImage.DecodeNextInst();
            addr = DSTATE.current_analyzer_state.address + decoded_inst.size;
            DSTATE.current_analyzer_state.address = addr;
        else
            DSTATE.sync_state = CONTEXT_STATE;
            LogDecompressor(FINAL_OUTPUT,
                            "Unprocessed execution due to no decode available");
            return;


        i = i + 1;


        further_analysis_possible = (i < e.payload.count);
        OutputInstruction(decoded_inst);
```

```
        LogElem(ANALYZE, e, "Q");
        return;
```

D9.5.16 Timestamp element

D9.5.16.1 AnalyzeTimestamp()

```
// AnalyzeTimestamp()
// ==================
// Analyzes a timestamp element.


AnalyzeTimestamp(Element e)
    LogElem(ANALYZE, e, "TIMESTAMP " ++ e.payload.timestamp);
    return;
```

D9.5.17 Cycle Count element

I_{HTQDP} To produce a total cycle count, a trace analyzer can cumulatively add the values from all *Cycle Count elements*.

R_{TVWLZ} A trace analyzer must not use the cycle count values in *Timestamp elements* to produce a total cycle count. Such cycle count values are not a *Cycle Count element*.

D9.5.17.1 AnalyzeCycleCount()

```
// AnalyzeCycleCount()
// ===================
// Analyzes a cycle count element.


AnalyzeCycleCount(Element e)
    LogDecompressor(ANALYZE,
                    "CYCLE_CNT: " ++ e.payload.cycle_count ++
                    " cycles since last CC");
    return;
```

D9.5.18 Functions

D9.5.18.1 OutputInstruction()

```
// OutputInstruction()
// ===================
// Output the traced instruction to the software that is consuming the output
// of the trace analyzer.


OutputInstruction(DecodedInst inst);
```

D9.5.18.2 CheckForReturnStackMatch()

```
// CheckForReturnStackMatch()
// =========================
// Check if there is a return stack match, and log the result.


CheckForReturnStackMatch()
    if DSTATE.sync_state == CONTEXT_STATE then
        if ReturnStack.IsEmpty() then
            LogDecompressor(ANALYZE, "No return stack entries available");
        else
            ReturnStackEntry top = ReturnStack.Pop();
            DSTATE.current_analyzer_state.address = top.address;
            DSTATE.current_analyzer_state.sub_isa = top.sub_isa;
            DSTATE.sync_state = FULL_SYNC_STATE;
            LogDecompressor(ANALYZE, "Popping match from return stack");
            LogReturnStack();


    return;
```

D9.5.18.3 UpdateBranchState()

```
// UpdateBranchState()
// ===================
// Returns an updated state based on what was executed.


ReconstructState UpdateBranchState(DecodedInst inst,
                                   ReconstructState in_state,
                                   boolean branch_was_taken)


    out_state = DSTATE.current_analyzer_state;
    out_state.address = in_state.address;
    out_state.sixty_four_bit = in_state.sixty_four_bit;
    out_state.sub_isa = in_state.sub_isa;


    if branch_was_taken then
        if inst.branchtype == InstType_BRANCH_INDIR then
            DSTATE.sync_state = CONTEXT_STATE;
            LogDecompressor(ANALYZE,
                            "Indirect branch - " ++
                            "waiting for address element or " ++
                            "return stack match...");
        else
            if inst.branchtype == InstType_BRANCH_DIR then
                out_state.address = out_state.address + inst.addressoffset;
            else
                out_state.address = out_state.address + inst.size;
            if !in_state.sixty_four_bit then
                out_state.address<63:32> = Zeros();
            out_state.sub_isa = inst.next_sub_isa;
    else
```

```
                out_state.address = out_state.address + inst.size;
            if !out_state.sixty_four_bit then
                out_state.address<63:32> = Zeros();


        return out_state;
```

D9.5.18.4 ProcessBranchInstruction()

```
// ProcessBranchInstruction()
// ==========================
// Store current state before a branch instruction, as it could change if there
// is a misprediction.


ProcessBranchInstruction(DecodedInst inst, boolean branch_was_taken)
        DSTATE.most_recent_branch_state = DSTATE.current_analyzer_state;
        DSTATE.most_recent_branch_decoded_inst = inst;
        DSTATE.most_recent_branch_was_taken = branch_was_taken;
        DSTATE.next_analyzer_state = UpdateBranchState(inst,
                                            DSTATE.current_analyzer_state,
                                            branch_was_taken);


        return;
```

D9.5.18.5 DecodedInst

```
// DecodedInst
// ===========
// Data extracted from an instruction.


type DecodedInst is (
    bits(32) instruction,    // The instruction itself
    InstType branchtype,     // Type of P0 instruction
    boolean is_link,         // Is it a linking branch?
    integer size,            // Size (32 or 16)
    SubISA next_sub_isa,     // sub_isa of the following instruction to be
                             // executed
    bits(64) addressoffset
)
```

Chapter D10
Programming

D10.1 Example code sequences

S_{PWXHJ} The enabling sequence should be from the trace sink, such as the trace buffer, to the trace unit. This is to ensure the trace sink is ready to capture trace before the trace unit generates any trace.

S_{WPVBX} The disabling sequence should be from the trace unit to the trace sink. This is to ensure that any buffered trace reaches the trace sink while the trace sink is still enabled.

D10.1.1 Enabling the trace unit

Listing D10.1: Example code sequence to enable the trace unit

```
1        ;; Program the trace unit registers, except TRCPRGCTLR
2        ISB                    ;; Synchronize the System Register updates.
3        MOV x0, #0x1
4        MSR TRCPRGCTLR, x0    ;; Enable the ETE.
5        ;; Wait for TRCSTATR.IDLE==0
6  poll_idle
7        ISB                    ;; Synchronize the write to TRCPRGCTLR
8        MRS x1, TRCSTATR
9        TBNZ x1, #1, poll_idle
```

D10.1.2 Disabling the trace unit

Listing D10.2: Example code sequence to disable the trace unit

```
1        STP x0, x1, [sp, #-16]!
2
3        MRS x0, TRFCR_EL1    ;; Save the current programming of TRFCR_EL1.
```

```
 4        MOV x1, #0x3
 5        BIC x1, x0, x1
 6        MSR TRFCR_EL1, x1    ;; Clear the values of TRFCR_EL1.ExTRE.
 7                             ;; to put the PE in to a Trace Prohibited region
 8        ISB                  ;; Synchronize the entry to the Trace Prohibited region
 9        TSB CSYNC            ;; Ensure that all trace has reached the
10                             ;; trace buffer and address translations have
11                             ;; taken place.
12        MOV x1, #0x0
13        MSR TRCPRGCTLR, x1   ;; Disable the trace unit
14        ;; Wait for TRCSTATR.IDLE==1 and TRCSTATR.PMSTABLE==1
15 poll_idle
16        ISB
17        MRS x1, TRCSTATR
18        AND x1, x1, #3
19        CMP x1, #3
20        B.NE poll_idle
21
22        MSR TRFCR_EL1, x0    ;; Restore the programming of TRFCR_EL1.
23
24        LDP x0, x1, [sp], #16
```

D10.1.3 Example save restore routine

The following example sequences are for saving the trace unit state over a power down, and restoring the trace unit state when power is restored.

Listing D10.3: Example code sequence to save the trace unit state

```
 1        STP x0, x1, [sp, #-16]!
 2        ;; Enter a Trace Prohibited region
 3        MRS x0, TRFCR_EL1    ;; Save the current programming of TRFCR_EL1.
 4        MOV x1, #0x3
 5        BIC x1, x0, x1
 6        MSR TRFCR_EL1, x1    ;; Clear the values of TRFCR_EL1.ExTRE.
 7        ISB                  ;; Synchronizes the entry to the Trace Prohibited region
 8        TSB CSYNC            ;; Ensure the trace unit is synchronized
 9        MOV x1, #1
10        MSR OSLAR_EL1, x1    ;; Lock the OS lock
11        ;; Wait for TRCSTATR.PMSTABLE==1
12 poll_pmstable
13        ISB
14        MRS x1, TRCSTATR
15        TST x1, #2
16        B.EQ poll_pmstable
17
18        MSR TRFCR_EL1, x0    ;; Restore the programming of TRFCR_EL1.
19
20 <save the trace unit registers, including TRCPRGCTLR>
21
22        ;; Wait for TRCSTATR.IDLE==1
23 poll_idle
24        ISB
25        MRS x1, TRCSTATR
26        TST x1, #1
27        B.EQ poll_idle
28
29        LDP x0, x1, [sp], #16
```

Listing D10.4: Example code sequence to restore the trace unit state

```
 1
 2 <restore the trace unit registers, including TRCPRGCTLR>
 3
 4        STP x0, x1, [sp, #-16]!
 5        MOV x0, #0
 6        MSR OSLAR_EL1, x0    ;; Clear the OS lock
 7        LDP x0, x1, [sp], #16
 8        ISB
```

I_{DHQGM} When programming the trace unit, it is important to be aware that the loops that poll TRCSTATR in Figure D8.2 might never complete. Arm recommends that such scenarios are avoided except in rare conditions. However, some system conditions might prevent a trace unit from either leaving the idle state or becoming idle. In particular, a trace unit might never become idle if the trace unit is unable to output all trace due to a system condition.

S_{TTPKR} If multiple reads of TRCSTATR are required, a Context synchronization event is required between each read of TRCSTATR to ensure any change to the trace unit state is observed.

D10.2 Minimal programming

The table Table D10.1 gives the values for programming the trace unit to enable trace for tracing of a single process at EL0.

Table D10.1: Minimal programming values

Register	Value	Description
TRCCONFIGR	0x000018C1	Enable: - The return stack. - Global timestamping. - Context identifier tracing. - Virtual context identifier tracing.
TRCEVENTCTL0R	0x00000000	Disable all event tracing
TRCEVENTCTL1R	0x00000000	
TRCSTALLCTLR	0x00000000	Disable stalling, if implemented
TRCSYNCPR	0x0000000C	Enable trace protocol synchronization every 4096 bytes of trace
TRCTRACEIDR	Nonzero	Set a value for the trace ID
TRCTSCTLR	0x00000000	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.
TRCVICTLR	0x006F0201	Enable ViewInst to trace everything, with the start/stop logic started Disable: - EL1 in Non-secure state. - EL2 in Non-secure state. - EL3-EL0 in Secure state.
TRCVIIECTLR	0x00000000	No address range filtering for logic started
TRCVISSCTLR	0x00000000	No start or stop points for ViewInst

S_{HBZVM} Disabling tracing of Secure state might not be strictly necessary as secure tracing might be disabled by MDCR_EL3.STE, but Arm recommends not enabling trace for un-required Exception levels, to limit the amount of trace.

S_{QCWTJ} Disabling tracing of EL1 and EL2 of Non-secure state might not be strictly necessary as non-secure tracing might be disabled by TRFCR_EL2.E2TRE and TRFCR_EL1.E1TRE, but Arm recommends not enabling trace for un-required Exception levels to limit the amount of trace.

D10.3 Filtering models

Different trace applications require different usage models of a trace unit. For example, one trace application might only require basic program flow trace, whereas another might require tracing of a specific program function.

The ETE architecture provides for each of these usage models. A trace unit can be implemented with a particular set of implementation options, so that a trade-off between functionality and cost can be achieved in meeting the requirements of a trace application. Discovery of the particular set of implementation is achieved by reading TRCIDR0 to TRCIDR13.

In a trace unit that includes all implementation options, the simplest way to use the trace unit is to turn on tracing of all aspects of *Processing Element* (PE) operation and let the trace analyzer pick out the required information. However, full trace comes at a high cost in terms of port bandwidth and trace storage. These costs have an impact on the design of a system, so that a higher pin count and larger buffers might be required.

A trace unit provides on-chip filtering, that facilitates a reduction of the trace bandwidth and therefore provides for a lower system cost. By suspending and enabling trace during a trace that is run to suit the particular requirements of the trace run, the best use of both port bandwidth and trace storage can be made.

The ETE architecture provides the following basic filtering models:

Continuous tracing This is where no filtering is applied. The following modes can be used:

- Continuous instruction tracing only, where only the instruction trace stream is output.

Instruction-based filtering This is where instruction tracing, and data tracing if it is implemented and enabled, is active only for certain code sequences, such as for a particular process or function.

For all the possible filtering modes, the trace unit can be programmed before a trace run to enable various options, including:

- Context identifier tracing, if implemented, to indicate to a trace analyzer the Context identifier value.
- Virtual context identifier tracing, if implemented, to distinguish between different virtual machines.
- Cycle counting, to enable a trace analyzer to analyze program performance.
- Global timestamping, if implemented, to enable correlation of the two trace streams with other trace sources in the system.
- Branch broadcasting, if implemented, to force all taken *P0 instruction* targets to be traced with an explicit target address.

A trace unit is programmed for continuous instruction tracing when no filtering is applied to the instruction trace stream.

When a trace unit is programmed for continuous instruction tracing, ViewInst is always active during a trace run. See D6.8 *Filtering trace generation*.

D10.4 Filtering used the exclude function

Register	Value	Description
TRCCONFIGR	`0x000018C1`	Enable: • the return stack, • global timestamping, • Context identifier, • Virtual context identifier tracing.
TRCEVENTCTL0R	`0x00000000`	Disable all event tracing.
TRCEVENTCTL1R	`0x00000000`	
TRCSTALLCTLR	`0x00000000`	Disable stalling, if implemented.
TRCSYNCPR	`0x0000000C`	Enable trace protocol synchronization every 4096 bytes of trace.
TRCTRACEIDR	Nonzero	Set a value for the trace ID.
TRCTSCTLR	`0x00000000`	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.
TRCVICTLR	`0x006F0201`	Enable ViewInst to trace everything, with the start/stop logic started. Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL3-EL0 in Secure state. tracing.
TRCVIIECTLR	`0x00010000`	Use ARC0 for the exclude logic.
TRCVISSCTLR	`0x00000000`	No start or stop points for ViewInst.
TRCACATR0	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR0	*Start Address*	
TRCACATR1	`0x00000000`	The comparator status to match on all instructions at this Virtual address
TRCACVR1	*End Address*	

D10.5 Filtering used the include function

Register	Value	Description
TRCCONFIGR	0x000018C1	Enable: • the return stack, • global timestamping, • Context identifier, • Virtual context identifier tracing.
TRCEVENTCTL0R	0x00000000	Disable all event tracing.
TRCEVENTCTL1R	0x00000000	
TRCSTALLCTLR	0x00000000	Disable stalling, if implemented.
TRCSYNCPR	0x0000000C	Enable trace protocol synchronization every 4096 bytes of trace.
TRCTRACEIDR	Nonzero	Set a value for the trace ID.
TRCTSCTLR	0x00000000	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace protocol synchronization.
TRCVICTLR	0x006F0201	Enable ViewInst to trace everything, with the start/stop logic started. Disable: • EL1 in Non-secure state. • EL2 in Non-secure state. • EL3-EL0 in Secure state. tracing.
TRCVIIECTLR	0x00000001	Use ARC0 for the include logic.
TRCVISSCTLR	0x00000000	No start or stop points for ViewInst.
TRCACATR0	0x00000000	The comparator status to match on all instructions at this Virtual address
TRCACVR0	*Start Address*	
TRCACATR1	0x00000000	The comparator status to match on all instructions at this Virtual address
TRCACVR1	*End Address*	

D10.6 OS Save and Restore routines

When the PE is context switching of trace unit the following registers need to save and restored. Not all these registers are necessarily implemented for all implementations. Please refer to the register description page for information on if the register is implemented.

- TRCPRGCTLR
- TRCCONFIGR
- TRCAUXCTLR
- TRCEVENTCTL0R
- TRCEVENTCTL1R
- TRCRSR
- TRCSTALLCTLR
- TRCTSCTLR

- TRCSYNCPR
- TRCCCCTLR
- TRCBBCTLR
- TRCTRACEIDR
- TRCQCTLR
- TRCVICTLR
- TRCVIIECTLR
- TRCVISSCTLR
- TRCVIPCSSCTLR
- TRCSEQEVR<n>
- TRCSEQRSTEVR
- TRCSEQSTR
- TRCEXTINSELR<n>
- TRCCNTRLDVR<n>
- TRCCNTCTLR<n>
- TRCCNTVR<n>
- TRCIMSPEC<n>
- TRCRSCTLR<n>
- TRCSSCCR<n>
- TRCSSCSR<n>
- TRCSSPCICR<n>
- TRCACVR<n>[31:0]
- TRCACVR<n>[63:32]
- TRCACATR<n>[31:0]
- TRCACATR<n>[63:32]
- TRCCIDCVR<n>[31:0]
- TRCCIDCVR<n>[63:32]
- TRCVMIDCVR<n>[31:0]
- TRCVMIDCVR<n>[63:32]
- TRCCIDCCTLR0
- TRCCIDCCTLR1
- TRCVMIDCCTLR0
- TRCVMIDCCTLR1

S_{HYDYT} If the trace unit has not been programmed since the last context switch then there is no requirement to save and restore the registers.

If the programming of the trace unit is known and has not changed only the following registers are required to saved.

- TRCRSR
- TRCVICTLR
- TRCSEQSTR
- TRCCNTVR<n>
- TRCIMSPEC<n>, if implemented and has dynamic state.
- TRCSSCSR<n>

If the trace unit is to powered down then the following registers must also be saved and restored.

- TRCCLAIMCLR on saving.
- TRCCLAIMSET on restoring.

Chapter D11
Trace Examples

D11.1 Basic Examples

D11.1.1 Simple example of basic program trace

Table D11.1: Example of program trace

	Execution	**Trace elements**	**Notes**
0x1000	B -> 0x2000	trace_info(...) trace_on() context() address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MOV		
0x2004	LDR		
0x2008	CMP		
0x200C	BEQ -> 0x3000	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BEQ instruction.
0x2010	STR		
	IRQ	exception (IRQ,0x2014)	An IRQ occurs. The *Exception element* indicates the STR instruction was executed.

D11.1.2 Simple example of basic program trace filtering applied

The example below shows basic program trace when filtering is applied. In this example, the trace unit is programmed to exclude all instructions in the address range `0x2000` to `0x200F` inclusive, and the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table D11.2: Example of program trace with filtering

	Execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	Y	trace_info(...) trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
`0x2000`	`MOV`	N		
`0x2004`	`LDR`	N		
`0x2008`	`CMP`	N		
`0x200C`	`BEQ -> 0x3000`	N		
`0x2010`	`STR`	Y	trace_on() address(`0x2010`)	
	`IRQ`		exception (IRQ,0x2014)	An IRQ occurs. The *Exception element* indicates the `STR` instruction was executed.

Chapter D12
Examples of basic program trace when exceptions occur

This section contains three examples:

- Basic program trace when an exception occurs, example one.
- Basic program trace when an exception occurs, example two.
- Example of basic program trace when two consecutive exceptions occur.

D12.1 Basic program trace when an exception occurs, example one

The example in Table D12.1 shows basic program trace and demonstrates the canceling of some speculative execution because of an exception. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table D12.1: Example of basic program trace when an exception occurs, example one

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(`0x1000`) atom(E)	Tracing begins here, therefore the trace unit must generate both: • *Context element*. • *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
`0x2000`	`MOV`		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the `BEQ` instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	Cancel back to and including 0x2000	cancel(1)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the three instructions that it implied. **Note** Although *Processing Element* (PE) execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction.
	IRQ	exception(IRQ,`0x2000`)	The trace unit generates an *Exception element* with the address `0x2000`, which indicates no instructions have executed since the target of the branch at `0x1000`.
	commit all execution	commit(2)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* that was generated for the IRQ exception.

D12.2 Basic program trace when an exception occurs, example two

The example in Table D12.2 shows basic program trace, and shows the trace generated when a synchronous Data Abort occurs. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table D12.2: Example of basic program trace when an exception occurs, example two

	Execution	Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info() trace_on() context() address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • *Context element.* • *Target Address element.* The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
`0x2000`	`MOV`		None of these instructions are traced as P0 elements, therefore no trace elements are generated.
`0x2004`	`LDR`		
`0x2008`	`CMP`		
`0x200C`	`BEQ -> 0x3000` (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BEQ instruction.
`0x2010`	`STR`		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	LDR aborts Cancel back to and including `0x2004`	cancel(1) exception_(data fault, `0x2004`)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the four instructions that it implied. **Note** Although PE execution has also canceled execution of the `STR` instruction, the trace analyzer is unaware of this, because the `STR` instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the `STR` instruction. The data fault exception occurred at `0x2004`. The *Exception element* indicates the `MOV` instruction at `0x2000` was executed. In summary: 1. The `MOV` instruction was first implied by the N *Atom element* at `0x200C`. However, the trace analyzer canceled this because of the *Cancel element*. 2. The `MOV` instruction is now implied by the *Exception element*.
	commit all execution	commit(2)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* that was generated for the IRQ exception.

D12.3 Example of basic program trace when two consecutive exceptions occur

The example in Table D12.3 extends the example shown in Table D12.2, and shows how exceptions are traced when two exceptions occur without any execution between them. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table D12.3: Example of basic program trace when two consecutive exceptions occur

	Execution	Trace elements	Notes
`0x1000`	B -> 0x2000	trace_info() trace_on() context() address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • *Context element*. • *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
`0x2000`	MOV		None of these instructions are traced as P0 elements, therefore no trace elements are generated.
`0x2004`	LDR		
`0x2008`	CMP		
`0x200C`	BEQ -> 0x3000 (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BEQ instruction.
`0x2010`	STR		This instruction is not traced as a *P0 element*, therefore no trace element is generated.
	LDR aborts Cancel back to and including `0x2004`	cancel(1) exception_(data fault, `0x2004`)	This cancels the N *Atom element* that was generated for the branch at `0x200C`. The trace analyzer must discard the N *Atom element*, plus the four instructions that it implied. **Note** Although PE execution has also canceled execution of the STR instruction, the trace analyzer is unaware of this, because the STR instruction was never traced. This is because no *P0 elements* were generated that would indicate execution of the STR instruction. The data fault exception occurred at `0x2004`. The *Exception element* indicates the MOV instruction at `0x2000` was executed. In summary: 1. The MOV instruction was first implied by the N *Atom element* at `0x200C`. However, the trace analyzer canceled this because of the *Cancel element*. 2. The MOV instruction is now implied by the *Exception element*.
	IRQ	address(`0x4000`) exception(IRQ,`0x4000`)	This *Exception element* contains the address of the exception vector of the DataFault exception. This implies that no instructions have executed since the DataFault exception.

Execution	Trace elements	Notes
commit all execution	commit(3)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`, plus the *Exception element* generated for the Data fault exception and the *Exception element* that was generated for the IRQ exception.

Chapter D13
Examples of changes in Context

When the *Processing Element* (PE) executes an instruction that changes the execution context, the exact time at which the new element is traced depends on the PE operation after the write. An example of an instruction that changes the execution context is an instruction that writes a value to the CONTEXTIDR_EL1. See D2.3.3 *Context Element* for more information about the rules controlling the generation of Context elements. This section provides examples of PE trace that contain changes of execution context to illustrate these rules. This section is split into the following:

- Exception in software executed after context synchronization.
- Exception immediately after ISB.
- Exception immediately before ISB.

D13.1 Exception in software executed after context synchronization

Table D13.1 shows a write to the CONTEXTIDR_EL1 register, followed by an `ISB` to synchronize that write, followed by an exception that changes the context again.

Table D13.1: Program trace containing a context changing operation

	Execution	Context	Trace elements	Notes
0x1000	B-> 0x2000	0xAA	trace_info() trace_on() context(0xAA) address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MSR CONTEXTIDR	0xAA		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA.
0x2004	ADD	0xAA or 0xBB		
0x2008	ISB	0xAA or 0xBB	atom(E)	The trace unit generates an E *Atom element*, because the `ISB` is a Context synchronization event. All execution is traced as executing in context 0xAA.
0x200C	SUB	0xBB	context(0xBB)	A *Context element* is traced to indicate the new context.
	IRQ	0xBB	exception(IRQ,0x2010)	An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context(0xCC) address(0x3000) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by an *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

D13.2 Exception immediately after ISB

Table D13.2 shows the same execution as Table D13.1 but the exception occurs one instruction earlier. This means that no execution takes place between the ISB and the exception.

Table D13.2: Program trace containing a context changing operation (exception immediately after ISB)

	Execution	Context	Trace elements	Notes
0x1000	B-> 0x2000	0xAA	trace_info() trace_on() context(0xAA) address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MSR CONTEXTIDR	0xAA		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA.
0x2004	ADD	0xAA or 0xBB		
0x2008	ISB	0xAA or 0xBB	atom(E)	The trace unit generates an E *Atom element*, because the ISB is a Context synchronization event. All execution is traced as executing in context 0xAA.
	IRQ	0xBB	context(0xBB) exception(IRQ,0x200C)	A *Context element* is traced to indicate the new context. An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context(0xCC) address(0x3000) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by an *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

D13.3 Exception immediately before ISB

Table D13.3 the same as Table D13.2 but the exception occurs one instruction earlier. This means that the exception occurs before the ISB instruction that was present in previous examples.

Table D13.3: Program trace containing a context changing operation (exception immediately before ISB)

	Execution	Context	Trace elements	Notes
0x1000	B-> 0x2000	0xAA	trace_info() trace_on() context(0xAA) address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • A *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MSR CONTEXTIDR	0xAA		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated. The instructions might be executed from context 0xAA or 0xBB but they are always traced as occurring from context 0xAA.
0x2004	ADD	0xAA or 0xBB		
	IRQ	0xAA or 0xBB	exception(IRQ,0x2008)	An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context(0xCC) address(0x3000) atom(E)	A *Context element* is traced to indicate the new context. A *Target Address element* is also traced, because an *Exception element* is always followed by an *Target Address element* to indicate the address that the exception has been taken to. Finally, the instruction executed is a taken branch, so the trace unit must generate an E *Atom element*.

Chapter D14
An example of the use of the trace unit return stack

This section contains two examples of tracing the same piece of program code. However:

- In the first example the trace unit return stack is disabled.
- In the second example trace unit the return stack is enabled.

The examples demonstrate that use of the trace unit return stack can help to reduce the amount of trace generated. Table D14.1 the first example, and Table D14.2 shows the second example. In these examples, the trace unit is programmed for basic program flow, where only branch instructions are traced as *P0 instructions*, and is programmed to start tracing when the instruction at 0x1000 is accessed.

Table D14.1: Table A-14 Basic program trace when Branch with Link instructions are executed and the return stack is disabled

	Execution	Trace elements	Notes
0x1000	BL -> 0x2000	trace_info() trace_on() context() address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*.
0x2000	MOV		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
0x2004	LDR		
0x2008	CMP		

	Execution	Trace elements	Notes
0x200C	BLEQ -> 0x3000 (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BLEQ instruction.
0x2010	STR		
0x2014	BX LR	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the STR and the BX instructions.
0x1004	MOV	address(0x1004)	This instruction is not traced as a *P0 element*, therefore no trace element is generated. However, the last instruction executed was a taken indirect branch instruction, so the trace unit generates an *Target Address element* to indicate the target of that branch.
0x1008	B -> 0x4000	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the MOV instruction at 0x1004 and the B instruction.
	commit all execution	commit(4)	This commits all four of the following: • The E *Atom element* generated for the branch at 0x1000. • The N *Atom element* generated for the branch at 0x200C. • The E *Atom element* generated for the branch at 0x2014. • The E *Atom element* generated for the branch at 0x1008.

Table D14.2: Table A-15 Basic program trace when Branch with Link instructions are executed and the return stack is enabled

	Execution	Trace elements	Notes
0x1000	BL -> 0x2000	trace_info() trace_on() context() address(0x1000) atom(E)	Tracing begins here, therefore the trace unit must generate both: • A *Context element*. • An *Target Address element*. The instruction executed is a taken branch, so in addition, the trace unit must generate an E *Atom element*. In addition, because the return stack is enabled, the Branch with Link instruction causes the address 0x1004 to be pushed onto the trace unit return stack.
0x2000	MOV		None of these instructions are traced as *P0 elements*, therefore no trace elements are generated.
0x2004	LDR		
0x2008	CMP		

	Execution	**Trace elements**	**Notes**
0x200C	BLEQ -> 0x3000 (not taken)	atom(N)	This branch is not taken, so the trace unit generates an N *Atom element*. The N *Atom element* implies the execution of the three previous instructions and the BLEQ instruction. Nothing is pushed onto the trace unit return stack because the branch is not taken.
0x2010	STR		
0x2014	BX LR	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the STR and the BX instructions.
0x1004	MOV		This instruction is not traced as a *P0 element*, therefore no trace element is generated. The address of this instruction matches the top entry on the trace unit return stack. Therefore, the trace analyzer knows to restart program execution here and an *Target Address element* is not required. The top entry on the return stack, address 0x1004, is popped from the return stack.
0x1008	B -> 0x4000	atom(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the MOV instruction at 0x1004 and the B instruction.
	commit all execution	commit(4)	This commits all four of the following: • The E *Atom element* generated for the branch at 0x1000. • The N *Atom element* generated for the branch at 0x200C. • The E *Atom element* generated for the branch at 0x2014. • The E *Atom element* generated for the branch at 0x1008.

D14.1 Transactions

D14.1.1 Simple successful transaction

This is an example of a successful transaction traced by a trace unit with no speculation in the trace element stream.

Table D14.3: Example of trace with a successful transaction

	Execution	Trace elements	Notes
0x1000	B -> 0x2000	Trace_info(...)	
		Trace_on()	
		Target_address(0x2000)	Trace on
0x2000	TSTART	Atom(E)	
		Transaction_start()	The transaction starts.
0x2004	B -> 0x2400	Atom(E)	
{...}			
0x2804	B -> 0x2808	Atom(E)	
0x2808	TCOMMIT	Transaction_commit()	Transaction finishes.
{...}			

D14.1.2 Simple Failed Transaction example

This is an example of a failed transaction traced by a trace unit with no speculation in the trace element stream.

Table D14.4: Example of trace with a transaction failure

	Execution	Trace elements	Notes
0x2000	TSTART	Trace_on() Target_address(0x2000) Atom(E) Transaction_start()	Trace on The transaction starts
0x2004	TST		
0x2008	BEQ	Atom(N)	
{...}			
0x2804	B -> 0x3000	Atom(E) Target_address(0x3000)	
	Transaction fails	Transaction_failure()	Transaction Fails
0x2004	TST	Target_address(0x2004)	This address is where execution resumes after the transaction failure.
0x2008	BEQ	Atom(E)	
{...}			

D14.1.3 Canceled Transaction failure example

Table D14.5: Example of trace with a failed transaction

		Execution	Trace elements	Notes
0x2000	TSTART		Trace_on() Target_address(0x2000) Atom(E) Transaction_start()	Trace on The transaction starts
0x2004	TST			
0x2008	BEQ		Atom(N)	
{...}				
0x2804	B -> 0x3000		Atom(E)	
	Transaction fails		Target_address(0x3000) Cancel(2) Transaction_failure()	Transaction fails
0x2004	TST		Target_address(0x2004)	
0x2008	BEQ		Atom(E)	
{...}				

D14.1.4 Speculated Transaction example

Table D14.6: Example of trace with a transaction failure

	Execution	Trace elements	Notes
{…}			
0x1000	B -> 0x2000	Atom(E)	This branch is speculatively taken, but was incorrectly speculated and will be corrected later.
0x2000	TSTART	Atom(E) Transaction_start()	The transaction starts.
0x2004	TST		
0x2008	BEQ	Atom(N)	
{…}			
0x2804	B -> 0x3000	Atom(E)	
		Cancel(4) Mispredict	The transaction was only speculatively started. It is optional if a *Transaction Failure element* is traced, because the *Cancel element* cancels the *Transaction Start element*. The *Mispredict element* corrects the incorrectly speculated first branch.
0x1004	{…}		
{…}			

Chapter D15
Pseudocode

D15.1 ETE element ASL

D15.1.1 Atom enumeration

```
// Atom
// ====
// Atom enum. Atoms are either E (taken) or N (not taken).
enumeration Atom {
    Atom_E,
    Atom_N
};
```

D15.1.2 AtomElement()

```
// AtomElement()
// =============
// Generates an Atom element based on the given atom.
Element AtomElement(Atom t)
    Element a;

    a.kind = ELEM_ATOM;
    a.debug_id = GetNextDebugId();
```

```
        a.payload.atom_type = t;


        LogElem(ELEMENT, a, if t == Atom_E then "E" else "N");
        return a;
```

D15.1.3 QElement()

```
// QElement()
// =========
// Generates a Q element based on the number of elements
// to resolve.
Element QElement(integer count)
    Element a;


    a.kind = ELEM_Q;
    a.debug_id = GetNextDebugId();
    a.payload.count = count;


    LogElem(ELEMENT, a, "count " ++ a.payload.count);
    return a;
```

D15.1.4 CancelElement()

```
// CancelElement()
// ===============
// Generates a Cancel element based on a given number
// of elements to cancel.
Element CancelElement(integer count)
    Element a;


    a.kind = ELEM_CANCEL;
    a.debug_id = GetNextDebugId();
    a.payload.count = count;


    LogElem(ELEMENT, a, "cancel " ++ count);
    return a;
```

D15.1.5 CommitElement()

```
// CommitElement()
// ===============
// Generates a commit element based on the
// number of elements to commit.
Element CommitElement(integer count)
    Element a;
```

```
            a.kind = ELEM_COMMIT;
            a.debug_id = GetNextDebugId();
            a.payload.count = count;


            LogElem(ELEMENT, a, "commit " ++ count);
            return a;
```

D15.1.6 ContextElement()

```
    // ContextElement()
    // ================
    // Generates a context element based on context ID, VMID,
    // Exception level, Security state and AArch32/64 state.
    Element ContextElement(bits(32) context_id,
                           bits(32) vmid,
                           bits(2) exception_level,
                           SecurityLevel secure,
                           boolean sixty_four_bit)
        Element a;


        a.kind = ELEM_CONTEXT;
        a.debug_id = GetNextDebugId();
        a.payload.context_id = context_id;
        a.payload.vmid = vmid;
        a.payload.exception_level = exception_level;
        a.payload.security = secure;
        a.payload.sixty_four_bit = sixty_four_bit;


        LogElem(ELEMENT, a,
            "c_id " ++ context_id ++
            ", vmid " ++ vmid ++
            ", ex_lvl " ++ a.payload.exception_level ++
            ", secure " ++ (a.payload.security == SecurityLevel_SECURE) ++
            ", 64_bit " ++ a.payload.sixty_four_bit);
        return a;
```

D15.1.7 CycleCountElement()

```
    // CycleCountElement()
    // ===================
    // Generates a Cycle Count element based on a number of cycles.
    Element CycleCountElement(integer count)
        Element a;


        a.kind = ELEM_CYCLE_COUNT;
        a.debug_id = GetNextDebugId();
        a.payload.count = count;


        LogElem(ELEMENT, a, "count " ++ a.payload.count);
```

```
        return a;
```

D15.1.8 DiscardElement()

```
// DiscardElement()
// ================
// Generates a Discard element.
Element DiscardElement()
    Element a;


    a.kind = ELEM_DISCARD;
    a.debug_id = GetNextDebugId();


    LogDecompressor(ELEMENT, " -
n");
    return a;
```

D15.1.9 ExceptionElement()

```
// ExceptionElement()
// ==================
// Generates an Exception element based on the address to branch
// to and the type of exception.
Element ExceptionElement(integer exception_type, bits(64) address)
    Element a;


    a.kind = ELEM_EXCEPTION;
    a.debug_id = GetNextDebugId();
    a.payload.exception_type = exception_type;
    a.payload.address = address;


    LogElem(ELEMENT, a, "ex_type " ++ exception_type ++ " addr " ++ address);
    return a;
```

D15.1.10 EventElement()

```
// EventElement()
// ==============
// Generates an Event element based on the number of the event that fired.
Element EventElement(integer idx)
    Element a;


    a.kind = ELEM_EVENT;
    a.debug_id = GetNextDebugId();
    a.payload.event_id = idx;
```

```
            LogElem(ELEMENT, a, " id " ++ a.payload.event_id);
            return a;
```

D15.1.11 MispredictElement()

```
// MispredictElement()
// ==================
// Generates a Mispredict element.
Element MispredictElement()
    Element a;

    a.kind = ELEM_MISPREDICT;
    a.debug_id = GetNextDebugId();

    LogDecompressor(ELEMENT, "MISPREDICT");
    return a;
```

D15.1.12 OverflowElement()

```
// OverflowElement()
// =================
// Generates an Overflow element.
Element OverflowElement()
    Element a;

    a.kind = ELEM_OVERFLOW;
    a.debug_id = GetNextDebugId();

    LogElem(ELEMENT, a, "-");
    return a;
```

D15.1.13 TimestampElement()

```
// TimestampElement()
// ==================
// Generates a Timestamp element based on a timestamp value
// and a cycle count value.
Element TimestampElement(integer timestamp, integer cycles)
    Element a;

    a.kind = ELEM_TIMESTAMP;
    a.debug_id = GetNextDebugId();
    a.payload.timestamp = timestamp;
    a.payload.cycle_count = cycles;

    LogElem(ELEMENT, a,
```

```
                    "timestamp " ++ timestamp ++
                    " cycles " ++ cycles);
            return a;
```

D15.1.14 TraceInfoElement()

```
// TraceInfoElement()
// ==================
// Generates a Trace Info element based on cycle counting parameters,
// speculation depth, and transaction status.
Element TraceInfoElement(boolean cc_enabled,
                         integer cc_threshold,
                         integer current_spec_depth,
                         boolean in_transaction)
    Element a;

    a.kind = ELEM_TRACE_INFO;
    a.debug_id = GetNextDebugId();
    a.payload.cc_enabled = cc_enabled;
    a.payload.cc_threshold = cc_threshold;
    a.payload.current_spec_depth = current_spec_depth;
    a.payload.in_transaction = in_transaction;

    LogElem(ELEMENT, a,
            "cc_enabled " ++ a.payload.cc_enabled ++
            " cc_threshold " ++ a.payload.cc_threshold ++
            " current_spec_depth " ++ a.payload.current_spec_depth);
    return a;
```

D15.1.15 TraceOnElement()

```
// TraceOnElement()
// ===============
// Generates a Trace On element.
Element TraceOnElement()
    Element a;

    a.kind = ELEM_TRACE_ON;
    a.debug_id = GetNextDebugId();

    LogDecompressor(ELEMENT, "TRACE_ON");
    return a;
```

D15.1.16 TargetAddressElement()

```
// TargetAddressElement()
// =====================
```

```
// Generates a Target Address element based on a given
// address and sub_isa.
Element TargetAddressElement(AddressHistoryBufferEntry reg)
    Element a;
    a.kind = ELEM_TARGET_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;
    a.debug_id = GetNextDebugId();

    LogElem(ELEMENT, a, "" ++ a.payload.address ++ " IS " ++
        (if reg.sub_isa == IS0 then "IS0" else "IS1"));
    return a;
```

D15.1.17 SourceAddressElement()

```
// SourceAddressElement()
// =====================
// Generates a Source Address element based on an instruction's address
// and sub_isa.
Element SourceAddressElement(AddressHistoryBufferEntry reg)
    Element a;

    a.kind = ELEM_SOURCE_ADDRESS;
    a.payload.address = reg.address;
    a.payload.sub_isa = reg.sub_isa;
    a.debug_id = GetNextDebugId();

    return a;
```

D15.1.18 TransactionStartElement()

```
// TransactionStartElement()
// ========================
Element TransactionStartElement()
    Element a;

    a.kind = ELEM_TRANS_START;
    return a;
```

D15.1.19 TransactionCommitElement()

```
// TransactionCommitElement()
// =========================
Element TransactionCommitElement()
    Element a;
```

```
        a.kind = ELEM_TRANS_COMMIT;
        return a;
```

D15.1.20 TransactionFailureElement()

```
// TransactionFailureElement()
// ==========================
Element TransactionFailureElement()
    Element a;

    a.kind = ELEM_TRANS_FAILURE;
    return a;
```

D15.2 ETE decompressor enumerations

D15.2.1 SubISA enumeration

```
// SubISA
// ======
// Represents sub instruction set.
// IS0 = AArch32 or AArch64, IS1 = AArch32 Thumb
enumeration SubISA {
    IS0,
    IS1
};
```

D15.2.2 SynchronisationState enumeration

```
// States to represent synchronisation of the reconstructor, state
// transitions as follows:
//  _____
// | Init State |      Input        | Final State |
// |_____|_____|_____|
// | NOT_SYNC   | context element   | CONTEXT     |
// | NOT_SYNC   | address element   | ADDRESS     |
// | ADDRESS    | context element   | FULL_SYNC   |
// | ADDRESS    | overflow element  | NOT_SYNC    |
// | ADDRESS    | discard element   | NOT_SYNC    |
// | ADDRESS    | trace on element  | NOT_SYNC    |
// | ADDRESS    | atom element      | NOT_SYNC    |
// | ADDRESS    | exception element | NOT_SYNC    |
// | CONTEXT    | address element   | FULL_SYNC   |
// | CONTEXT    | overflow element  | NOT_SYNC    |
// | CONTEXT    | discard element   | NOT_SYNC    |
// | CONTEXT    | trace on element  | NOT_SYNC    |
// | FULL_SYNC  | indirect branch   | CONTEXT     |
// | FULL_SYNC  | discard element   | NOT_SYNC    |
// | FULL_SYNC  | overflow element  | NOT_SYNC    |
// | FULL_SYNC  | trace on element  | NOT_SYNC    |
// |_____|_____|_____|


enumeration SynchronisationState {
    NOT_SYNC_STATE,         // Not syncing, need sync
    CONTEXT_STATE,          // Have context, need address
    ADDRESS_STATE,          // Have address, need context
    FULL_SYNC_STATE         // Fully synced
};
```

D15.2.3 InstType enumeration

```
// InstType
// ========
// Instruction type. Cannot use BranchType as this does not cover other P0
```

```
// non-branching instructions (WFE/WFI, ISB).
// WFX counts as 'Other' if it is not a P0 element (see TRCIDR2.WFXMODE).


enumeration InstType {
    InstType_BRANCH_DIR,        // Direct branch
    InstType_BRANCH_INDIR,      // Indirect branch
    InstType_WFX,               // WFI/WFE instruction
    InstType_ISB,               // Instruction barrier
    InstType_OTHER              // Non-P0 instructions
};
```

D15.3 ETE decompressor functions

D15.3.1 EndOfStream()

```
// EndOfStream()
// =============
// Returns TRUE iff all the data in the stream have been consumed.
boolean EndOfStream(bits(S) stream);
```

D15.3.2 ReservedEncoding()

```
// ReservedEncoding()
// ==================
// The trace byte stream is not compliant to the protocol. The trace analyzer
// has to stop.
ReservedEncoding();
```

D15.3.3 ReadAndConsume()

```
// ReadAndConsume()
// ================
// Reads the next N bits from the trace byte stream and returns them, also
// updating the trace byte stream pointer.
bits(N) ReadAndConsume(integer N, bits(S) stream);
```

D15.3.4 LogDecompressor()

```
// Instrumentation functions
// =========================


LogDecompressor(Decomp_Level lvl, string details);
LogElem(Decomp_Level lvl, Element e, string details);
integer GetNextDebugId();
ERROR(string msg);
LogReturnStack();
PrintElement(Element e);
string ExcepTypeToStr(integer type_val);
```

D15.3.5 SBZ()

```
// SBZ()
// =====
// Raise an error if the field B is not zero.
SBZ(bits(N) B);
```

D15.3.6   ResolutionQueue

```
// ResolutionQueue.Initialize()
// ============================
// If decompression starts at a Trace Info element that has a non-zero
// speculation depth, the trace analyzer must wait until the speculation
// of these unseen P0 elements has been resolved.
//
// Set the number of unseen P0 elements that are outstanding that need to be
// resolved.
ResolutionQueue.Initialize(integer i);


// ResolutionQueue.Uninitialized()
// ===============================
// Returns TRUE if the resolution queue is uninitialized.
boolean ResolutionQueue.Uninitialized();


// ResolutionQueue.Aligned()
// =========================
// Returns TRUE if all the unseen P0 elements have been resolved.
boolean ResolutionQueue.Aligned();


// ResolutionQueue.Align()
// =======================
// Mark the oldest oldest unseen P0 element as resolved.
ResolutionQueue.Align();


// ResolutionQueue.Length()
// ========================
// Returns the number of elements in the queue.
integer ResolutionQueue.Length();


// ResolutionQueue.PopBack()
// =========================
// Discards the element at the back (youngest) of the queue.
ResolutionQueue.PopBack();


// ResolutionQueue.Back()
// ======================
// Returns the element at the back (youngest) of the queue.
Element ResolutionQueue.Back();


// ResolutionQueue.PopFront()
// ==========================
// Removes the element at the front (oldest) from the queue.
ResolutionQueue.PopFront();


// ResolutionQueue.Front()
// =======================
// Returns the element at the front (oldest) of the queue.
```

```
Element ResolutionQueue.Front();


// ResolutionQueue.CommitFront()
// =============================
// Commits the element at the front of the queue.
ResolutionQueue.CommitFront()


// ResolutionQueue.Push()
// ======================
// Add element e to the back of the queue.
ResolutionQueue.Push(Element e);
```

D15.3.7 TransactionQueue

```
// TransactionQueue.Length()
// =========================
// Return the number of entries in the transaction queue.
integer TransactionQueue.Length();


// TransactionQueue.FrontPop()
// ===========================
// Remove the first entry in the transaction queue.
TransactionQueue.FrontPop();


// TransactionQueue.Front()
// ========================
// Return the element at the front of the transaction queue.
Element TransactionQueue.Front();


// TransactionQueue.Push()
// =======================
// Add an element to the back of the transaction queue.
TransactionQueue.Push(Element e);


// TransactionQueue.InTransaction()
// ================================
// Are we currently in a transaction?
boolean TransactionQueue.InTransaction();


// TransactionQueue.StartTransaction()
// ===================================
// Enter a transaction.
TransactionQueue.StartTransaction();


// TransactionQueue.EndTransaction()
// =================================
// Leave a transaction.
TransactionQueue.EndTransaction();
```

D15.3.8 ReturnStack

```
// ReturnStack.Reset()
// ==================
// Resets the return stack.
ReturnStack.Reset();


// ReturnStack.Push(bits(64) addr, SubISA sub_isa)
// ===============================================
// Pushes onto the return stack.
ReturnStack.Push(bits(64) addr, SubISA sub_isa);


// ReturnStack.Pop()
// =================
// Pops the top of the return stack.
ReturnStackEntry ReturnStack.Pop();


// ReturnStack.IsEmpty()
// =====================
// Returns TRUE iff the return stack is empty.
boolean ReturnStack.IsEmpty();
```

D15.3.9 AddressHistoryBufferEntry

```
// AddressHistoryBufferEntry
// =========================
// An entry in the address history buffer.
type AddressHistoryBufferEntry is (
    bits(64) address,
    SubISA sub_isa
)


AddressHistoryBufferEntry UNKNOWN_ADDRESS;
```

D15.3.10 AddressHistoryBuffer

```
// AddressHistoryBuffer.Reset()
// ============================
// Resets the address history buffer.
AddressHistoryBuffer.Reset();


// AddressHistoryBuffer.Add()
// ==========================
// Adds an address to the address history buffer.
AddressHistoryBuffer.Add(AddressHistoryBufferEntry entry);


// AddressHistoryBuffer.Add()
// ==========================
```

```
// Adds an address to the address history buffer.
AddressHistoryBuffer.Add(bits(64) address, SubISA sub_isa);


// AddressHistoryBuffer.Get()
// =========================
// Returns the given entry from the address history buffer.
AddressHistoryBufferEntry AddressHistoryBuffer.Get(integer n);
```

D15.3.11 ProgramImage

```
// ProgramImage.DecodeNextInst()
// ============================
// Returns the decoded next instruction in the program image.
DecodedInst ProgramImage.DecodeNextInst();


// ProgramImageDecodeAvilable()
// ===========================
// Returns TRUE iff we are currently inside the program image.
boolean ProgramImage.DecodeAvailable();
```

D15.3.12 ExceptionWithUnknownAddress()

```
// ExceptionWithUnknownAddress()
// ============================
// Does this exception type have an unknown
// prefered exception return address.

boolean ExceptionWithUnknownAddress(Element e)
    case e.payload.exception_type<4:0> of
        when '00000', '11001'
            return TRUE;
        when '11000'
            ERROR("Transation Failure Element");
        otherwise
            return FALSE;
```

D15.4 ETE data encodings

D15.4.1 POD()

```
// POD()
// =====
// Return data from stream in Plain Old Data Little Endian format.
bits(N) POD(integer N, bits(S) stream)
    return ReadAndConsume(N, stream);
```

D15.4.2 ULEB128()

```
// ULEB128()
// =========
// Gets N bits of continuable data from the stream.
bits(N) ULEB128(bits(S) stream)
    return BitReplacement(stream, Zeros(N));
```

D15.4.3 BitReplacement()

```
// BitReplacement()
// ================
// Gets N bits of continuable, bit replacement data from the stream.
bits(N) BitReplacement(bits(S) stream, bits(N) original)
    R = original;

    I = 0;
    bits(8) BYTE;
    repeat
        BYTE = ReadAndConsume(8, stream);
        R<I+6:I> = BYTE<6:0>;
        I = I + 7;
    until BYTE<7> == '0' || I >= N - 8;

    if BYTE<7> == '1' then
        BYTE = ReadAndConsume(8, stream);
        R<I+7:I> = BYTE;
        end = N MOD 7;
        if end == 0 then end = 7;
        if I + 8 > N then SBZ(BYTE<7:end>);

    return R;
```

D15.5 Common functions

D15.5.1 Replicate()

```
// Replicate()
// ===========
// Replicates the bitstring x, N times.
bits(M*N) Replicate(bits(M) x, integer N);
```

D15.5.2 Zeros()

```
// Zeros()
// =======
// Returns a zero bitstring of length n.
bits(n) Zeros(integer n);
```

Chapter D16
Functional Differences from ETMv4

ETE has a considerable overlap with the ETMv4 architecture *Arm® Embedded Trace Macrocell Architecture Specification ETMv4* [3], with the intent that broadly unified software stack can program a trace unit and interpret the trace stream from either an ETMv4 trace unit or an ETE trace unit.

This section describes the primary functional differences between ETMv4 and ETE.

- Removal of data trace documentation, since this is not permitted in A-profile.
- Removal of conditional non-branch documentation, since this is not permitted in A-profile.
- TRCDEVARCH.PRESENT == 1 is mandatory.
- TRCDEVARCH.ARCHVER and TRCDEVARCH.REVISION take new values.
- TRCIDR1.TRCARCHMAJ and TRCIDR1.TRCARCHMIN take new values.
- TRCIDR9 is fixed at zero.
- Context identifier tracing is mandatory, defined in TRCIDR2.CIDSIZE.
- Virtual context identifier tracing is mandatory when the *Processing Element* (PE) implements EL2, defined in TRCIDR2.VMIDSIZE.
- The Virtual context identifier is always based on CONTEXTIDR_EL2, with support for tracing VTTBR_EL2.VMID removed.
- 64-bit timestamp is the only supported timestamp size.
- Timestamping is mandatory in ETE.
- TRCIDR2.IASIZE is only permitted to indicate a 64-bit instruction address size.
- External Inputs are unified with the PMU event space, with new TRCEXTINSELR<n> registers introduced.
- TRCIDR5.NUMEXTIN indicates the unified External Input model.
- Added TRCRSR.EXTIN for reading and setting the External Input Selectors state.
- Added TRCRSR.EVENT for reading and setting the ETEEvent state.
- Added TRCRSR.TA for reading and setting whether tracing was active.
- Changed requirements for the tracing of Exceptions to be dependent on the new TRCRSR.TA field.
- Removal of memory-mapped accesses. This was deprecated in ETMv4.4 for Armv8-A.

- Removal of trace unit sharing.
- Added a requirement that trace must be output within finite time.
- Added a requirement that the trace unit resources are paused when entering a Trace Prohibited region.
- Added a bit to TRCSSCSR<n> to indicate that the Single-shot Comparator Control fired while the resources are paused.
- Added requirements for dependencies on the `TSB CSYNC` instruction.
- Execution of `TSB CSYNC` instruction requests a timestamp element.
- The Unified Power Domain Model from ETMv4.5 for Armv8-A is mandatory in ETE.
- Changes to the enable and disable code sequences.
- Addition of the tracing of Transactional state.
- Tightened the requirements for obeying the order of start point and stop points for the ViewInst start/stop function, and tightened the rules for programming the start/stop function.
- Addition of *Source Address elements*.
- Added rules to require no trace to be generated in Trace Prohibited regions, under some circumstances.
- Added constraints for the effect of system instructions causing the trace unit to become enabled or disabled.
- Additional constraints for the forced tracing of exceptions around Trace Prohibited regions, to ensure trace is not generated in Trace Prohibited regions.
- Removed the flexibility around tracing of an *Exceptional occurrence* immediately after a Trace Prohibited region or when trace generation becomes operative. Such *Exceptional occurrences* are not traced.
- Added a requirement that the resource operations must complete before a TSB instruction completes.
- Defined the behavior of the visibility of reads and writes to trace unit registers from system instructions, external debugger and by the trace unit.
- Changed branch broadcasting to be required in all implementations, see TRCIDR0.TRCBB.
- TRCSYNCPR is read/write in all implementations, see TRCIDR3.SYNCPR.
- Forced tracing of System Error exceptions is required in all implementations, see TRCIDR3.TRCERR.
- Changed cycle counting to be required in all implementations, see TRCIDR0.TRCCCI.
- Removed the trace unit OS Lock mechanism, and changed to require the PE OS Lock to affect the trace unit.
- Removed the Exception Return element and Exception Return packet.
- Constrained TRCCLAIMCLR and TRCCLAIMSET to not require explicit synchronization.
- Added more constraints to the operation of the Single-shot Comparator Controls when the trace unit becomes disabled, or when entering a Trace Prohibited region.
- Added more constraints to the operation of the ViewInst start/stop function when the trace unit becomes disabled, or when entering a Trace Prohibited region.
- Constrained the behavior of cycle counting after a trace unit buffer overflow, to require the cycle count to be traced as UNKNOWN on the first *Cycle Count element* after an overflow.
- Export of PMU events to the trace unit is not affected by PMCR.X or PMCR_EL0.X.
- *Event elements* are permitted to be generated before the first *Trace Info element*.

Part E
The Trace Buffer Extension

Chapter E1
Trace Buffer Extension

E1.1 Description

E1.1.1 About the Trace Buffer Extension

I$_{\text{LMNBQ}}$ In an Armv8 processor with CoreSight, trace generated by a trace unit is routed over a CoreSight trace fabric (AMBA ATB) through a series of trace funnels, replicators, and so on, to one or more trace sinks. The CoreSight *Trace Memory Controller* (TMC) is an example of a trace sink that can take various forms, one of which is an *Embedded Trace Router* (ETR). The ETR writes formatted trace to a buffer in memory.

I$_{\text{MRFPK}}$ The Trace Buffer Extension feature is identified as FEAT_TRBE.

When FEAT_TRBE is implemented, the *Processing Element* (PE) includes a *Trace Buffer Unit*. There is one logical Trace Buffer Unit for each PE in the processor.

When the Trace Buffer Unit is enabled, program-flow trace generated by the trace unit is written directly to memory by the Trace Buffer Unit, rather than routing it to a trace fabric. Figure E1.1 shows this.
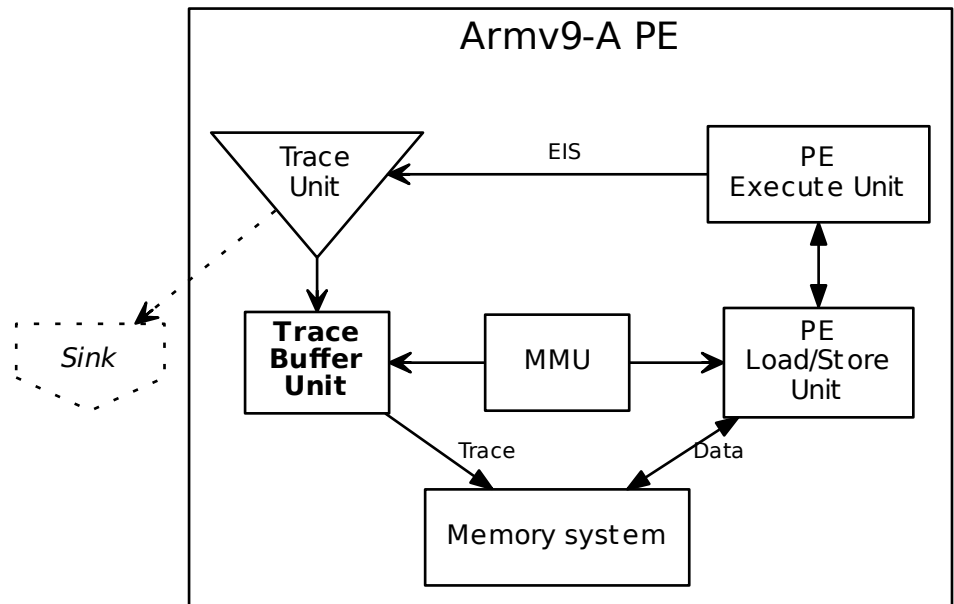
Figure E1.1: Logical organization of an Armv9-A PE including a trace unit and a Trace Buffer Unit

In this figure:

- *EIS* is an internal representation of the executed instruction stream.

- The *trace unit* converts the EIS into formatted trace data.

- *Sink* is described by the section E1.2.1.7 *Trace Buffer Unit disabled*.

X_{RHRGC} For use by self-hosted software in a platform Operating System environment, a trace buffer manager such as a Trace Buffer Unit or ETR must support a trace buffer that is mapped to a set of non-contigous physical blocks in memory. The Trace Buffer Unit achieves this using the PE VMSA-based MMU, meaning it cooperates well with other software.

This means:

- The trace buffer is normally virtually addressed.

- The trace buffer has an owning Exception level and owning Security state that define the translation regime the trace buffer uses.

- FEAT_TRBE provides a synchronization instruction, TSB CSYNC, that is used with a DSB operation to flush trace to the trace buffer.

- Trace is implicitly prohibited when the owning translation regime is not in context. That is, trace is prohibited if executing at a higher Exception level than the owning Exception level, or not executing in the owning Security state. This is an addition to the Trace Extension.

However, the Trace Buffer Extension also allows the trace buffer to be defined using physical addresses. This allows the Trace Buffer Unit to be used for debugging software that changes the virtual address mappings. In this configuration, the buffer must be contiguously mapped in physical memory.

The extent of the trace buffer is defined by a Base pointer and a Limit pointer. The Base pointer and Limit pointer are at-least 4KB-aligned, meaning a buffer must be at least one full virtual page.

The Trace Buffer Unit supports two types of operational modes:

- The trace buffer mode controls how the Trace Buffer Unit uses the trace buffer.

• The trigger mode controls how the Trace Buffer Unit reacts to a trigger condition signaled by the trace unit.

E1.1.2 System events

I_{XDSQV} The trace unit can be configured to react to PE events and events from the *Cross-trigger Interface* (CTI). The CTI is for use by external debuggers.

As part of the Trace Buffer Extension and FEAT_ETE, the PMU and FEAT_ETE event sources are unified into a single event number space. Unless otherwise stated, all architecturally-defined Common events that can be counted by the PMU are usable as an event at the trace unit.

The following additional architecturally-defined events are provided:

• The CTI_TRIGOUT<n> events are defined to map the system events from the CTI into the PMU event number space. As well as defining these events for the trace unit, this also provides a standard mechanism for counting *external events passed to the PE*, as recommended by the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

• The TRCEXTOUT<n> events are defined to allow the PMU to count the events that a FEAT_ETE implementation of the trace unit might generate.

• The PMU_OVFS and PMU_HOVFS events are defined to allow the trace unit to trigger directly from a PMU overflow without using the Performance Monitors overflow trigger for PMU counters accessible to EL1 and EL0, and EL2, respectively.

• The TRB_WRAP event is defined to allow the trace unit to trigger when the current write pointer reaches the end of the trace buffer and is wrapped.

See also:

• Chapter D1 *Embedded Trace Extension*
• E1.3 *Events*

E1.1.3 Interrupts

I_{KQXVW} An interrupt request is raised on a buffer mangement event, such as an abort or the trace buffer filling. This is the trace buffer management interrupt.

The interrupt request is passed to an interrupt controller, such as a *Generic Interrupt Controller* (GIC).

Arm recommends this is a *Private Peripheral Interrupt* (PPI).

E1.2 Specification

R_{KYJWX} If FEAT_ETE is implemented, then FEAT_TRBE is implemented.

R_{KLNJV} If FEAT_TRBE is implemented, then a trace unit that implements FEAT_ETE is implemented.

R_{LPMRM} If FEAT_TRBE is implemented, then FEAT_TRF is implemented.

S_{QVGQX} The FEAT_TRBE feature is identified to software by ID_AA64DFR0_EL1.TraceBuffer.

R_{PLYXP} Other than where stated otherwise, this specification describes a simple sequential model of the Trace Buffer Unit. That is, one which performs the simple loop of:

1. Collect a single byte of trace data from the trace unit.

2. If required, performs an address translation for the address of the current write pointer to the physical address for the write pointer.

3. If permitted, write the byte of trace data to the write address.

4. If collection is not stopped, increment the current write pointer.

5. If necessary, decrement the Trigger Counter.

Trace buffer management events are processed as part of this operation loop.

Implementations compliant with the architecture conform with the described behavior of the Trace Buffer Unit. This specification is not intended to describe how to build an implementation of the Trace Buffer Unit, nor to limit the scope of such implementations beyond the defined behaviors.

Except where the architecture specifies differently, the programmer-visible behavior of an implementation that is compliant with this specification is the same as a simple sequential model. Trace appears to be written sequentially by the Trace Buffer Unit.

This specification also describes rules for software to use the Trace Buffer Unit.

E1.2.1 The trace buffer

R_{YCHKJ} If and only if all of the following are true, then the Trace Buffer Unit is *Enabled*:

- `SelfHostedTraceEnabled()` == TRUE.

- TRBLIMITR_EL1.E is 1.

The pseudocode function `TraceBufferEnabled` shows this.

R_{JYXPM} If the Trace Buffer Unit is not Enabled, then the Trace Buffer Unit is *Disabled*. See E1.2.1.7 *Trace Buffer Unit disabled*.

The pseudocode function `TraceBufferEnabled` shows this.

I_{SJMTV} `SelfHostedTraceEnabled()` is defined by FEAT_TRF in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

R_{BGLHT} If and only if all of the following are true, then the Trace Buffer Unit is *Running*:

- The Trace Buffer Unit is Enabled.

- TRBSR_EL1.S is 0.

The pseudocode function `TraceBufferRunning` shows this.

R$_{FRHXV}$ If and only if all of the following are true, then *Collection is stopped*:

- The Trace Buffer Unit is Enabled.

- TRBSR_EL1.S is 1.

The pseudocode function `TraceBufferRunning` shows this.

I$_{JYSQZ}$ While the Trace Buffer Unit is Enabled, it collects trace data from the trace unit and does one of the following:

- Accepts the trace data and writes it to the trace buffer in memory.

- Discards the trace data. The trace data is lost.

- Rejects the trace data.

R$_{YMYQX}$ When the Trace Buffer Unit is Enabled and Running, and the Trace Buffer Unit is able to accept the trace data, the Trace Buffer Unit *Accepts* the trace data from the trace unit and writes it into the *trace buffer*.

R$_{LNTVR}$ When the Trace Buffer Unit is Enabled and Running, and the Trace Buffer Unit is not able to accept the trace data, the Trace Buffer Unit *Rejects* the trace data from the trace unit. The trace data might be retained by the trace unit until the Trace Buffer Unit Accepts the trace data.

I$_{SQYCT}$ For example, the Trace Buffer Unit might not be able to accept trace data while its internal buffers are full.

I$_{TRCDR}$ If the Trace Buffer Unit Rejects trace data and the trace unit is not able to retain the trace data, then the trace unit discards it and enters an Overflow state. Details of Overflow state and how the trace unit recovers from Overflow state are defined by the trace unit.

R$_{YMVZL}$ When the Trace Buffer Unit is Enabled and Collection is stopped, the Trace Buffer Unit *Discards* trace data from the trace unit. The trace data is lost.

R$_{PHSKP}$ When used with a trace unit that implements FEAT_ETE, the Trace Buffer Unit ignores the value of the ETE TRCTRACEIDR register.

E1.2.1.1 The trace buffer pointers

R$_{WKBRT}$ The trace buffer is defined by three *trace buffer pointer addresses*:

- The *Base pointer*.

- The *Limit pointer*.

- The *current write pointer*.

R$_{FVPBS}$ The trace buffer starts at the Base pointer and extends to the Limit pointer. The location at the Base pointer is included in the trace buffer. The location at the Limit pointer is not included in the trace buffer.

R$_{XBLPK}$ The Base pointer and Limit pointer must be aligned by software to the smallest implemented translation granule size.

R$_{VHNTF}$ For each byte of trace the Trace Buffer Unit Accepts and writes to the trace buffer at the address in the current write pointer, one of the following applies:

- If the current write pointer is not equal to the Limit pointer minus one, then the current write pointer is incremented by one.

- If the current write pointer is equal to the Limit pointer minus one, then all of the following occur:

  - The current write pointer is *wrapped* by setting it to the Base pointer.
  - TRBSR_EL1.WRAP is set to 1.
  - The TRB_WRAP event is generated.

R$_{BGBCJ}$ The current write pointer is not incremented when Collection is stopped.

R$_{VMVJH}$ The required alignment of the current write pointer is IMPLEMENTATION DEFINED.

I_{BTSCF} The Trace Buffer Unit can write trace data to memory in quantized units. The behavior is as if the bytes are written sequentially.

R_{JMPCB} The Base pointer is (TRBBASER_EL1.BASE << 12). Bits [11:0] of the Base pointer are zero.

R_{LLBBS} The Limit pointer is (TRBLIMITR_EL1.LIMIT << 12). Bits [11:0] of the Limit pointer are zero.

R_{KXRTY} The current write pointer is TRBPTR_EL1.PTR[63:0].

R_{PBGNS} The Trigger Counter is TRBTRG_EL1.TRG.

E1.2.1.2 Address translation enabled

R_{XRNCQ} If TRBLIMITR_EL1.nVM is 0 then the Base pointer, Limit pointer, and current write pointer are virtual addresses in the stage 1 translation regime of the owning translation regime.

R_{CMDTG} If TRBLIMITR_EL1.nVM is 0, then the stage 1 translation process for translating a virtual addresses, and checking for MMU faults is identical to that for any other virtual address in the owning translation regime.

I_{GKBYK} If TRBLIMITR_EL1.nVM is 0, R_{CMDTG} means all of the following apply:

- The virtual addresses are translated to stage 1 output addresses by stage 1 translation, and checked for stage 1 MMU faults. The stage 1 output addresses are:

  – Physical address in the owning Security state if the owning translation regime has no stage 2 translation.

  – Intermediate physical addresses (IPAs) in the owning Security state if the owning translation regime has stage 2 translations.

- If stage 1 translation is enabled for the owning translation regime, the memory type, and, as applicable, Cacheability, Shareability, and Device type attributes, for stage 1 output addresses are defined by the translation table entries for the virtual address being written to.

- If stage 1 translation is disabled for the owning translation regime, the memory type of the stage 1 output addresses is Device-nGnRnE, unless overridden by stage 2 controls.

- If SCTLR_ELx.C is 0 for the owning translation regime and stage 1 translation is enabled then all accesses to Normal memory are Non-cacheable.

- TRBPTR_EL1[63:56] are ignored by address translation if the respective TBI bit is 1.

R_{SJFRQ} When the Trace Buffer Unit is Enabled, the Trace Buffer Unit might prefetch and cache address translations for the translation regime of the owning Exception level, including when the owning Exception level is out-of-context.

I_{QXJZX} R_{SJFRQ} means that, when the Trace Buffer Unit is enabled and the owning Exception level is a lower Exception level, then the Trace Buffer Unit might make memory accesses to translation table entries from the translation regime of the owning Exception level, using the settings of the System registers associated with that translation regime.

If the PE is not executing in the owning Security state, or the PE is executing at EL3 and SCR_EL3.NS does not indicate the owning Security state then the translation regime of the owning Exception level might not be the owning translation regime.

These memory accesses might be observed by other observers, to the extent that those accesses are required to be observed as determined by the shareability and cacheability of those translation table entries.

This is an exception to the rules in the section *Use of out-of-context translation regimes* of the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

See also G3.1 *Context switching*.

E1.2.1.3 Address translation disabled

R$_{\text{PBZRZ}}$ If TRBLIMITR_EL1.nVM is 1, the Base pointer, Limit pointer, and current write pointer are:

  • Physical address in the owning Security state if the owning translation regime has no stage 2 translation.

  • Intermediate physical addresses (IPAs) in the owning Security state if the owning translation regime has stage 2 translations.

These addresses are output directly by stage 1 without any address translation.

R$_{\text{FJKLW}}$ If TRBLIMITR_EL1.nVM is 1, TRBMAR_EL1 defines the memory type, and, as applicable, Cacheability, Shareability, and Device type attributes, for the stage 1 output addresses.

I$_{\text{GLNHS}}$ If TRBLIMITR_EL1.nVM is 1, the values of SCTLR_ELx.{C,M} for the owning translation regime are ignored for the purposes of determining the trace buffer Cacheability attributes.

S$_{\text{ZMPXW}}$ Locations are accessed with *mismatched attributes* if all accesses to the location do not use a common definition of attributes of that location. It is possible to generate mismatched attributes for a location by accessing that location using different translations, from different Observers, and so on.

If TRBLIMITR_EL1.nVM is 1 it is possible to generate mismatched attributes for a location from within the same stage 1 translation regime, using TRBMAR_EL1.

Software must be aware of the consequences of and permitted behaviors when accessing a memory location with mismatched attributes. For more information, including a full definition of *mismatched attributes* and the permitted behaviors, see the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

R$_{\text{MXRFD}}$ If TRBLIMITR_EL1.nVM is 1 and TRBPTR_EL1[top:PAMax()] is nonzero, a stage 1 Address Size fault is generated when the Trace Buffer Unit attempts to write to memory, where PAMax() is defined by the *Arm® Architecture Reference Manual, for A-profile architecture* [1], and:

  • If FEAT_LPA is implemented, top is 51.

  • If FEAT_LPA is not implemented, top is 47.

R$_{\text{BRRRK}}$ If TRBLIMITR_EL1.nVM is 1 and TRBPTR_EL1[63:(top+1)] is nonzero when the Trace Buffer Unit attempts to write to the trace buffer, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

  • A Stage 1 Address Size fault is generated.

  • TRBPTR_EL1[63:(top+1)] are ignored and treated as zero.

The value of top is as defined by R$_{\text{MXRFD}}$.

E1.2.1.4 Stage 2 translation

R$_{\text{JCMKS}}$ If the owning translation regime has stage 2 translations, the stage 2 process of translating the stage 1 output intermediate physical addresses and attributes to a physical address and attributes, and checking for MMU faults is identical to that for any other intermediate physical address generated by the owning translation regime.

I$_{\text{ZSDMR}}$ For example:

  • The intermediate physical addresses are translated to physical addresses by stage 2 translation, and checked for stage 2 MMU faults.

  • The attributes from stage 1 are combined with the attributes from the stage 2 translation to generate the physical memory attributes.

  • If the Effective value of HCR_EL2.DC in the owning translation regime is 1, then stage 1 translation is disabled and the memory type produced by stage 1 is Normal Non-shareable, Inner Write-Back Cacheable Read-Allocate Write-Allocate, Outer Write-Back Cacheable Read-Allocate Write-Allocate, regardless of the value of SCTLR_EL1.C.

- If the Effective value of HCR_EL2.MIOCNCE in the owning translation regime is 0, then for permitted accesses to a memory location that use a common definition of the Shareability and Cacheability of the location, there is no loss of coherency if the Inner Cacheability attribute for those accesses differs from the Outer Cacheability attribute.

E1.2.1.5 Accesses to the trace buffer

R_{JTVDD} Writes to the trace buffer by the Trace Buffer Unit are privileged writes within the owning translation regime.

I_{TTKZQ} The memory type, and, as applicable, Cacheability, Shareability, and Device type attributes, for accesses made by the Trace Buffer Unit are determined by the translation tables or TRBMAR_EL1.

See:

- R_{CMDTG} and I_{GKBYK}, if translation is enabled.
- R_{FJKLW}, if translation is disabled.
- R_{JCMKS} and I_{ZSDMR}, if the owning translation regime has stage 2 translations.

R_{FBKCC} It is IMPLEMENTATION DEFINED whether address translations performed by the Trace Buffer Unit manage dirty state and the Access flag. This is discoverable by software from TRBIDR_EL1.F.

R_{SHWSL} If hardware management of dirty state by the Trace Buffer Unit is implemented, and hardware management of dirty state is enabled for the owning translation regime, then the Trace Buffer Unit can speculatively update the translation table descriptor for any Page or Block in the trace buffer before writing data to it, if the write is otherwise permitted. This includes the case where a trace buffer management event means the Trace Buffer Unit stops writing data before the Page or Block is written to.

R_{BWNRF} The access granule for writes to the trace buffer by the Trace Buffer Unit is IMPLEMENTATION DEFINED, up to a maximum of 2KB, and might vary from time to time.

R_{CMSNC} Writes to any Device memory type by the Trace Buffer Unit occur once.

R_{RZTDD} A memory access from the Trace Buffer Unit that crosses a Page or Block boundary to a memory location that has a different memory type or Shareability attribute results in CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- Each memory access generated by the Trace Buffer Unit uses the memory type and Shareability attribute associated with its own address.

- The access generates an Alignment fault caused by the memory type:
  - If only the stage 1 translation generated the mismatch, or there is only one stage of translation in the owning translation regime, the resulting trace buffer management event is a stage 1 Data Abort.
  - If only the stage 2 translation generated the mismatch, the resulting trace buffer management event is a stage 2 Data Abort.
  - If both stages of translation generate the mismatch, the resulting trace buffer management event is either a stage 1 Data Abort or a stage 2 Data Abort.

- The trace data is discarded and the current write pointer might not be updated.

R_{XRQTN} A memory access from the Trace Buffer Unit to Device memory that crosses a boundary corresponding to the smallest translation granule size of the implementation causes CONSTRAINED UNPREDICTABLE behavior. In this case, the implementation performs one of the following behaviors:

- Each memory accesses generated by the Trace Buffer Unit is performed as if the boundary has no effect on the memory accesses.

- Each memory accesses generated by the Trace Buffer Unit is performed as if the boundary has no effect on the memory accesses except that there is no guarantee of ordering between it and other memory accesses.

- The access generates an Alignment fault caused by the memory type:

      – If only the stage 1 translation causes the boundary to be crossed, or there is only one stage of translation in the owning translation regime, the resulting trace buffer management event is a stage 1 Data Abort.

      – If only the stage 2 translation causes the boundary to be crossed, the resulting trace buffer management event is a stage 2 Data Abort.

      – If both stages of translation cause the boundary to be crossed, the resulting trace buffer management event is either a stage 1 Data Abort or a stage 2 Data Abort.

- The trace data is discarded and the current write pointer might not be updated.

Note

The boundary referred by R_{XRQTN} is between two Device memory regions that are both:

- Of the size of the smallest implemented translation granule.
- Aligned to the size of the smallest implemented translation granule.

I_{VKQBR} Although the Trace Buffer Unit behaves as if trace data is written a byte at a time, it is not required to do so.

For example, R_{BWNRF} and R_{CMSNC} mean that if the memory type for the trace buffer is Device-nGnRnE, then all of the following apply:

- Writes are not repeated and not re-ordered.

- A write Completes only after it reaches its endpoint in the memory system.

- The access granule size at the endpoint in the memory system is not defined by the architecture. However, a specific implementation might define the granule to permit interoperability with specific devices.

The access granule is not required to be fixed. For example, the Trace Buffer Unit might output a smaller granule when flushing trace data to the trace buffer.

See also I_{QQKZF}.

E1.2.1.6 The owning translation regime

R_{DPGJG} The *owning translation regime* is defined by the owning Security state and the owning Exception level.

R_{HBZNT} When the Trace Buffer Unit is Enabled, the *owning Security state* is:

- Non-secure state if and only if at least one of the following is true:
  - EL3 is not implemented and the PE executes in Non-secure state.
  - MDCR_EL3.NSTB is either 0b10 or 0b11.
- Secure state if and only if at least one of the following is true:
  - EL3 is not implemented and the PE executes in Secure state.
  - MDCR_EL3.NSTB is either 0b00 or 0b01.

R_{SKVWG} When the Trace Buffer Unit is Enabled, the *owning Exception level* is:

- EL1 if and only if at least one of the following is true:
  - EL2 is not implemented in the owning Security state.
  - EL2 is disabled in the owning Security state.
  - MDCR_EL2.E2TB is either 0b10 or 0b11.
- EL2 if and only if all of the following is true:
  - EL2 is implemented and enabled in the owning Security state.
  - MDCR_EL2.E2TB is 0b00.

R_{XWDZV} When the Trace Buffer Unit is Enabled and the owning Exception level is EL1, all of the following apply:

- The owning translation regime is EL1&0.

- If TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL1&0 translation regime using the current ASID from TTBRx_EL1.

- If TRBLIMITR_EL1.nVM is 1, the trace buffer pointer addresses are intermediate physical addresses.

- Intermediate physical addresses (whether from the output of stage 1, or the pointers, as applicable) are subject to stage 2 translation using the current VMID if EL2 is implemented and enabled and HCR_EL2.VM is 1.

- The following are prohibited trace regions:

  - EL3.
  - EL2.
  - EL0, if EL2 is implemented and enabled and HCR_EL2.TGE is 1.

R_{SHXTV} When the Trace Buffer Unit is Enabled and the owning Exception level is EL2, all of the following apply:

- If HCR_EL2.E2H is 0, the owning translation regime is EL2.

- If HCR_EL2.E2H is 1, the owning translation regime is EL2&0.

- If HCR_EL2.E2H is 0 and TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL2 translation regime.

- If HCR_EL2.E2H is 1 and TRBLIMITR_EL1.nVM is 0, the trace buffer pointer addresses are virtual addresses in the EL2&0 translation regime using the current ASID from TTBRx_EL2.

- If TRBLIMITR_EL1.nVM is 1, the trace buffer pointer addresses are physical addresses.

- EL3 is a prohibited trace region.

I_{QCKVZ} The following table summarizes the owning translation regime.

In this table:

Enabled is the value of the function `TraceBufferEnabled()`.

NSTB is the Effective value of MDCR_EL3.NSTB.

E2TB is the Effective value of MDCR_EL2.E2TB.

EEL2 is the Effective value of SCR_EL3.EEL2.

E2H is the Effective value of HCR_EL2.E2H.

The pseudocode function `TraceBufferOwner` shows this.

Enabled	NSTB	E2TB	EEL2	E2H	Owning translation regime
FALSE	X	X	X	X	Disabled
TRUE	0b0X	X	0b0	X	Secure EL1&0
TRUE	0b0X	0b00	0b1	0b0	Secure EL2
TRUE	0b0X	0b00	0b1	0b1	Secure EL2&0
TRUE	0b0X	0b1X	0b1	X	Secure EL1&0
TRUE	0b1X	0b00	X	0b0	Non-secure EL2
TRUE	0b1X	0b00	X	0b1	Non-secure EL2&0
TRUE	0b1X	0b1X	X	X	Non-secure EL1&0

R_{RRCNN} When any of the following is true, then the translation of addresses generated by the Trace Buffer Unit is CONSTRAINED UNPREDICTABLE:

- The owning Security state is Secure and SCR_EL3.NS is 1.
- The owning Security state is Non-secure and SCR_EL3.NS is 0.

The PE behaves as if one of the following is true for these translations:

- The owning Security state is Secure and SCR_EL3.NS is 0.
- The owning Security state is Non-secure and SCR_EL3.NS is 1.

Note: The behavior might differ within the same translation.

I_{MJMWG} Secure and Non-secure translation regimes have different behaviors. Non-secure translation regimes only operate on Non-secure addresses, but a Secure stage 1 translation regime can generate both Secure and Non-secure output addresses and a Secure stage 2 translation regime can have both Secure and Non-secure input and output addresses.

In addition, stage 2 translation is disabled when EL2 is disabled in the current Security state.

R_{RRCNN} means that if software executing at EL3 changes the value of SCR_EL3.NS before ensuring all Trace operations are Complete, this might cause CONSTRAINED UNPREDICTABLE behaviors, including any of the following:

- The owning Security state is Non-secure but the PE can generate Secure output addresses at both stage 1 and, if applicable, stage 2.
- The owning Security state is Non-secure EL1&0 but the PE behaves as if stage 2 is disabled because, in this example, Secure EL2 is disabled.
- The owning Security state is Secure but the PE treats Secure output addresses as Non-secure addresses at both stage 1 and, if applicable, stage 2.
- The owning Security state is Secure EL1&0 but the PE behaves as if stage 2 is enabled even if Secure EL2 is disabled.

See also R_{NSFRQ} and R_{MRVPT}.

R_{MFFGX} When the Trace Buffer Unit is Enabled and the owning Security state is Non-secure state, Secure state is a prohibited trace region.

R_{VGWJN} When the Trace Buffer Unit is Enabled and the owning Security state is Secure state, Non-secure state is a prohibited trace region.

I_{DCRYN} The Self-hosted trace extension, FEAT_TRF, provides additional controls to define Trace Prohibited regions.

FEAT_TRF is defined in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

The following table summarizes the Trace Prohibited regions, by Exception level and state, when all of the following apply:

- `TraceBufferEnabled()` == TRUE.

- EL3, Non-secure EL2 and Secure EL2 are all implemented.

- EL3 is using AArch64.

In this table:

NS   is the Effective value of SCR_EL3.NS.

STE   is the Effective value of MDCR_EL3.STE.

NSTB   is the Effective value of MDCR_EL3.NSTB.

E2TB   is the Effective value of MDCR_EL2.E2TB.

EEL2   is the Effective value of SCR_EL3.EEL2.

TGE   is the Effective value of HCR_EL2.TGE.

The EL3, EL2, EL1, EL0 columns show which control, if any, enables tracing at the Exception level. In these columns:

P means tracing is prohibited.

E2TRE means tracing is allowed if TRFCR_EL2.E2TRE is 1 and prohibited otherwise.

E1TRE means tracing is allowed if TRFCR_EL1.E1TRE is 1 and prohibited otherwise.

E0HTRE means tracing is allowed if TRFCR_EL2.E0HTRE is 1 and prohibited otherwise.

E0TRE means tracing is allowed if TRFCR_EL1.E0TRE is 1 and prohibited otherwise.

The pseudocode function `TraceAllowed` shows this.

NS	STE	NSTB	E2TB	EEL2	TGE	EL3	EL2	EL1	EL0
0b1	X	0b0X	X	X	X	P	P	P	P
0b1	X	0b1X	0b1X	X	0b0	P	P	E1TRE	E0TRE
0b1	X	0b1X	0b1X	X	0b1	P	P	n/a	P
0b1	X	0b1X	0b00	X	0b0	P	E2TRE	E1TRE	E0TRE
0b1	X	0b1X	0b00	X	0b1	P	E2TRE	n/a	E0HTRE
0b0	0b0	X	X	X	X	P	P	P	P
0b0	0b1	0b1X	X	X	X	P	P	P	P
0b0	0b1	0b0X	X	0b0	X	P	n/a	E1TRE	E0TRE
0b0	0b1	0b0X	0b1X	0b1	0b0	P	P	E1TRE	E0TRE
0b0	0b1	0b0X	0b1X	0b1	0b1	P	P	n/a	P
0b0	0b1	0b0X	0b00	0b1	0b0	P	E2TRE	E1TRE	E0TRE
0b0	0b1	0b0X	0b00	0b1	0b1	P	E2TRE	n/a	E0HTRE

R_{MCYDC} When the Trace Buffer Unit is Disabled, the owning translation regime, owning Security state, and owning Exception level are not defined.

E1.2.1.7 Trace Buffer Unit disabled

R_{HNTLG} When the Trace Buffer Unit is Disabled, the Trace Buffer Unit Discards trace data from the trace unit.

R_{BSMLW} The Trace Buffer Unit does not prefetch and cache address translations when the Trace Buffer Unit is Disabled.

I_{YHJDQ} When the Trace Buffer Unit is Disabled the trace unit might send trace data to an IMPLEMENTATION DEFINED trace bus.

R_{JYTYH} The trace unit does not send trace data to the IMPLEMENTATION DEFINED trace bus when the Trace Buffer Unit is Enabled.

I_{FPXHD} Figure E1.1 shows this IMPLEMENTATION DEFINED trace bus as a dotted line to an external trace *Sink*.

Details of this bus are outside the scope of this architecture, and might require further configuration. For example, if the trace unit implements FEAT_ETE and the trace bus is AMBA ATB, the ATID value is configured through the trace unit external trace registers.

E1.2.1.8 Restrictions on programming the Trace Buffer Unit

R_{MSPSD}   A current write pointer value is *out-of-range* if any of the following are true:

- The current write pointer is less-than the Base pointer, treating both pointers as unsigned integers.

- The current write pointer is greater-than-or-equal-to the Limit pointer, treating both pointers as unsigned integers.

- Bits [63:56] of the current write pointer are not equal to bits [63:56] of the Base pointer.

- Bits [63:56] of the current write pointer are not equal to bits [63:56] of the Limit pointer.

Note: R_{MSPSD} means the current write pointer is out-of-range if the Base pointer is not less-than the Limit pointer or bits [63:56] of the Base pointer are not equal to bits [63:56] of the Limit pointer.

R_{XXZHM}   A current write pointer or Trigger Counter value is *misaligned* if it is not a multiple of an IMPLEMENTATION DEFINED alignment specified by TRBIDR_EL1.Align.

R_{HXZZM}   A current write pointer or Trigger Counter value is a valid *restart value* if it was previously initialized with a value that was not out-of-range and not misaligned and later read from the applicable register when all of the following are true:

- The Trace Buffer Unit is Disabled.

- All Trace operations are Complete. See R_{NSFRQ} for the definition of Complete.

- No External Abort has been reported to the Trace Buffer Unit. TRBSR_EL1.EA is 0.

- No write by the Trace Buffer Unit has generated an Alignment fault.

- No write by the Trace Buffer Unit has generated an asynchronous SError interrupt exception.

R_{XZWXQ}   A current write pointer or Trigger Counter value is a *fault value* if it was previously initialized with a value that was not out-of-range and not misaligned or a value that was a valid restart value, and later read from the applicable register when all of the following are true:

- The Trace Buffer Unit is Disabled.

- All Trace operations are Complete. See R_{NSFRQ} for the definition of Complete.

- One of the following is true:

  – An External Abort has been reported to the Trace Buffer Unit. TRBSR_EL1.EA is 1.

  – A write by the Trace Buffer Unit has generated an Alignment fault.

  – A write by the Trace Buffer Unit has generated an asynchronous SError interrupt exception.

See also R_{XRLSC}.

I_{CRSGP}   An MMU fault does not generate a fault value. If software is able to fix the fault, then the Trace Buffer Unit can restart using the current write pointer and Trigger Counter values.

However, following an MMU fault:

- R_{YMVZL} means the Trace Buffer Unit Discards trace because Collection is stopped. That is, trace will be lost.

- R_{BQTGW} means that the Trigger Counter might be incorrect if a Detected Trigger has occurred.

See also S_{GTLCY}.

I_{SFTPM}   Following a trace buffer management event, or on a context switch, the current write pointer and Trigger Counter might be misaligned. If TRBIDR_EL1.Align is nonzero, software should treat bits [M:0] as SBZP when writing to the applicable register, where M is (TRBIDR_EL1.Align-1) in each of the following situations:

- When first creating a trace buffer, software sets bits [M:0] to zero, meaning the registers are set to an aligned value.

- On a context switch, the definitions of a restart value and fault value mean software does not have to validate or modify the value read from hardware.

A current write pointer restart value or fault value will not be out-of-range.

I_{VRFQC} A fault value is for error handling purposes only. Software must not cause the Trace Buffer Unit to become Enabled and Running with the current write pointer having a fault value.

Software context switching the Trace Buffer Unit will avoid this issue because the trace buffer management event sets TRBSR_EL1.S to 1, meaning the Trace Buffer Unit will not become Running following the context switch.

R_{JWWWM} If the current write pointer is written by a direct write with a misaligned value that is not a restart value and not a fault value, the value returned by a subsequent direct read of the current write pointer is UNKNOWN.

R_{MGZWR} If the current write pointer has an out-of-range value, or a misaligned value that is not a restart value when the Trace Buffer Unit attempts to write to the trace buffer, then any of the following might occur:

- If the value is out-of-range the current write pointer might be wrapped before or after the write, and the TRB_WRAP event might be generated.

- If the value is misaligned the write might generate an Alignment fault.

- The Trace Buffer Unit might write the trace data to any address in memory that is writable by a privileged access in the owning translation regime. These addresses are:

  – Virtual addresses in the owning translation regime if TRBLIMITR_EL1.nVM is 0.

  – Intermediate physical addresses in the owning Security state if TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations.

  – Physical addresses in the owning Security state if TRBLIMITR_EL1.nVM is 1 and the owning translation regime has no stage 2 translation.

- The write might generate a trace buffer management event with an UNKNOWN reason:

  – TRBSR_EL1.S is either set to 1 or unchanged.

  – TRBSR_EL1.WRAP is either set to 1 or unchanged.

  – TRBSR_EL1.EC is set to an UNKNOWN value.

  – TRBSR_EL1.MSS is set to an UNKNOWN value.

  – The TRB_WRAP event might be generated.

R_{CPDDM} If the Trigger Counter is written by a direct write with a misaligned value that is not a restart value, then all of the following apply:

- If the value is not a fault value, the value returned by a subsequent direct read of the Trigger Counter register is UNKNOWN.

- The generation of a Trigger Event while the Trace Buffer Unit remains Enabled and Running is UNPRE-DICTABLE.

I_{YSXXN} R_{XXZHM} and R_{CPDDM} mean an implementation that always keeps the current write pointer and/or Trigger Counter aligned to the IMPLEMENTATION DEFINED alignment specified by TRBIDR_EL1.Align, where TRBIDR_EL1.Align is greater-than-zero (byte alignment), can implement bits $[M{:}0]$ of the applicable register(s) as RAZ/WI bits, where M is (TRBIDR_EL1.Align-1).

R_{HXZZM} allows an implementation where an External Abort is reported to the Trace Buffer Unit and handled synchronously to implement TRBPTR_EL1$[M{:}N]$ (where N is implementation-specific and typically determined by the minimum memory access granule) as read/write bits for the purpose of reporting an External Abort fault address, but otherwise ignore the value in these bits. (If $N > 0$, bits $[(N-1){:}0]$ can be implemented as RAZ/WI.)

R_{DJMDD} When TRBLIMITR_EL1.E is 1, the PE might ignore a direct write to any of the following registers, other than a direct write to TRBLIMITR_EL1 that modifies TRBLIMITR_EL1.E:

- The current write pointer, TRBPTR_EL1.

- The Base pointer, TRBBASER_EL1.

- The Limit pointer, TRBLIMITR_EL1.

- The Trigger Counter, TRBTRG_EL1.

- TRBSR_EL1.

- TRBMAR_EL1.

Note

This means software must use appropriate Context synchronization operations to order a direct write that modifies TRBLIMITR_EL1.E with respect to other direct writes to Trace Buffer Unit registers. This includes a write to enable the Trace Buffer Unit by setting TRBLIMITR_EL1.E to 1.

See also:

- E1.2.3 *Synchronization and the Trace Buffer Unit*.
- E1.2.3.8 *UNPREDICTABLE behavior*.
- G3.1 *Context switching*.

E1.2.1.9 *Memory System Performance Resource and Monitoring Extension* (MPAM)

R_{WXSYG} If the MPAM Extension, FEAT_MPAM, is implemented then the MPAM information for accesses made by the Trace Buffer Unit to the trace buffer use the MPAM values of the owning Exception level and owning Security state.

FEAT_MPAM is defined by the Arm® Architecture Reference Manual Supplement; Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture [6].

I_{XLNWD} For example, if the owning Exception level is EL2 the trace buffer writes use MPAM2_EL2.PARTID_D and MPAM2_EL2.PMG_D. MPAM_NS is set for the owning Security state.

E1.2.1.10 Memory Tagging Extension

R_{YGMLW} If FEAT_MTE is implemented then the Trace Buffer Unit generates an Unchecked access for each access to the trace buffer.

Note: This is the case even when a Tagged Normal memory type is accessed.

See also:

- "Memory Tagging Extension" chapter of the *Arm® Architecture Reference Manual, for A-profile architecture* [1]

E1.2.1.11 Cache and TLB operations

I_{WYVXK} Translations used by the Trace Buffer Unit might be cached in a TLB.

R_{GQJMC} TLB maintenance operations that affect the TLB of the PE also affect any TLB caching translations for the Trace Buffer Unit of that PE.

R_{RNPNM} The PE is permitted, but not required, to cache all translations used by the Trace Buffer Unit in TLB caching structures that combine stage 1 and stage 2 of the translation. This includes when TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations.

S_{LSZDR} When TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations, the Trace Buffer Unit uses intermediate physical addresses (IPAs). R_{RNPNM} permits, but does not require, such translations to be cached in a TLB in such a way that an IPAS2 TLB maintenance operation is *not* sufficient to invalidate the cached copies. In this case, there is no virtual address (VA) for the translation.

If TRBLIMITR_EL1.nVM is 1 and the owning translation regime has stage 2 translations, then the following code executed at EL2 or above is sufficient to invalidate all cached copies of the stage 2 translations used by the Trace Buffer Unit of the IPA held in Xt for the current VMID:

```
1  TLBI IPAS2E1, Xt
2  DSB
3  TLBI VMALLE1
```

Equivalent architectural requirements apply to the IPAS2L instruction, except that the only TLB entries that must be invalidated by an IPAS2L instruction are those that come from the final level of the translation table lookup.

Equivalent sequences guaranteed to invalidate all entries invalidated by the above code sequence can be used, such as TLBI ALL or TLBI VMALLS1S2.

R_{DNFWB} Cache maintenance operations that affect the caches of the PE also affect data caching by the Trace Buffer Unit of that PE.

I_{MZQPT} R_{GQJMC} and R_{DNFWB} mean that the completion of any cache or TLB maintenance instruction includes its completion on all Trace Buffer Units for PEs that are affected by both the instruction and the DSB operation that is required to guarantee visibility of the maintenance instruction. See E1.2.3.7 *Detailed synchronization litmus tests* for more information.

E1.2.1.12 Effect on the exclusive monitors and transactions

R_{DCVBN} If an operation between Load-Exclusive and Store-Exclusive instructions is traced, and the the trace data is written to an unrelated address, then the write has no effect on the exclusive monitors.

R_{MDJNK} If an operation inside a transaction is traced, and the the trace data is written to an unrelated address, then the write has no effect on the transaction.

R_{NWSKV} If the Trace Buffer Unit writes to the marked address of an exclusives monitor in the Exclusive Access state, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- The write has the same effect on the exclusives monitor as a store by the PE or any other Observer to that address.

- The write has no effect on the exclusives monitor.

R_{FVRXJ} If the Trace Buffer Unit writes to the working set of a transaction, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- The write has the same effect on the transaction as a store by any other Observer to that address.

- The write has no effect on this transaction.

E1.2.2 Trace buffer management

I_{GYHBH} The Trace Buffer Extension supports the following *trace buffer modes*:

Circular Buffer mode   In Circular Buffer mode, when the current write pointer reaches the Limit pointer, it is wrapped by setting it to the Base pointer.

Wrap mode   As Circular Buffer mode, except that an interrupt request is generated when the current write pointer is wrapped.

Fill mode   As Wrap mode, except that trace collection stops when the current write pointer is wrapped.

I_{FZXSV} The trace buffer mode is controlled by TRBLIMITR_EL1.FM.

I_{NFZKS} A *trace buffer management event* occurs:

- On an Alignment fault, or MMU fault.

- On an External Abort.

- On a Trigger Event, if enabled.

- When the current write pointer is wrapped to the Base pointer and the trace buffer mode is not Circular Buffer mode. This event is known as:

  - A buffer wrap event, if the trace buffer mode is Wrap mode.

  - A buffer full event, if the trace buffer mode is Fill mode.

- On a programming error, when permitted as an UNPREDICTABLE behavior of the PE. For more information, see E1.2.1.8 *Restrictions on programming the Trace Buffer Unit* and E1.2.3.8 *UNPREDICTABLE behavior*.

- On an IMPLEMENTATION DEFINED event.

R_{HLKSG} On a trace buffer management event all of the following occurs:

- The interrupt request bit, TRBSR_EL1.IRQ, is set to 1.

- The *trace buffer management interrupt* signal, **TRBIRQ**, is asserted.

- Additional syndrome for the event might be written to TRBSR_EL1.MSS.

R_{LRTBP} **TRBIRQ** is a level triggered interrupt request driven by TRBSR_EL1.IRQ. This means that all of the following apply:

- A direct write that sets TRBSR_EL1.IRQ to 1 causes the interrupt request to be asserted.

- The interrupt request remains asserted until software clears TRBSR_EL1.IRQ to 0.

R_{TPPCF} When a GIC is implemented, **TRBIRQ** is configured as a PPI. **TRBIRQ** is signaled by the PE that implements the Trace Buffer Unit.

I_{HHRLX} The PPI number is not defined by the architecture. Arm recommends that the the PPI number is discoverable to an Operating System, for example using ACPI or Device Tree interfaces.

S_{XLNJY} Software must configure the trace buffer management interrupt to be taken to the correct Exception level.

I_{HJFLC} Buffer full, Alignment fault, and MMU fault trace buffer management events are synchronous. This means that the effect of these trace buffer management events setting TRBSR_EL1.S to 1, Collection is stopped, happens before any further trace is collected by the Trace Buffer Unit from the trace unit.

I_{JLZDN} The Trigger Event trace buffer management event initiates a trace unit flush meaning other trace might be written to the trace buffer. This might cause a second trace buffer management event to be generated before Collection is stopped by the Trigger Event trace buffer management event.

I_{ZLVHR} The **TRBIRQ** interrupt is always taken asynchronously by the PE, even if the event is reported synchronously to the Trace Buffer Unit.

I_{GVGPB} Following an Alignment fault, MMU fault, or External Abort trace buffer management event, TRBPTR_EL1 serves as a Fault Address Register.

For a fault or synchronous External Abort trace buffer management event, the frozen TRBPTR_EL1 is the address that generated the fault or External Abort.

For an asynchronous External Abort trace buffer management event, the frozen TRBPTR_EL1 is not guaranteed to be the address that generated the External Abort.

E1.2.2.1 Prioritization of a trace buffer management event

R_{MKCHT} Where multiple synchronous trace buffer management events occur on writing trace data, the PE prioritizes them as follows (from highest to lowest priority), reporting the highest priority event:

1. Synchronous fault.

2. Synchronous External Abort.

3. Buffer full event.

4. Buffer wrap event.

I_{BRLXK} Do not confuse the prioritization of trace buffer management events with the prioritization of trace buffer management interrupts by an interrupt controller.

R_{GTMJD} Asynchronous and IMPLEMENTATION DEFINED trace buffer management events are not prioritized relative to synchronous trace buffer management events.

E1.2.2.2 Buffer full and Buffer wrap events

R_{MBSHC} If the current write pointer is wrapped to the Base pointer and the trace buffer mode is Fill mode, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.

- TRBSR_EL1.WRAP is set to 1.

- The TRB_WRAP event is generated.

- If TRBSR_EL1.S is 0 then all of the following occur:

  - TRBSR_EL1.S is set to 1, Collection is stopped.

  - TRBSR_EL1.EC is set to `0x00`, other buffer management event.

  - TRBSR_EL1.BSC is set `0b00001`, buffer filled.

- The other fields in TRBSR_EL1 are unchanged.

After the trace buffer management event, the current write pointer will point to the Base pointer.

R_{VBDJZ} If the current write pointer is wrapped to the Base pointer and the trace buffer mode is Wrap mode, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.

- TRBSR_EL1.WRAP is set to 1.

- The other fields in TRBSR_EL1 are unchanged.

- The TRB_WRAP event is generated.

Because TRBSR_EL1.S is unchanged, trace continues to be collected and written to the trace buffer.

I_{GDSTS} If the current write pointer is wrapped to the Base pointer and the trace buffer mode is Circular Buffer mode, then all of the following occur:

- TRBSR_EL1.WRAP is set to 1.

- The other fields in TRBSR_EL1 are unchanged.

- The TRB_WRAP event is generated.

I_{VCDNP} If TRBSR_EL1.S is 1, the current write pointer is not updated, meaning the current write pointer is never wrapped when TRBSR_EL1.S is already 1.

S_{HKNBM} Software can configure the PMU to count the TRB_WRAP event and monitor how many times the current write pointer has wrapped, particularly in Circular Buffer mode or Wrap mode.

I_{JSQPH} See also G3.2 *Controlling generation of trace buffer management events*.

E1.2.2.3 Trigger Event

I_{HHLBM} The Trace Buffer Extension supports detection of a trigger condition from the trace unit. A trigger condition is typically used to stop trace capture to ensure trace is captured around a point of interest.

The trace unit defines how software programs the trace unit to generate trigger conditions.

A *Detected Trigger* is signaled to the Trace Buffer Unit by the trace unit when the trace unit detects a trigger condition. The trace unit defines whether the Detected Trigger is signaled synchronously or asynchronously to the trace data stream.

A *Trigger Event* occurs when the Trigger Counter has counted the specified number of trace bytes after a Detected Trigger. Software can set the Trigger Counter to zero to skip this step.

The *Trigger Counter* is a counter used to delay a Trigger Event for a specified number of trace bytes after a Detected Trigger.

 Figure E1.2 shows this.
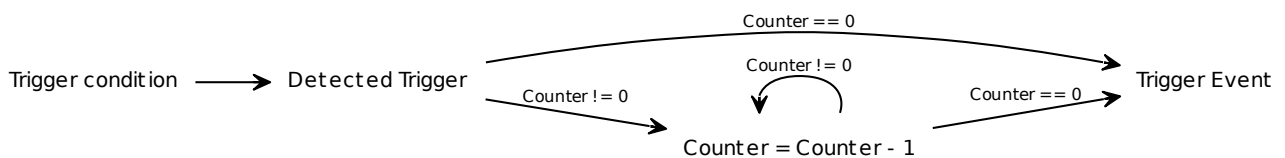
Figure E1.2: Trigger condition to Trigger Event

I_{YRQCN} For a trace unit that implements FEAT_ETE, event 0 is the trigger condition.

S_{DHJTB} When the trigger mode is set to Stop on trigger, software uses the Trigger Counter to control how trace is collected around the Detected Trigger as follows:

- Set the Trigger Counter to zero to trace *before* the Detected Trigger.

- Set the Trigger Counter to half the size of the trace buffer to trace *around* the Detected Trigger.

- Set the Trigger Counter to the size of the trace buffer to trace *after* the Detected Trigger.

I_{KLRBB} E1.2.1.8 *Restrictions on programming the Trace Buffer Unit* defines additional constraints on writing to the Trigger Counter.

R_{KSQJW} If a Detected Trigger occurs while the Trace Buffer Unit is Enabled, then all of the following occur:

- TRBSR_EL1.TRG is set to 1.

- The other fields in TRBSR_EL1 are unchanged.

R_{BQTGW} If all of the following are true, then the Trace Buffer Unit decrements the Trigger Counter by 1 for each byte of trace data written to the trace buffer by the Trace Buffer Unit:

- The Trigger Counter is nonzero.

- TRBSR_EL1.TRG is set to 1.

If the write generates an Alignment fault, MMU fault, or External Abort, it is UNPREDICTABLE whether the Trigger Counter is decremented.

R_{QFGNQ} If a Detected Trigger occurs when the Trigger Counter is nonzero and TRBSR_EL1.TRG is 0, the Trace Buffer Unit might decrement the Trigger Counter by an amount up to the value specified by TRBIDR_EL1.Align without writing any additional trace data to the trace buffer.

I_{QQKZF} An implementation might include an internal buffer that collects bytes of trace data into more convenient units before writing them to memory. For example, the width of the system bus or the length of a cache line.

In such an implementation, TRBIDR_EL1.Align specifies the size of this unit, and the Trigger Counter is decremented by the size of this unit when the write occurs, meaning the Trigger Counter is always aligned to the size specified by TRBIDR_EL1.Align.

However this means that if the Detected Trigger occurs when such an internal buffer is not empty, the Trace Buffer Unit will over-decrement the counter when the internal buffer is written to memory. R_{QFGNQ} permits this.

See also R_{BWNRF}.

R_{DHLQG} A Trigger Event is generated when the Trace Buffer Unit is Enabled and one of the following occurs:

- A Detected Trigger occurs when the Trigger Counter is zero and TRBSR_EL1.TRG is 0.

- The Trace Buffer Unit decrements the Trigger Counter to zero.

I_{ZGFSK} A Trigger Event is not generated when a Detected Trigger occurs, the Trigger Counter is set to zero and TRBSR_EL1.TRG is already 1.

A Trigger Event is not generated when the Trace Buffer Unit is Disabled.

R_{RWJNN} The Detected Trigger might be generated by a Trace operation $\mathbf{t_T}$. This might be the Trace operation generated by an instruction that also matched the trigger condition, or might be a Trace operation generated asynchronously by the trace unit to mark the trigger condition in the trace data. The trace unit defines this relationship for triggers.

R_{MLZZW} A Trigger Event might be generated by a Trace operation as follows:

- If the Trigger Event is generated when a Detected Trigger occurs when the Trigger Counter is zero and TRBSR_EL1.TRG is 0, and the Detected Trigger is generated by a Trace operation $\mathbf{t_T}$ then the Trigger Event is generated by the same Trace operation $\mathbf{t_T}$.

- If the Trigger Event is generated when the Trace Buffer Unit decrements the Trigger Counter to zero, then Trigger Event is generated by the Trace operation that generated the trace data that caused the Trigger Counter to decrement to zero.

A Trigger Event is not generated by a specific Trace operation if the Trigger Event is generated when a Detected Trigger occurs when the Trigger Counter is zero and TRBSR_EL1.TRG is 0, and the Detected Trigger is not generated by a specific Trace operation.

I_{GPHHS} The link between a Trigger Event and a Trace operation that generated it affects when the Trace operation is Microarchitecturally-finished and the behavior of the TSB CSYNC instruction. See R_{JQDJD}.

I_{NGLLQ} The Trace Buffer Extension supports the following *trigger modes*:

Stop on trigger Trace collection is stopped and an interrupt request is generated after a Trigger Event.

IRQ on trigger An interrupt request is generated after a Trigger Event.

Ignore trigger The Trace Buffer Unit ignores the trigger condition.

In the Stop on trigger and IRQ on trigger modes, software specifies the amount of trace that is collected after the trigger condition before the Trigger Event.

R_{XRRWP} If a Trigger Event is generated when collection is not stopped and the trigger mode is Stop on trigger, then all of the following occur:

- The Trace Buffer Unit initiates a *trace unit flush* of the trace unit.
- The TRB_TRIG event is generated.

On completion of the trace unit flush all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- If TRBSR_EL1.S is 0, then all of the following occur:
  - TRBSR_EL1.S is set to 1, Collection is stopped.
  - TRBSR_EL1.EC is set to `0x00`, other buffer management event.
  - TRBSR_EL1.BSC is set to `0b000010`, Trigger Event.
- The other fields in TRBSR_EL1 are unchanged.

After the trace buffer management event, the current write pointer will point to either the first byte after the last trace byte written to the trace buffer, or, if the last trace byte written to the trace buffer was the last byte in the trace buffer, the Base pointer.

I_{MCGFL} The trace unit defines the behavior and completion of a trace unit flush, including which Trace operations, if any, are accepted by the Trace Buffer Unit before the trace unit flush completes.

If the Detected Trigger is generated by a Trace operation t_T then the trace unit flush does not complete before the Trace Buffer Unit Accepts the trace data for t_T.

I_{VPLRF} Because the Trace Buffer Unit initiates a trace unit flush before stopping this means that, before TRBSR_EL1.S is set to 1:

- More trace might be written to the trace buffer after the Trigger Event is detected.
- This might generate other management events that set TRBSR_EL1.S to 1.

R_{XYPYF} If a Trigger Event is generated when collection is not stopped and the trigger mode is IRQ on trigger, then all of the following occur:

- The Trace Buffer Unit initiates a *trace unit flush* of the trace unit.
- The TRB_TRIG event is generated.

On completion of the trace unit flush all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.
- The other fields in TRBSR_EL1 are unchanged.

Because TRBSR_EL1.S is unchanged, trace continues to be collected and written to the trace buffer.

I_{LMQHK} If a Trigger Event is generated and the trigger mode is Ignore trigger, then all of the following occur:

- TRBSR_EL1 is unchanged.
- The TRB_TRIG event is generated.

R$_{\text{MWWHM}}$ If a Trigger Event is generated when Collection is stopped, then all of the following occur:

- TRBSR_EL1 is unchanged.

- The TRB_TRIG event is generated.

I$_{\text{NZYNN}}$ These rules mean that trace might be collected after the Trigger Event, but are included to ensure that trace for the instructions that caused the trigger condition is not discarded in common cases.

I$_{\text{JDWMT}}$ The trigger mode is controlled by TRBLIMITR_EL1.TM.

I$_{\text{SJWBD}}$ See also G3.2 *Controlling generation of trace buffer management events*.

E1.2.2.4 Faults

R$_{\text{QKLXR}}$ A write by the Trace Buffer Unit might generate one or more of the following faults:

Alignment fault If TRBPTR_EL1 is misaligned, the behavior is UNPREDICTABLE and a write to the trace buffer by the Trace Buffer Unit might generate an Alignment fault. See also R$_{\text{MGZWR}}$.

Translation fault Any access outside the virtual address or intermediate physical address space generates a Translation fault.

The translation of a virtual address or intermediate physical address to a physical address might generate Translation fault.

Writes to the trace buffer are made as privileged writes in the owning translation regime, meaning they are not affected by the TCR_ELx.E0PDy bits for the owning translation regime.

Address Size fault The translation of a virtual address or intermediate physical address to a physical address, or use of an out-of-range physical address, might generate an Address Size fault.

Permission fault Writes to the trace buffer are made as privileged writes in the owning translation regime. If the write does not have write permission, a Permission fault is generated. The value of PSTATE.PAN is ignored.

If the Trace Buffer Unit does not manage the dirty state in translation tables, then accesses ignore the Dirty Bit Modifier bit in Page and Block descriptors and as a result, might generate a Permission fault.

Access Flag fault If the Trace Buffer Unit does not manage the Access flag in translation tables or hardware management of the Access flag state is disabled for the owning translation regime, then any access to a Page or Block with the Access flag bit set to 0 in a descriptor will generate an Access Flag fault.

TLB Conflict fault IMPLEMENTATION DEFINED.

Unsupported atomic hardware update fault If hardware update of the translation tables is not guaranteed atomic in regard to other agents that access the memory, the translation of a virtual address to a physical address might generate an Unsupported atomic hardware update fault.

This document uses *MMU fault* to mean any of these faults other than Alignment fault.

R$_{\text{FYBCG}}$ Writes to the trace buffer by the Trace Buffer Unit never generate watchpoints.

I$_{\text{DTKGB}}$ Faults do not generate an actual Data Abort exception. The ESR and FAR registers are unchanged.

S$_{\text{GTLCY}}$ To avoid MMU faults, Arm recommends:

- Software pins the Pages or Blocks used for the trace buffer. This includes a hypervisor pinning these Pages or Blocks in the stage 2 translation.

- If the Trace Buffer Unit does not manage the Access Flag and dirty state, software marks the Pages or Blocks as accessed and dirty. Software can discover whether address translations performed by the Trace Buffer Unit manage dirty state and the Access flag from TRBIDR_EL1.F.

R_{FSPBK} If a write by the Trace Buffer Unit generates an Alignment fault or MMU fault, and TRBSR_EL1.S is 0, then all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.

- TRBSR_EL1.S is set to 1, Collection is stopped.

- TRBSR_EL1.EC is set to the appropriate one of the following values:

  - $0x24$, stage 1 Data Abort on write to trace buffer.

  - $0x25$, stage 2 Data Abort on write to trace buffer.

- TRBSR_EL1.FSC is set to indicate the type of the fault.

- TRBPTR_EL1 is set to the address that generated the fault.

- The other fields in TRBSR_EL1 are unchanged.

I_{ZNPQG} In the case of a stage 2 Data Abort on a write to trace buffer, the PE does not record whether the fault was due to a stage 2 fault on the access, or a stage 2 fault on a stage 1 translation table access.

E1.2.2.5 External aborts

R_{DJLWB} A write to the trace buffer might generate an *External Abort*, including an External Abort on a translation table walk or translation table update:

External Abort The write might generate a synchronous or asynchronous External Abort.

External Abort on translation table walk or translation table update The translation of a virtual address or intermediate physical address to a physical address might generate an External Abort.

R_{GQWBF} When a write to the trace buffer generates an External Abort, then one of the following occurs, and it is IMPLE-MENTATION DEFINED which:

- The External Abort is reported to the Trace Buffer Unit and treated as a trace buffer management event.

- The External Abort is not reported to the Trace Buffer Unit and generates an asynchronous SError interrupt exception at the PE.

R_{NZTKV} When an External Abort is reported to the Trace Buffer Unit, then one of the following occurs, and it is IMPLE-MENTATION DEFINED which:

- The External Abort is handled synchronously by the Trace Buffer Unit.

- The External Abort is handled asynchronously by the Trace Buffer Unit.

R_{TVKJR} If the Trace Buffer Unit handles External Aborts asynchronously, then all of the following apply:

- The External Abort trace buffer management event might not be generated until after a first trace buffer management event has set TRBSR_EL1.S is 1.

- Writes to the trace buffer might generate a second trace buffer management event after the External Abort trace buffer management event has set TRBSR_EL1.S is 1.

- The Trace Buffer Unit might collect further trace data from the trace unit and write it to memory before the External Abort trace buffer management event sets TRBSR_EL1.S to 1.

R_{XRLSC} When a write by the Trace Buffer Unit generates an External Abort reported to the Trace Buffer Unit, all of the following occur:

- The External Abort bit, TRBSR_EL1.EA, is set to 1.

- The Trace Buffer Unit stops writing trace data to the trace buffer.

- If the Trace Buffer Unit handles External Aborts synchronously, TRBPTR_EL1 is set to the address that generated the External Abort.

- If the Trace Buffer Unit handles External Aborts asynchronously, TRBPTR_EL1 is not guaranteed to be set to the address that generated the External Abort.

- If TRBSR_EL1.S is 0 then all of the following occur:

  - A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.

  - TRBSR_EL1.S is set to 1, Collection is stopped.

  - TRBSR_EL1.EC is set to the appropriate one of the following values:

    * `0x24`, stage 1 Data Abort on write to trace buffer.

    * `0x25`, stage 2 Data Abort on write to trace buffer.

  - TRBSR_EL1.FSC is set to indicate the type of External Abort.

- The other TRBSR_EL1 fields are unchanged.

I_{TZNMV} Reporting the External Abort to the Trace Buffer Unit and generating a trace buffer management event:

- Sets TRBSR_EL1.S to 1 and so Discards further trace data.

- Allows error recovery software to isolate the event to the actions of the Trace Buffer Unit.

Not reporting the External Abort to the Trace Buffer Unit and generating an asynchronous SError interrupt exception at the PE:

- Means that, while tracing is not prohibited, the trace unit might continue generating trace data that the Trace Buffer Unit Accepts. However, the SError interrupt might be taken to an Exception level where tracing is prohibited.

- Might not allow error recovery software to isolate the event and error containment.

E1.2.2.6 IMPLEMENTATION DEFINED management events

R_{CHNSL} When IMPLEMENTATION DEFINED conditions are met all of the following occur:

- A trace buffer management event is generated. This sets TRBSR_EL1.IRQ to 1.

- TRBSR_EL1.S is set to 1, Collection is stopped.

- TRBSR_EL1.EC is set to `0x1F`, IMPLEMENTATION DEFINED buffer management event.

- TRBSR_EL1.MSS is set to an IMPLEMENTATION DEFINED value.

- The other fields in TRBSR_EL1 are unchanged.

I_{LKLHH} The intent of this event is for cases such as errata workarounds to allow an implementation to report any failure to write data to the buffer that is not covered by other codes. Arm recommends that such mechanisms are disabled on reset.

E1.2.3 Synchronization and the Trace Buffer Unit

I_{NMWZY} Program-flow trace data is generated by traced instructions. When an instruction is executed:

1. The PE decides whether to create a *Trace operation* for the instruction.

2. If created, the Trace operation generates the program-flow trace data.

For some trace unit implementations, Speculative instructions might generate Trace operations, as well as architecturally Resolved instructions.

See Chapter G2 *Stages of execution* for more information on these terms.

The trace unit might also generate asynchronous Trace operations, that are not causally related to an executed instruction. If the trace unit implements the FEAT_ETE then the ETE Resources can generate Trace operations that are not causally related to an instruction or Speculative instruction.

The architecture defines a Trace Synchronization event that synchronizes the operation of Trace operations with the execution of instructions. Without correct use of the Trace Synchronization event, a Trace operation might for instance read a stale value from a System register, causing trace data to be written to the wrong memory location, or the Trace Buffer Unit to otherwise generate unpredictable software behavior. See also E1.2.3.8 *UNPREDICTABLE behavior*.

R_{ZVDST} Trace operations operate independently of the instructions that are executed on the PE and make indirect reads and indirect writes of System registers as an *external agent*.

I_{TPVQR} The *Arm® Architecture Reference Manual, for A-profile architecture* [1] defines the synchronization requirements for direct reads, direct writes, indirect reads and indirect writes of System registers made by instructions and external agents.

R_{NRLDV} The indirect reads of the TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE} *trace filter controls* when determining whether the current Execution stream is part of a prohibited trace region and an instruction **A** should generate a Trace operation, are treated as indirect reads made by **A**.

Note: This rule is defined by FEAT_TRF.

R_{SXXQJ} Each System register access made by the trace unit is one of the following, and the trace unit defines which:

- An indirect read or indirect write $\mathbf{rw_A}$ made by an instruction **A**. For example, to determine whether to generate a Trace operation for **A**.

- An indirect read or indirect write $\mathbf{rw_{tA}}$ made by a Trace operation $\mathbf{t_A}$. This is in addition to System register accesses defined by this manual as indirect reads or indirect writes made by the Trace Buffer Unit.

- An other indirect read or indirect write **rw**, not directly related to either an instruction or Trace operation. The trace unit defines the synchronization requirements for these registers.

I_{DDLVC} In addition to the registers listed by R_{NRLDV}, for the ETE, the following ETE System registers are indirectly read by an instruction **A** to determine whether **A** should generate a Trace operation:

- TRCPRGCTLR.EN, the Trace unit enable bit in the Programming Control Register.

- TRCOSLSR.OSLK, the Trace OS Lock Status Register.

This manual defines which System registers are indirectly read or indirectly written by the Trace Buffer Unit as part of the Trace operation $\mathbf{t_A}$ for a traced instruction **A**. For example:

- Trace Buffer Unit System registers.

- VMSA System registers and SCR_EL3.NS, when translating addresses generated by the Trace Buffer Unit.

Other System registers are indirectly read or indirectly written by the trace unit as part of the Trace operation $\mathbf{t_A}$ for a traced instruction **A**. For example:

- Other ETE System registers.

- If context tracing is enabled, the applicable Context ID register or registers, CONTEXTIDR_EL1 or CONTEXTIDR_EL2.

- If trace timestamping is enabled, any applicable counter offset, CNTVOFF_EL2 or CNTPOFF_EL2.

Some ETE System registers are indirectly read or indirectly written by, for example, the ETE Resources when generating Trace operations or updating the ETE Resources, and are made by neither a Trace operation of an instruction nor an instruction.

The behavior of the ETE is defined by FEAT_ETE. See also E1.2.3.2 *Trace synchronization and the Trace Unit*.

I_{GVCVL} The indirect reads and indirect writes to Trace Buffer Unit, VMSA and other System registers made by the Trace Buffer Unit are made by the Trace operation t_{OP}.

I_{BYSGH} To synchronize Trace operations, software must use the `TSB CSYNC` instruction to generate a Trace Synchronization event.

R_{GTCKK} In the absence of any explicit synchronization, the trace unit generates the trace data for an instruction and the Trace Buffer Unit Accepts, Discards, or Rejects the trace data in finite time. However:

- If the Trace Buffer Unit Accepts the trace data, then the write of the trace data to memory requires explicit synchronization to Complete.

- The indirect writes to System registers made by a Trace operation require explicit synchronization to guarantee they are observable.

E1.2.3.1 Trace Synchronization event

R_{VWJNN} Executing a `TSB CSYNC` instruction generates a *Trace Synchronization event*.

R_{JQDJD} A Trace operation t_{OP} is not *Microarchitecturally-finished* before all of the following are true:

- All indirect reads and indirect writes of System registers made by t_{OP} have been performed.

- If t_{OP} generates a Trigger Event that in turn initiates a *trace unit flush*, then all of the following are true:

    - The trace unit flush is complete.

    - All Trace operations the Trace Buffer Unit Accepts before the trace unit flush completes are Microarchitecturally-finished.

    - All indirect writes to System registers made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

Indirect reads and writes include but are not limited to the following:

- All indirect reads and indirect writes of the Trace Buffer Unit, VMSA System registers, and SCR_EL3.NS made by memory accesses performed by t_{OP}.

- All indirect writes to System registers made by a trace buffer management event generated by t_{OP}.

However, this does explicitly exclude any indirect writes of System registers made in response to an External Abort by the access.

R_{NSFRQ} A Trace operation t_{OP} is *Complete* when it is Microarchitecturally-finished and all memory accesses performed by t_{OP} are Complete and any indirect writes of System registers made in response to an External Abort response to the access have been performed.

R_{MRVPT} If, following a Context synchronization event **CSE** the PE is executing in a Trace Prohibited region, a `TSB`↪`CSYNC` executed in the Trace Prohibited region and Non-debug state in program order after **CSE** is not Microarchitecturally-finished before all of the following are true:

- All Trace operations t_A generated by instructions **A** in program order before **CSE** are Microarchitecturally-finished.

- All Trace operations $\mathbf{t_S}$ generated by Speculative instructions **S** that are not in speculative execution order after **CSE** are Microarchitecturally-finished.

- All Trace operations $\mathbf{t_R}$ generated by the trace unit are Microarchitecturally-finished.

- The trace unit enters a state where the trace unit does not generate further Trace operations and does not signal a Detected Trigger. The trace unit remains in this state while the PE is executing in the Trace Prohibited region.

- If a *trace unit flush* is initiated by a Trigger Event before the TSB CSYNC is Microarchitecturally-finished, the trace unit flush is complete, all Trace operations the Trace Buffer Unit Accepts before the trace unit flush completes are Microarchitecturally-finished, and any indirect writes made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

These Trace operations are *synchronized* by the TSB CSYNC.

R_{ZCDDS} A direct write $\mathbf{W_2}$ to a System register made by an instruction **B** is Coherence-after an indirect read or indirect write $\mathbf{rw_1}$ of the same System register made by a Trace operation $\mathbf{t_A}$ for a traced instruction **A** if all of the following are true:

- Either **A** is executed in program order before a Context synchronization event **CSE**, or **A** is **CSE**.

- **CSE** is in program order before a Trace synchronization barrier **TSB**.

- **B** is executed in program order after **TSB**.

- After executing **CSE** the PE is in a Trace Prohibited region and **TSB** is executed in Non-debug state in the same Trace Prohibited region.

Note

- R_{ZCDDS} emerges from the requirement in R_{MRVPT} for the Trace operations to be Microarchitecturally-finished by the TSB CSYNC operation.

- E1.2.3.3 *Self-hosted trace extension synchronization rules* and E1.2.3.6 *Trace synchronization in Debug state* define further rules for the operation of TSB CSYNC.

I_{ZKRZH} The PE does not stall indefinitely (or until interrupted) waiting for a TSB CSYNC. For example, the TSB CSYNC must not wait until there is no trace data left to write if the trace unit is capable of producing a constant stream of trace data.

I_{ZLDPS} A PE might abandon a TSB CSYNC executed in Non-debug state before it is Microarchitecturally-finished to take an interrupt, so long as the preferred return address is set such that the TSB CSYNC is re-executed when the interrupt handler completes. That is, the TSB CSYNC is only Speculatively executed.

R_{CKVWP} Absent any Context synchronization event or DSB Data synchronization barrier, a TSB CSYNC instruction is not required to execute in program order with respect to other instructions or memory accesses. This means that software must execute additional barriers to guarantee that the Trace operations are Microarchitecturally-finished and/or Complete.

X_{FVTHX} A simple description of TSB CSYNC is that it does not complete and allow other instructions to execute until the trace unit and Trace Buffer Unit has output all trace for instructions that were executed in program order before the most recent Context synchronization event.

An implementation might generate a *trace unit flush* to ensure this happens.

Furthermore, if executed in a Trace Prohibited region, the trace unit is inactive once the TSB CSYNC is Microarchitecturally-finished.

However, the writes to memory and the effects of the operations on System registers (such as the trace buffer pointer address registers) might require further synchronization to be observable to following instructions.

E1.2.3.2 Trace synchronization and the Trace Unit

I_{ZYZGZ} The ETE has *Resources* that can generate Trace operations that are not directly generated by an instruction or Speculative instruction.

The ETE specification defines the following rules:

- The Resources do not generate Trace operations in the *Paused* state.

- If, following a Context synchronization event, the PE is executing in a Trace Prohibited region and the ETE is enabled, the ETE pauses the ETE Resources.

- How software synchronizes indirect writes to System registers made by Trace operations generated by Resources.

The behavior of the ETE is defined by FEAT_ETE.

I_{GFJWK} For a trace unit that implements FEAT_ETE, FEAT_ETE defines further rules defining the behavior of the trace unit when a TSB CSYNC instruction is executed.

R_{QVGHP} A Trace operation t_R generated by ETE Resources inherits the synchronization requirements for a Trace operation generated by an instruction **A**, even if no Trace operation is generated by **A**, provided that one of of the following applies:

- The requirement is that **A** is executed in program-order after **CSE** and tracing was prohibited before **CSE** and is allowed after **CSE**.

- The requirement is that either **A** is executed in program-order before **CSE** or **A** is **CSE**, and tracing was allowed before **CSE** and is prohibited after **CSE**.

R_{YWSDB} If the trace unit becomes enabled when the PE is executing a Trace Prohibited region, it does not generate any Trace operations, including Trace operations for Speculative instructions and other Trace operations not generated by instructions, until the PE enters a region where tracing is allowed.

I_{XGQRM} The trace unit defines the sequence by which software enables the trace unit.

E1.2.3.3 Self-hosted trace extension synchronization rules

I_{CMGRF} FEAT_TRF defines further rules for the TSB CSYNC instruction. In particular, how a TSB CSYNC instruction synchronizes direct reads and indirect writes to a System register with respect to indirect reads and indirect writes of the same System register made by Trace operations.

These rules are repeated in this document as R_{NRLDV}, R_{BFJKD}, R_{MJQXM}, R_{TSPXF}, and R_{VTNCS}.

R_{TSPXF} is similar to R_{ZCDDS} and R_{XQVZW} . However:

- R_{TSPXF} applies whether the TSB CSYNC operation is executed in a trace prohibited or trace allowed region, in both Non-debug state and Debug state. R_{ZCDDS} applies only when the TSB CSYNC is executed in a Trace Prohibited region and the PE is in Non-debug state, and R_{XQVZW} applies only when the TSB CSYNC is executed when the trace unit is disabled and the PE is in Debug state.

- R_{TSPXF} applies only to System registers accessed by the trace unit as part of a Trace operation. R_{ZCDDS} and R_{XQVZW} apply to all System register accesses made by the Trace operation and includes indirect reads and indirect writes made by the Trace Buffer Unit.

See R_{SXXQJ} and I_{DDLVC}.

R_{NNRHD} is a further rule from FEAT_TRF concerning synchronization of the trace filter controls System registers by a Context synchronization event. This rule was not present in the original FEAT_TRF specification, and was added as an later erratum.

R_{BFJKD} An indirect read r_1 of a System register made by a Trace operation t_A for a traced instruction **A** Reads-from a direct write W_2 to the same System register made by an instruction **B** if all of the following are true:

- **A** is executed in program order after a Context synchronization event **CSE**.

- **B** is executed in program order before **CSE**.

See also I_{FVBPF}.

S_{YPWVJ} R_{BFJKD} means that, if the PE enters a region where tracing is allowed by executing a Context synchronization event, such as an ERET instruction when SCTLR_ELx.EOS is 1, then all indirect reads and writes of System registers made by Trace operations generated after entering the tracing allowed region will observe the values in those System registers written by direct writes before the Context synchonization event.

R_{MJQXM} An indirect write w_1 of a System register made by a Trace operation t_A for a traced instruction **A** is Coherence-after a direct write W_2 of the same System register made by an instruction **B** if all of the following are true:

- **A** is executed in program order after a Context synchronization event **CSE**.

- **B** is executed in program order before **CSE**.

See also I_{FVBPF}.

R_{TSPXF} A direct write W_2 to a System register made by an instruction **B** is Coherence-after an indirect read or indirect write rw_1 of the same System register made by the trace unit as part of a Trace operation t_A for a traced instruction **A** if all of the following are true:

- Either **A** is executed in program order before a Context synchronization event **CSE**, or **A** is **CSE**.

- **CSE** is in program order before a Trace synchronization barrier **TSB**.

- **B** is executed in program order after **TSB**.

See also I_{MQTRQ}.

R_{VTNCS} A direct read R_2 of a System register made by an instruction **B** Reads-from an indirect write w_1 to the same System register made by a Trace operation t_A for a traced instruction **A** if all of the following are true:

- Either **A** is executed in program order before a first Context synchronization event CSE_1, or **A** is CSE_1.

- CSE_1 is in program order before a Trace synchronization barrier **TSB**.

- **TSB** is executed in program order before a second Context synchronization event CSE_2.

- **B** is executed in program order after CSE_2.

See also I_{SLWRW}.

R_{NNRHD} An instruction **A** in program-order after a direct write **W** that modifies one of the trace filter controls, TRFCR_EL1.{E1TRE, E0TRE} and TRFCR_EL2.{E2TRE, E0HTRE}, Reads-from **W** when determining whether **A** should generate a Trace operation, if there is no intervening direct write to the same register and any of the following is true:

- **A** is in program-order after a Context synchronization event **CSE** and **CSE** is in program-order after **W**. (This is the architectural rule described by *Chapter G1 Synchronization requirements for System registers* and illustrated in *Figure G1.3*.)

- An instruction **B** Reads-from **W** when determining whether **B** should generate a Trace operation, and **A** is in program-order after **B**.

I_{VKRGM} R_{NNRHD} means that for the instructions between a direct write to one of the trace filter controls to either enable or disable trace at the current Exception level and a following Context synchronization event, although it is UNPREDICTABLE whether each instruction observes the old or new values of the trace filter controls, once one instruction has observed the new value, all subsequent instructions also observe the new value.

However this might not happen, and all instructions might observe the old values until the Context synchronization event occurs.

This guarantees that trace switches either on and off cleanly, and is required by program-flow trace protocols.

E1.2.3.4 Trace synchronization and memory barriers

R_{VLWDM} If, following a Context synchronization event **CSE** the PE is executing in a Trace Prohibited region, a DSB with required access types of reads and write is executed in program order after a TSB CSYNC operation that is executed in the Trace Prohibited region in program order after **CSE**, then in addition to the requirements in the *Arm® Architecture Reference Manual, for A-profile architecture* [1], the DSB does not complete until all explicit memory accesses of the required access type made by the Trace operations synchronized by the TSB CSYNC are Complete for the set of observers in the required shareability domain.

See also R_{MPXXN}.

R_{DFHWV} An explicit read, explicit write, translation table walk, cache maintenance operation, or TLB invalidate operation M_1 will be Observed-by a read or a write RW_2 of a Location made by a Trace operation t_A relating to a traced instruction **A** if all of the following are true:

- **A** is executed in program-order after a Context synchronization event **CSE**.
- **CSE** is in program order after a Data Synchronization Barrier **DSB**.
- **DSB** does not complete before M_1 is complete.

I_{NLHGN} In R_{DFHWV}, **CSE** is required to allow **A** to be traced as a Speculative instruction before it is Canceled or Resolved. The equivalent rule for *Statistical Profiling Extension* (SPE) does not require **CSE** as SPE cannot write profiling records to memory until the profiled operation is Canceled or Resolved. See E1.2.3.5 *Trace of Speculative execution*.

R_{JPPZZ} For the indirect writes to TRBPTR_EL1 and TRBSR_EL1 that are made as a result of an External Abort on a write of trace data to memory, the synchronization rules apply only after the write to memory has completed.

E1.2.3.5 Trace of Speculative execution

I_{ZGCVG} In the standard model of execution for an instruction, instructions are executed as Speculative operations and then later become one of the following:

- Resolved. These instructions will then proceed to Complete.
- Canceled.
- Transaction-failed, if FEAT_TME is implemented.
- Transaction-canceled, if FEAT_TME is implemented.

A trace unit might generate trace for Speculative instructions before they are Resolved, Canceled, Transaction-failed or Transaction-canceled, and this trace can be written to memory. This means that, as well as indirectly reading System registers or memory, a Trace operation t_S for a Speculative instruction **S** might perform any of the following:

- Indirectly write to System registers.
- Write to memory.

This sets Trace operations apart from the normal operation of instructions, as the Arm architecture prohibits, for example, a Canceled instruction from updating a System register or memory. (Performance Monitors and Statistical Profiling can also cause speculative updates of System registers and memory.)

I_{HBYGS} The preceding rules deal with the ordering of Trace operations for Resolved (and ultimately Complete) instructions, and the requirements for synchronization based on the position of those instructions in the program order of the Execution stream.

This section extends the rules to cover Speculative instructions that are later Canceled, Transaction-canceled or Transaction-failed, and for Trace operations generated by ETE Resources. Because these Trace operations are not generated by instructions that Complete, they cannot be determined to be in *program order* with respect to architecturally Complete instructions.

R_{FBFRS} A Speculative instruction **S** is in *Speculative execution-order* after an instruction **A** if **S** will be in program order after **A** if **S** is Resolved, even if **S** is subsequently Canceled, Transaction-failed, or Transaction-canceled.

I_{VFLVX} For example, a Speculative instruction **S** is in Speculative execution-order after a Resolved instruction **A** if either of the following are true:

- The branch predictor mispredicted **A** and, had the prediction been correct, **S** would be in program order after **A**. *Branch predictor* means any structure that causes the PE to execute Speculative instructions. This includes, for example, branch history buffers, branch target caches, and instruction trace caches. It is not limited to structures that predict only the direction and/or target of branch instructions.

- **S** forms part of a Transaction **T** that was Transaction-canceled or Transaction-failed and **A** is the Resolved TSTART operation for the outermost Transaction containing **T**.

I_{KTCMP} Speculative execution-order does not provide a complete ordering. A pair of Speculative instructions **A** and **B** might not be ordered with respect to each other. For example, if **A** and **B** are respectively the result of different incorrect predictions by the branch predictor. However, each Speculative instruction is in Speculative execution-order after at least one Resolved instruction.

I_{VBBVY} The *Arm® Architecture Reference Manual, for A-profile architecture* [1] requires that instructions that generate Context synchronization events do not appear to be executed speculatively.

R_{HCVVS} A Trace operation $\mathbf{t_S}$ generated by a Speculative instruction **S** that is in Speculative execution-order after a Resolved instruction **A** inherits the synchronization requirements for a Trace operation generated by **A**, even if no Trace operation is generated by **A**, provided that one of of the following applies:

- The requirement is that **A** is executed in program-order after **CSE**, and **S** is in Speculative execution-order after **CSE**.

- The requirement is that either **A** is executed in program-order before **CSE** or **A** is **CSE**, and **S** is not in Speculative execution-order after **CSE**.

See also I_{WLLQH} and I_{JGXWM}.

E1.2.3.6 Trace synchronization in Debug state

R_{SCRQL} If FEAT_TRBE is implemented, a TSB CSYNC instruction can be executed in Debug state.

Note

FEAT_TRF does not require that TSB CSYNC can be executed in Debug state.

R_{WYHRT} If the trace unit is disabled, then a TSB CSYNC executed in Debug state is not Microarchitecturally-finished before all of the following are true:

- All Trace operations $\mathbf{t_A}$ generated by instructions **A** in program order before the PE entered Debug state are Microarchitecturally-finished.

- All Trace operations $\mathbf{t_S}$ generated by Speculative instructions **S** that are not in speculative execution order after the entry to Debug state are Microarchitecturally-finished.

- All Trace operations t_R generated by the trace unit are Microarchitecturally-finished.

- The trace unit enters a state where the trace unit does not generate further Trace operations and does not signal a Detected Trigger. The trace unit remains in this state while the PE is in Debug state.

- If a *trace unit flush* is initiated by a Trigger Event before the TSB CSYNC is Microarchitecturally-finished, the trace unit flush is complete, all Trace operations the Trace Buffer Unit Accepts before the trace unit flush completes are Microarchitecturally-finished, and any indirect writes made by the Trace Buffer Unit on completion of the trace unit flush have been performed.

These Trace operations are synchronized by the TSB CSYNC.

R_{XQVZW} A direct write W_2 to a System register made by an instruction **B** is Coherence-after an indirect read or indirect write rw_1 of the same System register made by a Trace operation t_A for a traced instruction **A** if all of the following are true:

- **A** is executed in program order before the PE entered Debug state.

- **B** is executed in program order after a Trace synchronization barrier **TSB**.

- **TSB** was executed in Debug state when the trace unit was disabled.

Note

- R_{XQVZW} emerges from the requirement in R_{WYHRT} for the Trace operations to be Microarchitecturally-finished by the TSB CSYNC operation.

- E1.2.3.1 *Trace Synchronization event* and E1.2.3.3 *Self-hosted trace extension synchronization rules* define further rules for the operation of TSB CSYNC.

R_{MPXXN} A DSB with required access types of reads and write is executed after a TSB CSYNC operation executed in Debug state, then in addition to the requirements in the *Arm® Architecture Reference Manual, for A-profile architecture* [1], the DSB does not complete until all explicit memory accesses of the required access type made by the Trace operations synchronized by the TSB CSYNC are Complete for the set of observers in the required shareability domain.

E1.2.3.7 Detailed synchronization litmus tests

I_{DJJBQ} This section details example synchronization scenarios and litmus tests for TSB CSYNC. These are derived from FEAT_TRF and E1.2.3.1 *Trace Synchronization event*.

This section uses the terms **program order**, **Reads-from** and **Coherence-after** to define the ordering of System register and memory accesses made by Trace operations. These terms are defined for memory accesses in the *Arm® Architecture Reference Manual, for A-profile architecture* [1]. For the purposes of this section, these terms are used for System registers as well as memory accesses.

See Chapter G1 *Synchronization requirements for System registers* for more information on how these terms are used in this section.

The terms **external agent**, **Observer**, and **Observed-by**, also used in this section, are also defined in the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

This section does not describe the synchronization rules in Debug state defined in E1.2.3.6 *Trace synchronization in Debug state*. In general, litmus tests for Debug state can be derived by applying the following modifications:

- Where a rule mentions a Context synchronization event (CSE) coming before a TSB CSYNC operation, if the TSB CSYNC is executed in Debug state, then the entry to Debug state can replace that CSE for the rule.

- Where a rule mentions the PE executing instructions in a Trace Prohibited region following the CSE, then executing the instructions in Debug state with the trace unit disabled is sufficient for the rule.

Exit from Debug state is a Context synchronization event.

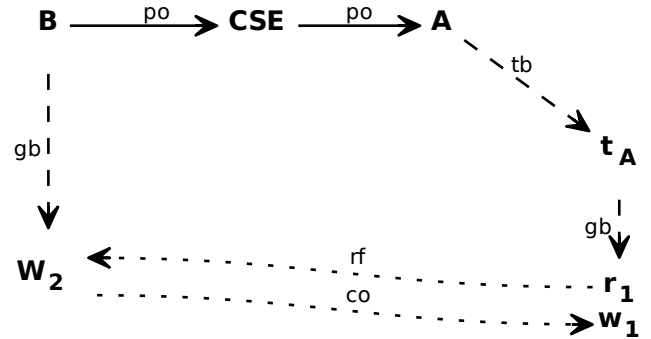I_{FVBPF} Figure E1.3 shows R_{BFJKD} and R_{MJQXM}.

Figure E1.3: Indirect read or indirect write by Trace operation after direct write

I_{MQTRQ} Figure E1.4 shows R_{TSPXF}.
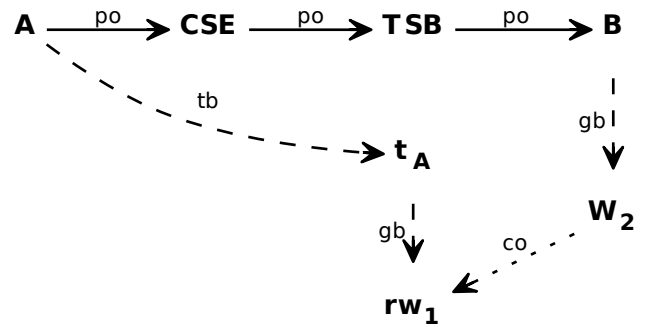

Figure E1.4: Direct write after indirect read or indirect write by Trace operation

I_{SLWRW} In R_{VTNCS}, the second Context synchronization event **CSE$_2$** is required to ensure the direct read **B** is not executed
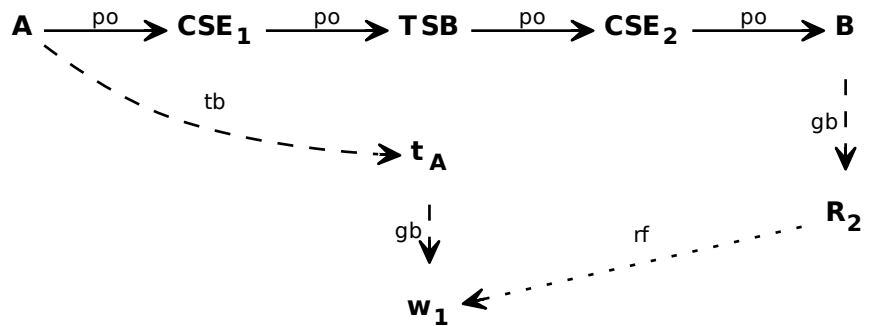   before the synchronization barrier **TSB**. Figure E1.5 shows this.

Figure E1.5: Direct read after indirect write by Trace operation

---

Note

If the trace is not prohibited after the Context synchronization event, further Trace operations might be generated that are not guaranteed to be synchronized by the `TSB CSYNC`.

Trace is prohibited at higher Exception levels than the owning Exception level. This means that if the PE takes an exception to a higher Exception level than the owning Exception level then trace is prohibited at by taking the exception.

I_{NWRPJ} Figure E1.6 shows R_{DFHWV} for an explicit read or a write $\mathbf{M_1}$ of a Location made by an instruction $\mathbf{B}$ in program order before **DSB**.
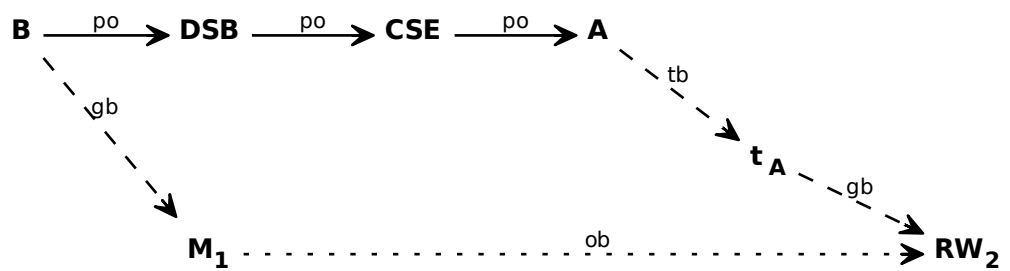

Figure E1.6: Trace operation observing memory operation

I_{JBRHG} If all of the following are true, R_{VLWDM} requires that a `DSB` with required access types of reads and writes does not complete until at least all reads or writes **RW** made by the Trace Buffer Unit for all Trace operations $\mathbf{t_A}$ relating to a traced instruction $\mathbf{A}$ are complete for the set of the observers in the required shareability domain:

- Either $\mathbf{A}$ is executed in program order before a Context synchronization event **CSE**, or $\mathbf{A}$ is **CSE**.

- The PE is executing in a Trace Prohibited region after **CSE**.

- **CSE** is in program order before a Trace synchronization barrier **TSB**.

- **TSB** is executed in program order before the `DSB`.

Figure E1.7 shows a read or a write $\mathbf{RW_1}$ of a Location made by the Trace Buffer Unit for a Trace operation $\mathbf{t_A}$ relating to a traced instruction $\mathbf{A}$ is complete and therefore will be Observed-by a read or a write $\mathbf{RW_2}$ of the same Location made by an instruction $\mathbf{B}$ executed in program order after a `DSB ISH` instruction.
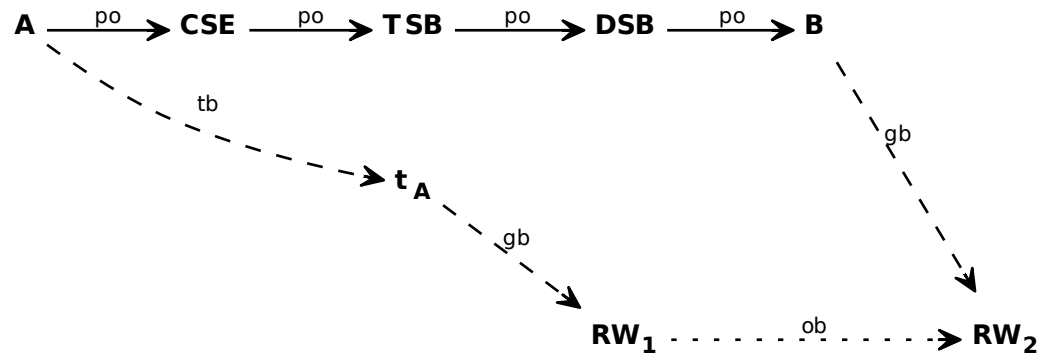
Figure E1.7: Observation of Trace operation memory access

I_{WLLQH} R_{HCVVS} defines Trace synchronization for Speculative instructions.

For example, an indirect read r_1 of a System register made by a Trace operation t_S for a traced Speculative instruction **S** Reads-from a direct write W_2 to the same System register made by an instruction **B**, if all of the following are true:

- **S** is executed in Speculative execution-order after a Resolved instruction **A**.

- **A** is executed in program order after a Context synchronization event **CSE**.

- **B** is executed in program order before **CSE**.
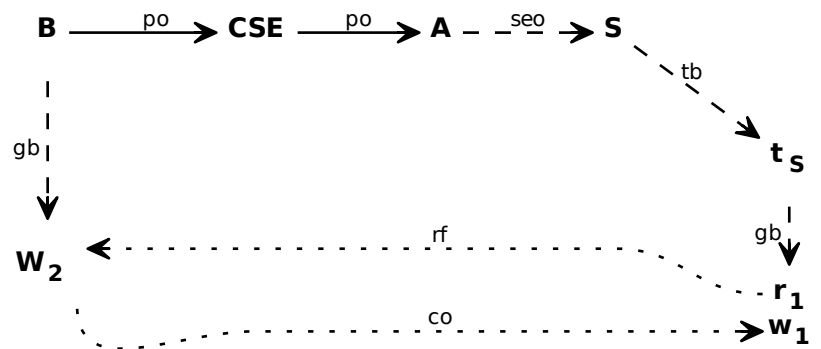
Figure E1.8 shows this.

Figure E1.8: Speculative indirect read or indirect write by Trace operation after direct write

I_{JGXWM} R_{HCVVS} defines Trace synchronization for Speculative instructions.

For example, if all of the following are true, a DSB with required access types of reads and writes does not complete until at least all reads or writes **RW** made by the Trace Buffer Unit for all Trace operations t_S relating to a traced Speculative instructions **S** are complete for the required shareability domain:

- **S** is executed in Speculative execution-order after a Resolved instruction **A**.

- **A** is executed in program order before a Context synchronization event **CSE**.

- **S** is not in Speculative execution-order after **CSE**.

- **CSE** is in program order before a Trace synchronization barrier **TSB**.

- The PE is executing in a Trace Prohibited region after **CSE**.

- **TSB** is executed in program order before the DSB.

Figure E1.9 shows a read or a write RW_1 of a Location made by the Trace Buffer Unit for a Trace operation t_S relating to a traced Speculative instruction **S** is Complete and therefore will be Observed-by a read or a write RW_2 of the same Location made by an instruction **B** executed in program order after a DSB ISH instruction.
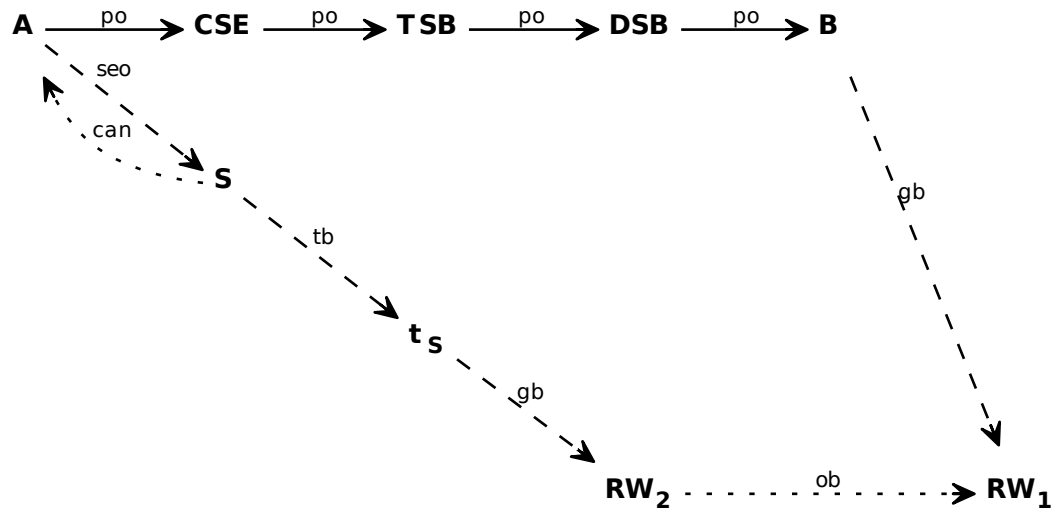

Figure E1.9: Observation of Speculative Trace operation memory access

E1.2.3.8 UNPREDICTABLE behavior

R_{PFJJZ} In the absence of correct synchronization events, it is UNPREDICTABLE whether an indirect read made by a Trace operation of a System register updated by a direct write will return the old or the new values. A Trace operation might make multiple indirect reads of a System register, and each might return a different one of the old or the new values.

I_{GGGPH} For example, a Trace operation might read MDCR_EL2.E2TB twice, as follows:

1. When the Trace operation is first generated, to evaluate TraceAllowed() and determine whether trace is prohibited.

2. When the Trace operation is complete and ready to be written to memory, to evaluate TraceBufferOwner() to determine the context for TRBPTR_EL1.

If MDCR_EL2.E2TB is updated without proper synchronization between these two events, both the old and new value might be used.

I_{NRQQF} In the absence of correct synchronization events, it is UNPREDICTABLE whether a direct read of a System register updated by an indirect write made by a Trace operation will return the old or the new values.

I_{TRWFM} If an instruction is traced because the Trace Buffer Unit is Enabled and Running, but a later indirect read of a System register by the Trace operation for the instruction determines that the trace data cannot be written to memory, because the Trace Buffer Unit now appears to be Disabled, then one of the following occurs, and it is CONSTRAINED UNPREDICTABLE which:

- The trace data is written to memory.

- The trace data is sent to an IMPLEMENTATION DEFINED trace bus.

- The trace data is written to memory and sent to an IMPLEMENTATION DEFINED trace bus.

- The Trace Buffer Unit Discards the trace data.

This also includes any trace data that might be flushed by the trace unit, for example due to a `TSB CSYNC` operation.

If the Trace Buffer Unit Discards the trace data, a trace buffer management event might be generated:

- TRBSR_EL1.EC is set to a CONSTRAINED UNPREDICTABLE choice of the following values:

  – `0x00`, other buffer management event.
  – `0x1F`, IMPLEMENTATION DEFINED buffer management event.

- If TRBSR_EL1.EC is set to `0x00` then TRBSR_EL1.BSC is set to `0b000000` to indicate that the trace buffer is not full.

It is also CONSTRAINED UNPREDICTABLE whether any of the following occur, whether or not the trace data is written to memory:

- The current write pointer and, if TRBSR_EL1.TRG is 1, the Trigger Counter are updated for the trace data.

- A trace buffer management event that would have been generated is observed and/or generated.

- A PMU event that would have been generated is generated.

I_{LLCLK} See also G3.1 *Context switching*.

E1.3 Events

Number	Mnemonic	Type	Description
0x400C	TRB_WRAP	Architectural	Trace buffer current write pointer wrapped.
0x400D	PMU_OVFS	Architectural	PMU overflow, counters accessible to EL1 and EL0.
0x400E	TRB_TRIG	Architectural	Trace buffer Trigger Event.
0x400F	PMU_HOVFS	Architectural	PMU overflow, counters reserved for use by EL2.
0x4010	TRCEXTOUT0	Microarchitectural	Trace unit external output 0.
0x4011	TRCEXTOUT1	Microarchitectural	Trace unit external output 1.
0x4012	TRCEXTOUT2	Microarchitectural	Trace unit external output 2.
0x4013	TRCEXTOUT3	Microarchitectural	Trace unit external output 3.
0x4018	CTI_TRIGOUT4	Microarchitectural	Cross-trigger Interface output trigger 4.
0x4019	CTI_TRIGOUT5	Microarchitectural	Cross-trigger Interface output trigger 5.
0x401A	CTI_TRIGOUT6	Microarchitectural	Cross-trigger Interface output trigger 6.
0x401B	CTI_TRIGOUT7	Microarchitectural	Cross-trigger Interface output trigger 7.

E1.3.1 Common microarchitectural events

0x4010, TRCEXTOUT0, Trace unit external output 0 The event is generated each time an event is signaled by the trace Unit external event 0.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[48] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

0x4011, TRCEXTOUT1, Trace unit external output 1 The event is generated each time an event is signaled by the trace Unit external event 1.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[49] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

0x4012, TRCEXTOUT2, Trace unit external output 2 The event is generated each time an event is signaled by the trace unit external event 2.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[50] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

0x4013, TRCEXTOUT3, Trace unit external output 3 The event is generated each time an event is signaled by the trace unit external event 3.

It is IMPLEMENTATION DEFINED whether this event is available as an external input to the ETE.

PMCEID0_EL0[51] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

0x4018, CTI_TRIGOUT4, Cross-trigger Interface output trigger 4 The event is generated each time an event is signaled on Cross-trigger Interface (CTI) output trigger 4.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[56] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

0x4019, CTI_TRIGOUT5, Cross-trigger Interface output trigger 5 The event is generated each time an event is signaled on CTI output trigger 5.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[57] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

0x401A, CTI_TRIGOUT6, Cross-trigger Interface output trigger 6 The event is generated each time an event is signaled on CTI output trigger 6.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[58] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

0x401B, CTI_TRIGOUT7, Cross-trigger Interface output trigger 7 The event is generated each time an event is signaled on CTI output trigger 7.

Note

CTI output triggers are input events to the PMU and trace unit.

PMCEID0_EL0[59] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_ETE is implemented.

E1.3.2 Common architectural events

0x400C, TRB_WRAP, Trace buffer current write pointer wrapped The event is generated each time the current write pointer is wrapped to the base pointer.

PMCEID0_EL0[44] reads as `0b1` if this event is implemented and `0b0` otherwise.

This event must be implemented if FEAT_TRBE is implemented.

0x400D, PMU_OVFS, PMU overflow, counters accessible to EL1 and EL0 The event is generated each time one of the following occurs:

- An event is counted by an event counter <n> and all of the following are true:
  - PMINTENSET_EL1[n] is 0b1.
  - One of the following is true:
    * Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[31:0], and either FEAT_PMUv3p5 is not implemented or PMCR_EL0.LP is 0b0.
    * Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[63:0], FEAT_PMUv3p5 is implemented, and PMCR_EL0.LP is 0b1.
  - Either EL2 is implemented and <n> is less than MDCR_EL2.HPMN, or EL2 is not implemented and <n> is in the range [0 .. (PMCR_EL0.N-1)].
- A cycle is counted by PMCCNTR_EL0, PMINTENSET_EL1[31] is 0b1, and one of the following is true:
  - Counting the cycle causes unsigned overflow of PMCCNTR_EL0[31:0] and PMCR_EL0.LC is 0b0.
  - Counting the cycle causes unsigned overflow of PMCCNTR_EL0[63:0] and PMCR_EL0.LC is 0b1.

This event cannot be counted by the PMU. PMCEID0_EL0[45] reads as 0b0. This event must be implemented if all of the following are true:

- FEAT_PMUv3 is implemented.
- FEAT_ETE is implemented.

0x400E, TRB_TRIG, Trace buffer Trigger Event The event is generated when a Trace Buffer Extension Trigger Event occurs.

It is IMPLEMENTATION DEFINED whether this event can be counted by the PMU.

PMCEID0_EL0[46] reads as 0b1 if this event is implemented and can be counted by the PMU, and 0b0 otherwise.

This event must be implemented if FEAT_TRBE is implemented.

0x400F, PMU_HOVFS, PMU overflow, counters reserved for use by EL2 The event is generated each time an event is counted by an event counter <n> and all of the following are true:

- EL2 is implemented.
- PMINTENSET_EL1[n] is 0b1.
- One of the following is true:
  - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[31:0], and either FEAT_PMUv3p5 is not implemented or MDCR_EL2.HLP is 0b0.
  - Counting the event causes unsigned overflow of PMEVCNTR<n>_EL0[63:0], FEAT_PMUv3p5 is implemented, and MDCR_EL2.HLP is 0b1.
- <n> in the range [MDCR_EL2.HPMN .. (PMCR_EL0.N-1)].

The event is not transmitted to a trace unit if SelfHostedTraceEnabled() is TRUE and TRFCR_EL2.E2TRE is 0b0.

Note

This is in addition to the rules for the export of all events to a trace unit described in "Controls to prohibit trace at Exception levels" in *Arm® Architecture Reference Manual, for A-profile architecture* [1].

This event cannot be counted by the PMU. PMCEID0_EL0[47] reads as 0b0.

This event must be implemented if all of the following are true:

- FEAT_PMUv3 is implemented.
- FEAT_ETE is implemented.
- EL2 is implemented.

Part F
The Branch Record Buffer Extension

Chapter F1
Branch Record Buffer Extension

Details of control path, either at the loop level or at the function call level, is useful when compiling and optimizing software. These directed optimizations extract information about hotspots and common control paths in the code.

FEAT_BRBE provides a mechanism for capturing control path history in a low cost manner.

FEAT_BRBEv1p1 extends FEAT_BRBE to enable branch recording at EL3.

F1.1 Branch Record Buffer Extension specification

R_{VNFCR} FEAT_BRBE is an OPTIONAL feature from Armv9.2.

I_{FKLJM} ID_AA64DFR0_EL1.BRBE indicates whether FEAT_BRBE is implemented.

R_{VYMSG} FEAT_BRBEv1p1 is an OPTIONAL feature from Armv9.3.

R_{KRBNR} If FEAT_BRBE is implemented, an Armv9.3-compliant implementation includes the FEAT_BRBEv1p1 feature.

I_{GFXKY} ID_AA64DFR0_EL1.BRBE indicates whether FEAT_BRBEv1p1 is implemented.

F1.1.1 Branch records

I_{JBTBH} Each *Branch record* consists of 3 registers:

- BRBINF<n>_EL1.
- BRBSRC<n>_EL1.
- BRBTGT<n>_EL1.

R_{GBCQW} Taken branch instructions, as defined by the section "Branches, Exception Generating and System instructions" in *Arm® Architecture Reference Manual, for A-profile architecture* [1], generate a Branch record.

R_{LFVJR} Exceptions generate a Branch record.

R_{RPVXK} A *Half-source Branch record* has BRBINF<n>_EL1.VALID == 0b10.

R_{LPLYK} A *Half-target Branch record* has BRBINF<n>_EL1.VALID == 0b01.

R_{GSMRH} A *Full Branch record* has BRBINF<n>_EL1.VALID == 0b11.

R_{MLGCF} When a Branch record is generated for any branch or exception which does not transition between a BRBE Prohibited Region and a BRBE non-Prohibited Region, the Branch record is a Full Branch record.

See F1.1.5 *Branch records for exceptions* and F1.1.6 *Branch records for exception returns* for more details on when a Half-source Branch record or a Half-target Branch record is generated.

I_{ZCHRF} When an exception, exception return instruction, or Instruction Synchronization Barrier instruction causes a Context synchronization event which synchronizes an update to one or more System registers which are indirectly read when generating a Branch record, the synchronization of those register updates occurs before the registers are indirectly read. Such order is generally consistent with indirect reads of System registers performed by events which cause a Context synchronization event.

R_{CBHRY} The reason for the Branch record is captured in BRBINF<n>_EL1.TYPE.

F1.1.2 Cycle counting

I_{RRBFF} Each Branch record contains a cycle count value which is representative of the time taken between each Branch record being generated. The cycle count value can be used to determine the relative performance of the program between each Branch record. For large cycle count values, the value stored in each Branch record is encoded to use less storage, with a small loss of precision in the value.

R_{HQXNW} The size of the cycle counter used to generate cycle count values is IMPLEMENTATION DEFINED, from one of the sizes indicated in BRBIDR0_EL1.CC.

R_{KVRBB} Each Branch record contains a cycle count value which indicates the number of *Processing Element* (PE) clock cycles that occurred between the previous Branch record being generated and this Branch record being generated.

R_{SBXCF} In a multithreaded implementation, the cycle counter only counts cycles on which the thread was active.

Note: This is identical to how the CPU_CYCLES PMU event counts cycles when PMEVTYPER<n>_EL0.MT==0b0. For more information, see the section *Cycle event counting on multithreaded implementations* in *Arm® Architecture Reference Manual, for A-profile architecture* [1].

R_{PGXMB} For the purposes of the cycle count, a Branch record is generated only when the corresponding branch instruction or exception is guaranteed to be architecturally executed and the target address has been calculated. Arm recommends that the Branch record is generated as soon after this point as possible.

I_{KBDLR} When a branch target address contains an address tag, the target address captured in the Branch record is the virtual address with the address tag removed.

R_{MJDLC} The cycle count value in a Branch record is Branch Cycle Count Unknown when any of the following are true:

- BRBCR_EL2.CC == 0b0, if EL2 is implemented.
- BRBCR_EL1.CC == 0b0.
- This is the first Branch record after the PE exited a BRBE Prohibited Region.
- This is the first Branch record after cycle counting has been enabled.
- This is the first Branch record after BRBFCR_EL1.PAUSED is cleared from 0b1 to 0b0.
- This is the first Branch record after execution of a BRB IALL instruction.

Note: This applies even when EL2 is disabled in the current Security state.

R_{PBJTJ} When the cycle count value in a Branch record is *Branch Cycle Count Unknown*:

- BRBINF<n>_EL1.CCU has the value 0b1.
- BRBINF<n>_EL1.CC contains a value which is all zeros.

The number of cycles indicated by this Branch record is UNKNOWN.

R_{XGSSZ} If the cycle count value in a Branch record would exceed the maximum value of the cycle counter, then:

- BRBINF<n>_EL1.CCU has the value 0b0.
- BRBINF<n>_EL1.CC contains a value which is all ones.

R_{JZWPG} If the cycle count value in a Branch record is not UNKNOWN and would not exceed the maximum value of the cycle counter, then:

- BRBINF<n>_EL1.CCU has the value 0b0.
- BRBINF<n>_EL1.CC contains the cycle count value, encoded as defined in BRBINF<n>_EL1.CC.

F1.1.3 Mispredicted branches

I_{QHGFW} Each Branch record generated for a branch instruction contains an indication of whether the branch was correctly or incorrectly predicted by the PE. Branch prediction behavior is IMPLEMENTATION DEFINED and this is an indication of whether such prediction succeeded, or not.

R_{XHWRB} For a Branch record for a branch instruction one of the following occurs:

- If EL2 is implemented and BRBCR_EL2.MPRED == 0b0 then BRBINF<n>_EL1.MPRED has the value 0b0.
- else if BRBCR_EL1.MPRED == 0b0 then BRBINF<n>_EL1.MPRED has the value 0b0.
- otherwise:
  - BRBINF<n>_EL1.MPRED has the value 0b0 for a correctly predicted branch.
  - BRBINF<n>_EL1.MPRED has the value 0b1 for an incorrectly predicted branch.

Note: This applies even when EL2 is disabled in the current Security state.

R_{DHNPJ} For a Branch record for an exception BRBINF<n>_EL1.MPRED has the value 0b0.

R_{LBRGV} An incorrectly predicted branch is when any of the following is true:

- The direction of a conditional branch was incorrectly predicted at least once during the execution of the instruction.
- The target of a branch was incorrectly predicted at least once during the execution of the instruction.
- The branch was not predicted by a branch predictor.

R_{RDLQF} A correctly predicted branch is one that is not incorrectly predicted.

F1.1.4 BRBE Prohibited regions

I_{NVWPM} An executable program might contain regions of code that are prohibited to generate Branch records, and these regions are called *BRBE Prohibited Regions*. These regions are usually associated with a different Security state or Exception level.

I_{DMPQZ} BRBE Prohibited Regions are controlled by the following:

- BRBCR_EL1.E0BRE.
- BRBCR_EL1.E1BRE.
- BRBCR_EL2.E0HBRE.
- BRBCR_EL2.E2BRE.
- MDCR_EL3.SBRBE.

I_{HPZWM} While executing outside a BRBE Prohibited Region, Branch records might not be generated because the Branch Record Buffer Extension has a number of filtering functions.

R_{FHGJN} Execution in AArch32 state is a BRBE Prohibited Region.

R_{LPYBQ} Execution in Debug state is a BRBE Prohibited Region.

R_{JWWFY} When FEAT_BRBEv1p1 and EL3 are implemented:

- When MDCR_EL3.{E3BREC,E3BREW} == 0b01 or MDCR_EL3.{E3BREC,E3BREW} == 0b10, *self-hosted EL3 branch recording* is *enabled*.

- When MDCR_EL3.{E3BREC,E3BREW} == 0b00 or MDCR_EL3.{E3BREC,E3BREW} == 0b11, self-hosted EL3 branch recording is *disabled*.

R_{GLKGW} Execution at EL3 is a BRBE Prohibited Region when any of the following are true:

- FEAT_BRBEv1p1 is not implemented.
- self-hosted EL3 branch recording is disabled.

R_{SFZQD} Execution at EL2 is a BRBE Prohibited Region when any of the following are true:

- BRBCR_EL2.E2BRE == 0b0.
- MDCR_EL3.SBRBE == 0b00.
- MDCR_EL3.SBRBE == 0b01 and the PE is in Secure state.

R_{YPHYG} Execution at EL1 is a BRBE Prohibited Region when any of the following are true:

- BRBCR_EL1.E1BRE == 0b0.
- MDCR_EL3.SBRBE == 0b00.
- MDCR_EL3.SBRBE == 0b01 and the PE is in Secure state.

R_{DBPCP} Execution at EL0 is a BRBE Prohibited Region when any of the following are true:

- EL2 is disabled in the current security state or HCR_EL2.TGE == 0b0, and BRBCR_EL1.E0BRE == 0b0.
- EL2 is enabled in the current security state and HCR_EL2.TGE == 0b1, and BRBCR_EL2.E0HBRE == 0b0.
- MDCR_EL3.SBRBE == 0b00.
- MDCR_EL3.SBRBE == 0b01 and the PE is in Secure state.

R$_{\text{YGGSC}}$ While the PE is executing code from a BRBE Prohibited Region, no data is captured in Branch records that might provide information about execution in the BRBE Prohibited Region.

F1.1.5 Branch records for exceptions

R$_{\text{YSKQK}}$ When an exception is taken from a BRBE Prohibited Region to a BRBE Prohibited Region, no Branch record is generated.

R$_{\text{KRJQC}}$ When an exception is taken from a BRBE non-Prohibited Region, or an exception is taken to a BRBE non-Prohibited Region:

- If the target Exception level is EL1, a Branch record is generated only if BRBCR_EL1.EXCEPTION == 0b1.
- If the target Exception level is EL2, a Branch record is generated only if BRBCR_EL2.EXCEPTION == 0b1.
- If the target Exception level is EL3, a Branch record is generated only if FEAT_BRBEv1p1 is implemented and self-hosted EL3 branch recording is enabled.

R$_{\text{YBJDJ}}$ When a Branch record is generated for an exception:

- If the exception is taken from a BRBE Prohibited Region, then a Half-target Branch record is generated.
- If the exception is taken from a BRBE non-Prohibited Region to a BRBE Prohibited Region, then a Half-source Branch record is generated.
- If the exception is taken from a BRBE non-Prohibited Region to a BRBE non-Prohibited Region, then a Full Branch record is generated.

R$_{\text{LLCTG}}$ When entering Debug state:

- If the entry is from a BRBE Prohibited Region, no Branch record is generated.
- If the entry is from a BRBE non-Prohibited Region, then a Half-source Branch record is generated.

I$_{\text{MZNRY}}$ When a Half-source Branch record or a Full Branch record is generated for an Illegal Execution state exception, the source information in the Branch record indicates where the exception was taken from, in the same way as all other exceptions.

R$_{\text{BZCRW}}$ A Branch record for an exception which contains a valid source address has the source address set to the preferred exception return address for the exception.

R$_{\text{FYLTC}}$ A Branch record for an exception which contains a valid target address has the target address set to the address of the exception vector.

F1.1.6 Branch records for exception returns

R$_{\text{LMXHS}}$ When an exception return instruction is executed in a BRBE Prohibited Region and branches to a BRBE Prohibited Region, no Branch record is generated.

R$_{\text{ZSHDL}}$ When an exception return instruction is executed in a BRBE non-Prohibited Region, or an exception return instruction branches to a BRBE non-Prohibited Region:

- If the exception return instruction is executed at EL3, a Branch record is generated only if FEAT_BRBEv1p1 is implemented and self-hosted EL3 branch recording is enabled.
- If the exception return instruction is executed at EL2, a Branch record is generated only if BRBCR_EL2.ERTN == 0b1.
- If the exception return instruction is executed at EL1, a Branch record is generated only if BRBCR_EL1.ERTN == 0b1.

R$_{\text{ZTGMW}}$ When a Branch record is generated for an exception return instruction:

- If the exception return instruction is executed in a BRBE Prohibited Region then a Half-target Branch record is generated.

- If the exception return instruction is executed in a BRBE non-Prohibited Region and branches to a BRBE Prohibited Region then a Half-source Branch record is generated.
- If the exception return instruction is executed in a BRBE non-Prohibited Region and branches to a BRBE non-Prohibited Region then a Full Branch record is generated.

R_{RBCFP} When exiting from Debug state:

- If the exit is to a BRBE Prohibited Region, no Branch record is generated.
- If the exit is to a BRBE non-Prohibited Region, then a Half-target Branch record is generated.

I_{NGWPR} When a Half-target Branch record or a Full Branch record is generated for an exception return instruction which is an illegal return or a legal return which sets PSTATE.IL to 0b1, the target information in the Branch record indicates the target of the branch:

- BRBTGT<n>_EL1.ADDRESS contains the target of the branch.
- BRBINF<n>_EL1.EL contains the value that is loaded in to PSTATE.EL.

I_{JCYHD} When a Half-target Branch record or a Full Branch record is generated for an exception return instruction which is an illegal return or a legal return which sets PSTATE.IL to 0b1, for the purposes of determining whether the target is a BRBE Prohibited Region the value that is loaded in to PSTATE.EL is used as the target Exception level.

Note that PSTATE.EL is unchanged on an illegal return, so the current Exception level is the target of the illegal return, regardless of where the return was attempting to return to.

F1.1.7 Transactional Memory Extension

R_{GVCJH} When an entire transaction is executed in a BRBE non-Prohibited Region and the transaction fails or is canceled then BRBFCR_EL1.LASTFAILED is set to 0b1.

R_{JMSZF} When an entire transaction is executed in a BRBE Prohibited Region and the transaction fails or is canceled then BRBFCR_EL1.LASTFAILED is unchanged.

R_{CBTBH} When a transaction is executed partially in a BRBE Prohibited Region and partially in a BRBE non-Prohibited Region and the transaction fails or is canceled then it is CONSTRAINED UNPREDICTABLE whether BRBFCR_EL1.LASTFAILED is set to 0b1 or is unchanged.

R_{KBSZM} When a Branch record is generated, other than via the injection mechanism, the value of BRBFCR_EL1.LASTFAILED is copied to the LASTFAILED field in the Branch record and BRBFCR_EL1.LASTFAILED is set to 0b0.

I_{HJZWG} When a transaction fails or is canceled, a Branch record is not generated.

I_{JBPHS} When a transaction fails or is canceled, Branch records generated in the transaction are not removed from the Branch record buffer.

I_{TFKNW} Attempting to execute the BRB IALL or BRB INJ instructions in Transactional state results in the transaction failing with ERR cause.

F1.1.8 PE Speculation

R_{KXTKS} The Branch records only contain information for a branch, exception, or exception return that is architecturally executed.

F1.1.9 Branch record filtering

X_{NZWBP} For Branch records generated outside a BRBE Prohibited Region it is useful to reduce the number of records that are generated to match their use. Table F1.1 lists the some different use cases.

Table F1.1: Example use cases for filtering

Use case	Description
Control path	• All branches • Subroutine returns • Exceptions • Exception returns
Call path	• Branch with link instructions • Subroutine returns
Kernel Calls	• Exceptions • Exception returns

F1.1.9.1 Filtering on type

I_{FSNVG} The Branch records can be filtered by independently enabling the generation of the following types:

- Exception.
- Exception return.
- Direct Branch with link.
- Indirect Branch with link.
- Return from subroutine.
- Indirect Branches.
- Conditional Direct Branches.
- Unconditional Direct Branches.

R_{LYGJZ} Control of when Branch records for exceptions are generated is controlled by BRBCR_EL1.EXCEPTION and BRBCR_EL2.EXCEPTION. See F1.1.5 *Branch records for exceptions* for details.

Table F1.2: Exception mapping for exceptions taken to AArch64 state

Reason	Type
Branch Target exception	Inst fault
Breakpoint	Inst debug
FIQ	FIQ
HVC	Call
Halting debug event	Debug halt
IRQ	IRQ
Illegal execution state	Trap
Instruction Abort	Inst fault
Instruction or event trapped by a control bit	Trap
MemCopy or MemSet	Trap

Reason	Type
Misaligned PC	Alignment
PAC Fail	Data fault
SError interrupt	System Error
SMC	Call
SVC	Call
Software Breakpoint Instruction	Inst debug
Software Step	Inst debug
Stack Pointer Misalignment	Alignment
Synchronous Data Abort	Data fault
UNDEFINED instruction	Trap
Watchpoint	Data debug

R_{RCGVB} Control of when Branch records for exception return instructions are generated is controlled by BRBCR_EL1.ERTN and BRBCR_EL2.ERTN. See F1.1.6 *Branch records for exception returns* for details.

Table F1.3: A64 return from exception instructions

Instruction	Description
ERET	Return From Exception.
ERETAA	Authenticate and Exception return.
ERETAB	Authenticate and Exception return.

R_{RBDXK} Branch records for Direct branch with link instructions are only generated when the instruction is executed in a BRBE non-Prohibited Region and if any of the following are true:

- BRBFCR_EL1.DIRCALL == 0b1 and BRBFCR_EL1.EnI == 0b0.
- BRBFCR_EL1.DIRCALL == 0b0 and BRBFCR_EL1.EnI == 0b1.

Table F1.4: A64 direct branch with link instructions

Instruction	Description
BL	Branch with link.

R_{VBGTZ} Branch records for Indirect branch with link instructions are only generated when the instruction is executed in a BRBE non-Prohibited Region and if any of the following are true:

- BRBFCR_EL1.INDCALL == 0b1 and BRBFCR_EL1.EnI == 0b0.
- BRBFCR_EL1.INDCALL == 0b0 and BRBFCR_EL1.EnI == 0b1.

Table F1.5: A64 indirect branch with link instructions

Instruction	Description
BLR	Branch with link to register.
BLRAA	Authenticate and branch with link.
BLRAAZ	Authenticate and branch with link.
BLRAB	Authenticate and branch with link.
BLRABZ	Authenticate and branch with link.

R_{CKNBH} Branch records for return from subroutine instructions are only generated when the instruction is executed in a BRBE non-Prohibited Region and if any of the following are true:

- BRBFCR_EL1.RTN == 0b1 and BRBFCR_EL1.EnI == 0b0.
- BRBFCR_EL1.RTN == 0b0 and BRBFCR_EL1.EnI == 0b1.

Table F1.6: A64 return from subroutine instructions

Instruction	Description
RET	Return From subroutine.
RETAA	Authenticate and function return.
RETAB	Authenticate and function return.

R_{KKLDV} Branch records for indirect branch instructions, unless covered by other rules, are only generated when the instruction is executed in a BRBE non-Prohibited Region and if any of the following are true:

- BRBFCR_EL1.INDIRECT == 0b1 and BRBFCR_EL1.EnI == 0b0.
- BRBFCR_EL1.INDIRECT == 0b0 and BRBFCR_EL1.EnI == 0b1.

Table F1.7: A64 indirect branch instructions

Instruction	Description
BR	Branch to register.
BRAA	Authenticate and branch.
BRAAZ	Authenticate and branch.
BRAB	Authenticate and branch.
BRABZ	Authenticate and branch.

R_{BBNSZ} Branch records for conditional direct branch instructions, unless covered by other rules, are only generated when the instruction is executed in a BRBE non-Prohibited Region and if any of the following are true:

- BRBFCR_EL1.CONDDIR == 0b1 and BRBFCR_EL1.EnI == 0b0.
- BRBFCR_EL1.CONDDIR == 0b0 and BRBFCR_EL1.EnI == 0b1.

Table F1.8: A64 conditional direct branch instructions

Instruction	Description
B.cond	Conditional Branch.
BC.cond	Branch Consistent conditionally.
CBZ or CBNZ	Compare with zero and branch.
TBZ or TBNZ	Test and branch.

Note: BC.cond and B.cond instructions with the AL or NV condition code are considered conditional.

R_{FJYVT} [Branch records](#) for unconditional direct branch instructions, unless covered by other rules, are only generated when the instruction is executed in a [BRBE non-Prohibited Region](#) and if any of the following are true:

- BRBFCR_EL1.DIRECT == 0b1 and BRBFCR_EL1.EnI == 0b0.
- BRBFCR_EL1.DIRECT == 0b0 and BRBFCR_EL1.EnI == 0b1.

Table F1.9: A64 unconditional direct branch instructions

Instruction	Description
B	Unconditional Branch.

R_{FJYDC} It is IMPLEMENTATION DEFINED whether [Branch records](#) are generated for the following instructions when the instruction is executed in a [BRBE non-Prohibited Region](#) and if any of the following are true:

- BRBFCR_EL1.DIRECT == 0b1 and BRBFCR_EL1.EnI == 0b0.
- BRBFCR_EL1.DIRECT == 0b0 and BRBFCR_EL1.EnI == 0b1.

Table F1.10: Optional A64 direct branch instructions

Instruction	Description
ISB	Instruction Synchronization Barrier.

S_{XZRTW} Writing a value of 0b0000_0001 to the filter controls, BRBFCR_EL1<23:16>, ensures [Branch records](#) are generated for all branch instructions.

F1.1.10 Branch record buffer operation

R_{LKWNB} The Branch Record Buffer Extension operation is controlled by the BRBCR_EL1, BRBCR_EL2, and BRBFCR_EL1 registers.

R_{PYBRZ} Generation of [Branch records](#) is *Paused* when BRBFCR_EL1.PAUSED == 0b1.

R_{YDZNK} When generation of [Branch records](#) is [Paused](#), [Branch records](#) are not generated.

R_{NXCWF} If EL2 is implemented, a [BRBE freeze event](#) occurs when all of the following are true:

- BRBCR_EL1.FZP is 0b1.
- Generation of [Branch records](#) is not [Paused](#).

- PMOVSCLR_EL0[(MDCR_EL2.HPMN-1):0] is non-zero.
- The PE is in a BRBE non-Prohibited Region.

R_{GXGWY} If EL2 is implemented, a BRBE freeze event occurs when all of the following are true:

- BRBCR_EL2.FZP is 0b1.
- Generation of Branch records is not Paused.
- PMOVSCLR_EL0[(PMCR_EL0.N-1):MDCR_EL2.HPMN] is non-zero.
- The PE is in a BRBE non-Prohibited Region.

Note: This applies even when EL2 is disabled in the current Security state.

R_{PKTXQ} If EL2 is not implemented, a BRBE freeze event occurs when all of the following are true:

- BRBCR_EL1.FZP is 0b1.
- Generation of Branch records is not Paused.
- PMOVSCLR_EL0[(PMCR_EL0.N-1):0] is non-zero.
- The PE is in a BRBE non-Prohibited Region.

R_{BHYTD} On a *BRBE freeze event*:

- BRBFCR_EL1.PAUSED is set to 0b1.
- The current timestamp is captured in BRBTS_EL1.

R_{QKQZL} The source of value of the timestamp captured in BRBTS_EL1 is selected by the combination of programming of BRBCR_EL2.TS and BRBCR_EL1.TS. See Table F1.11 and BRBETimeStamp().

Table F1.11: Captured timestamp

BRBCR_EL2.TS	BRBCR_EL1.TS	Captured timestamp
0b00 (delegate)	0b01 (virtual)	PhysicalCountInt() - CNTVOFF_EL2
	0b10 (offset physical)	PhysicalCountInt() - physical offset
	0b11 (physical)	PhysicalCountInt()
0b01 (virtual)	0bxx	PhysicalCountInt() - CNTVOFF_EL2
0b10 (offset physical)	0bxx	PhysicalCountInt() - physical offset
0b11 (physical)	0bxx	PhysicalCountInt()

If any of the following are true, the physical offset is zero, otherwise the physical offset is the value of CNTVOFF_EL2:

- EL3 is implemented and SCR_EL3.ECVEn == 0b0.
- EL2 is implemented and CNTHCTL_EL2.ECV == 0b0.

R_{GWMZV} When a valid Branch record is captured in the Branch record buffer storage, the BRB_FILTRATE event is generated.

R_{GMCHN} When BRB_FILTRATE is generated for an exception or an exception return, it is an Exception-related event.

I_{WGZSC} It is CONSTRAINED UNPREDICTABLE whether a BRB_FILTRATE event is generated after a BRB INJ causes a Branch record to be injected.

X_{SJDRW} Branch record buffer storage injection occurs during software context switch events, which occur at a higher Exception level than the one being profiled. It is expected that the Branch record buffer storage is in a BRBE Prohibited Region during these events, and therefore Branch record generation is prohibited.

In the use case in which the PMU is not prohibited during software context switch events, it is expected that event counters set to count the BRB_FILTRATE event are prohibited from counting during these events.

Additionally, preventing the generation of BRB_FILTRATE on a Branch record injection would require hardware to suppress the event.

I_{PGDQV} The following definition is taken from *Arm® Architecture Reference Manual, for A-profile architecture* [1]:

The PMU must filter [Exception-related events] *according to the Exception level in which the event occurred. . . . The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions at EL0.*

R_{KSKLG} If a BRB_FILTRATE event causes unsigned overflow of the event counter counting that event and this in turn causes a BRBE freeze event then:

- The Branch record for the operation that generated the BRB_FILTRATE event will be generated and captured in the Branch record buffer.
- It is CONSTRAINED UNPREDICTABLE whether the Branch Record Buffer Extension generates Branch records for other operations in program order after the operation that generated the BRB_FILTRATE event that would otherwise be generated when generation of Branch records is not Paused.

Note: Arm recommends that implementations minimize capture of additional branches.

I_{RMGDV} The architecture does not define when PMU events are counted relative to the instructions that caused the event. Events generated by an instruction might be counted before or after the instruction becomes architecturally executed, and events might be counted for operations that do not become architecturally executed, meaning events can be counted speculatively and/or out-of-order with respect to the simple sequential execution of the program. Events might also be counted simultaneously by other event counters when the overflow occurs, including events from different instructions. In addition, multiple instances of an event might occur simultaneously, meaning that an event counter unsigned overflow can yield a nonzero value in the event counter.

Furthermore, the Branch records are generated only for architecturally executed operations R_{KXTKS}.

These properties mean that, unless otherwise stated, on a BRBE freeze event, it is CONSTRAINED UNPREDICTABLE whether the branches that define the basic block containing the instruction causing that event are captured in the Branch record buffer.

An exception to this relaxation applies for the BRB_FILTRATE event.

S_{JCHLT} If a direct read of PMOVSCLR_EL0 returns a non-zero value for a subset of the overflow flags, such that one of R_{NXCWF}, R_{GXGWY}, or R_{PKTXQ} means that a BRBE freeze event should occur, then a read of BRBFCR_EL1 ordered after the read of PMOVSCLR_EL0 will return BRBFCR_EL1.PAUSED == 0b1.

Note: Direct reads of System registers require explicit synchronization for following direct reads of other System registers to be ordered after the first direct read.

S_{SRJND} If a direct read of BRBFCR_EL1.PAUSED returns the value 0b1, then no operations ordered after the direct read will generate further Branch records until BRBFCR_EL1.PAUSED is cleared by software. The subsequent operations can be ordered by a Context synchronization event.

F1.1.11 Branch record buffer

I_{FBHCC} The Branch record buffer can contain:

- 8 Branch records.
- 16 Branch records.
- 32 Branch records.

- 64 Branch records.

This is known as the *Branch record buffer storage*.

R_{KSLSM} The Branch record buffer storage has a maximum number of Branch records as defined by BRBIDR0_EL1.NUMREC.

I_{PPBZP} The Branch record buffer provides System registers to access the Branch records stored in the Branch record buffer storage. These System registers provide access to up to 32 Branch records without the need for explicit synchronization between each System register read. When more than 32 Branch records are implemented, the Branch record buffer provides a banking mechanism to provide access to multiple banks, each bank containing up to 32 Branch records. BRBFCR_EL1.BANK controls which bank is currently selected, and updates to BRBFCR_EL1.BANK require explicit synchronization before accessing the bank.

R_{WLSWP} Accessing Branch records 0 to 31 is performed by setting BRBFCR_EL1.BANK to 0b00.

R_{WRJLW} Accessing Branch records 32 to 63 is performed by setting BRBFCR_EL1.BANK to 0b01.

R_{TJGLK} The Branch record with index 0 is the youngest captured branch.

R_{HTRNR} The Branch record with index n is younger than Branch record with index n+1.

R_{DTPDK} On the generation of a new Branch record and if the Branch record buffer storage is full then the oldest Branch record is lost.

X_{QMKLH} To ensure efficiency when accessing the buffer, the buffer must only contain a contiguous set of valid Branch records, with no invalid Branch records between any two valid Branch records. A valid Branch record is one with BRBINF<n>_EL1.VALID != 0b00, and an invalid Branch record is one with BRBINF<n>_EL1.VALID == 0b00.

R_{SQLCX} When the buffer contains M valid Branch records, where M > 0 and M is less than the maximum number of Branch records, all of the following are true:

- Branch records with index 0 to M-1 are all valid.
- All other Branch records are invalid.

R_{PGDLX} The creation of a Branch record is considered an indirect write to BRBTGT<n>_EL1, BRBSRC<n>_EL1 and BRBINF<n>_EL1, and therefore requires explicit synchronization before being read.

I_{SFFNF} The generation of Branch records performs indirect reads and indirect writes of System registers.

I_{KFYTV} *Arm® Architecture Reference Manual, for A-profile architecture* [1] defines the synchronization requirements for direct reads, direct writes, indirect reads and indirect writes of System registers made by instructions and external agents.

F1.1.12 Invalidating the Record Buffer

R_{LLHYN} Execution of BRB IALL causes all Branch records to be invalidated.

R_{PFRNW} A Branch record, **R**, is invalidated by the instruction BRB IALL, **W**, if all of the following are true:

- **R** is caused by a branch operation or exception, **B**.
- **B** is either:
  – in program order before a *Context Synchronization Event*, **CSE**.
  – is the *Context Synchonization Event*.
- **CSE** is in program order before **W**.

R_{LWPKR} A Branch record **R** is not invalidated by the instruction BRB IALL, **W**, if all of the following are true:

- **R** is caused by a branch operation or exception, **B**.
- **B** is in program order after a *Context Synchronization Event*, **CSE**
- **CSE** is in program order after **W**.

R$_{\text{WMZKF}}$ It is CONSTRAINED UNPREDICTABLE if a Branch record **R** is invalidated by the instruction BRB IALL, **W**, if all of the following are true:

- **CSE$_1$** is in program order before **W**.
- **R** is caused by a branch operation or exception, **B**.
- **B** is in program order after a *Context Synchronization Event*, **CSE$_1$**
- **B** is either:
    - in program order before a *Context Synchronization Event*, **CSE$_2$**.
    - is the *Context Synchonization Event*, **CSE$_2$**.
- **CSE$_2$** is in program order after **W** and there are no other CSEs between **CSE$_1$** and **CSE$_2$**.

If a Branch record is invalidated, all older Branch records are invalidated.

R$_{\text{JSCWK}}$ When a Branch record has been invalidated, it remains invalid until it is overwritten by any of the following:

- A new Branch record is created.
- A Branch record is injected using the BRB INJ instruction.

F1.1.13 Programmers Model

R$_{\text{BNGTH}}$ Reads from an unimplemented Branch record return the value zero.

R$_{\text{PKZCF}}$ All Branch records captured while generation of Branch records is not Paused, must represent a continuous block of execution for all BRBE non-Prohibited Regions.

I$_{\text{XSBPR}}$ The captured Branch records might not represent a continuous block if generation of Branch records is Paused at any time. To avoid this non-continuous nature, the BRB IALL instruction can be used to invalidate all Branch records while generation is Paused.

R$_{\text{PMRRL}}$ If a Branch record cannot be captured for a branch instruction or exception that is not prohibited and has been selected to generate a record, then all the Branch records must be invalidated. The reasons for a PE being unable to capture a Branch record are IMPLEMENTATION DEFINED and Arm recommends that such reasons are rare.

I$_{\text{QJFSV}}$ When a process is migrated to a PE with a smaller number of Branch records implemented then the information from the older Branch records will be lost.

I$_{\text{BDKJJ}}$ When FEAT_BRBE is implemented, the following fields are added to System registers to control access to the Branch record buffer functionality:

- When EL2 is implemented:
    - HDFGRTR_EL2.nBRBIDR.
    - HDFGRTR_EL2.nBRBCTL.
    - HDFGWTR_EL2.nBRBCTL.
    - HDFGRTR_EL2.nBRBDATA.
    - HDFGWTR_EL2.nBRBDATA.
    - HFGITR_EL2.nBRBINJ.
    - HFGITR_EL2.nBRBIALL.

- When EL3 is implemented
    - MDCR_EL3.SBRBE.

R$_{\text{XVQKS}}$ The Branch record buffer registers are:

- BRBSRCINJ_EL1.
- BRBTGTINJ_EL1.
- BRBINFINJ_EL1.
- BRBCR_EL1.
- BRBCR_EL2.
- BRBCR_EL12.

- BRBFCR_EL1.
- BRBIDR0_EL1.
- BRBSRC<n>_EL1.
- BRBTGT<n>_EL1.
- BRBINF<n>_EL1.
- BRBTS_EL1.

S_{GCYTK} When self-hosted EL3 branch recording is enabled, the following registers must be programmed:

- BRBCR_EL1 and BRBCR_EL2. Software must program these registers to control the following:
  - Recording of exceptions taken to EL1 and EL2.
  - Recording of exception returns from EL1 and EL2.
  - Branch recording at EL0, EL1, and EL2.
  - Occurrence of BRBE freeze events on PMU overflows.
  - Timestamp source.
  - Misprediction information.
  - Cycle count information.
- BRBFCR_EL1. Software must program this register to control the following:
  - Selection of Branch Record Buffer bank.
  - Selection of branch types to record at all Exception levels.
  - Pausing of Branch recording.
  - If FEAT_TME is implemented, recording of transaction failure information.

Software should also consider how MDCR_EL3.SBRBE should be programmed to either allow or prohibit access to the captured branch records from lower Exception levels. Even if self-hosted EL3 branch recording is not being used, software should consider whether a `BRB IALL` instruction should be executed before executing software at lower Exception levels.

S_{YLMQQ} Software must invalidate the Branch records after a PE reset to ensure that details of execution before the reset event are not leaked.

X_{FPJPQ} MDCR_EL3.E3BREC resets to `0b0` on a Cold reset, and MDCR_EL3.E3BREW resets to `0b0` on a Warm reset. This allows software to program MDCR_EL3.E3BREC and MDCR_EL3.E3BREW such that the BRBE continues recording after Warm reset, or stops recording at Warm reset.

I_{NZMWQ} When FEAT_BRBEv1p1 is implemented, BRBE control fields reset to architecturally UNKNOWN values on a Cold reset and are unchanged on a Warm reset.

R_{MPTNR} Execution of `BRB IALL` is unchanged in Debug state.

R_{BRFKL} Execution of `BRB INJ` is unchanged in Debug state.

F1.1.13.1   Manual injection of Branch records

I_{DXNLX} The Branch Record Buffer Extension supports the ability to manually create Branch records and inject them in to the Branch record buffer storage. The primary purpose of the injection functionality is to support the restore of the Branch record buffer storage contents, particularly during software context switch events, including migration of software between PEs. The Branch record buffer storage contents are read out using direct reads of BRBSRC<n>_EL1, BRBTGT<n>_EL1, and BRBINF<n>_EL1.

R_{FYXNL} The *Branch record injection data registers* are:

- BRBSRCINJ_EL1.
- BRBTGTINJ_EL1.
- BRBINFINJ_EL1.

I_{FJHMP} Branch record injection consists of creating a single Branch record using direct writes to the Branch record injection data registers, then injecting the record in to the Branch record buffer storage using `BRB INJ`. This process injects a single Branch record as the youngest entry in the Branch record buffer storage. This process is repeated for each Branch record to be added to the Branch record buffer storage.

I_{BCZRK} When `BRB INJ` is executed outside of a BRBE Prohibited Region, it is CONSTRAINED UNPREDICTABLE whether a Branch record is injected.

R_{XVDNN} When `BRB INJ` is executed inside a BRBE Prohibited Region, the contents of the Branch record injection data registers are used to create a Branch record which is added to the Branch record buffer storage as the youngest entry.

R_{PWKFJ} Execution of `BRB INJ` does not require explicit synchronization to use the result of direct writes to the Branch record injection data registers in program order before `BRB INJ`.

I_{TVDMK} The creation of a Branch record as a result of execution of `BRB INJ` does not use the result of direct writes to the Branch record injection data registers in program order after `BRB INJ`.

Note: Explicit synchronization is not required to ensure this ordering.

R_{LKDTJ} After the execution of `BRB INJ`, the contents of the Branch record injection data registers are UNKNOWN.

I_{CPBKF} Changes to the BRB registers are subject to the rules for synchronization for system registers (see "Synchronization requirements for AArch64 System registers" in *Arm® Architecture Reference Manual, for A-profile architecture* [1]).

F1.2 Events

F1.2.1 Common architectural events

0x811F, BRB_FILTRATE, Branch Record captured The counter counts each branch record captured in the branch record buffer. Branch records that are not captured because they are removed by filtering are not counted.

When BRB_FILTRATE is generated for an exception or an exception return, it is an Exception-related event.

This event must be implemented if FEAT_BRBE is implemented.

Part G
Appendixes

Chapter G1
Synchronization requirements for System registers

This document uses the following terms to describe the effects of a direct read, direct write, indirect read, or indirect write of a System register:

Program order The sequence in which instructions are executed in a simple sequential execution of the program.

Reads-from A direct read or indirect read R_2 *Reads-from* a direct write or indirect write W_1 to the same System register if and only if R_2 takes its data from W_1.

Coherence order A *Coherence order* relation for each System register in the *Processing Element* (PE) provides a total order on all direct writes and indirect writes from all agents to that System register, starting with a notional write of the reset value.

Coherence-after A direct write or indirect write W_2 to a System register is *Coherence-after* another direct write or indirect write W_1 to the same System register if and only if W_2 is sequenced after W_1 in the *Coherence order* of the System register.

A direct write or indirect write W_2 to a System register is *Coherence-after* a direct read or indirect read R_1 to the same System register if and only if R_1 *Reads-from* another direct write or indirect write W_3 to the same System register and W_2 is Coherence-after W_3.

The rules for synchronization are described using graphs containing nodes and arcs as follows:

- Each node denotes an operation. By convention:
    - When referring to a System register, **R**, **W**, and **RW** denote a direct read, direct write, and direct read or direct write respectively.
    - When referring to a memory access, **R**, **W**, and **RW** denote a read, write, and read or write respectively.
    - **r**, **w**, and **rw** denote an indirect read, indirect write, and indirect read or indirect write of a System register respectively.

- **CSE**, **TSB**, and **DSB** denote a Context synchronization event, `TSB CSYNC` operation, and a Data synchronization barrier `DSB` respectively.

- An edge denotes a relationship between the operation at the arrow head (the pointy end) and the operation at the arrow tail (the other end):

  - A solid arrow denotes program-order.

  - A dotted arrow denotes a read/write ordering.

  - A dashed arrow indicates the generation of an operation by another operation.

The label on the arrow defines the relationship as follows:

Table G1.1: Labels for ordering diagrams

Notation	Name	Description
po	program-order	*head* is in **program order** after *tail*.
rf	Reads-from	*tail* **Reads-from** *head*.
co	Coherence-after	*head* is **Coherence-after** *tail*.
fr	from-read	As *co*, except that the operation at *head* is a read.
ob	Observed-by	*tail* is **Observed-by** *head*. Only applies for different Observers.
tb	traced-by	*head* is the Trace operation for the instruction at *tail*.
gb	generated by	*head* is an operation generated by the instruction at *tail*.
seo	speculative execution-order	The PE speculated that the instruction at *head* was executed after *tail*, but the instruction was later Canceled, Transaction-failed, or Transaction-canceled. An seo arrow might be paired with a *can* arrow that shows this.
can	canceled	The instruction at *tail* was Canceled when the instruction at *head* was Resolved, or the Transaction containing *tail* Failed or was Canceled.

For example, some of the primary synchronization requirements for System registers are described by *Arm® Architecture Reference Manual, for A-profile architecture* [1] as follows:

- *Direct writes to [System] registers are not allowed to affect any instructions appearing in program order before the direct write.*

- *All direct writes to a register occur in program order with respect to all direct reads to the same register using the same encoding.*

- *Direct writes [to System registers] require synchronization before software can rely on the effects of changes to the System registers to affect instructions appearing in program order after the direct write.*

These would be described in this document as follows:

- A direct write W_2 to a System register made by an instruction **B** is Coherence-after a direct read, indirect read, direct write, or indirect write RW_1 of the same System register made by an instruction **A** (and hence **A** will read the old value) if all of the following are true:

  - **A** is executed in program order before **B**.
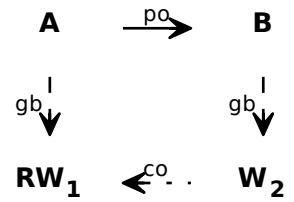
Figure G1.1 shows this.

$$A \xrightarrow{\text{po}} B$$

$$\downarrow{\text{gb}} \qquad\qquad \downarrow{\text{gb}}$$

$$RW_1 \xleftarrow{\text{co}} \cdot \quad W_2$$

Figure G1.1: Example: Any access before direct write

- An direct read R_1 of a System register made by an instruction **A** Reads-from a direct write W_2 to the same System register made by an instruction **B** (and hence **A** will read the new value) if all of the following are true:

    - **A** is executed in program order after **B**.

    - **A** and **B** use the same encoding for the System register.
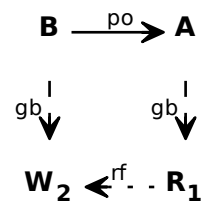
    Figure G1.2 shows this.

$$B \xrightarrow{\text{po}} A$$

$$\downarrow{\text{gb}} \qquad\qquad \downarrow{\text{gb}}$$

$$W_2 \xleftarrow{\text{rf}} \cdot \cdot R_1$$

Figure G1.2: Example: Direct read after direct write

- An indirect read r_1 of a System register made by an instruction **A** Reads-from a direct write W_2 to the same System register made by an instruction **B** (and hence **A** will read the new value) if all of the following are true:

    - **A** is executed in program order after a Context synchronization event **CSE**.

    - **B** is executed in program order before **CSE**.
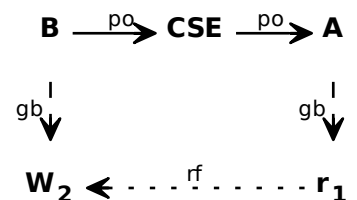
    Figure G1.3 shows this.

$$B \xrightarrow{\text{po}} CSE \xrightarrow{\text{po}} A$$

$$\downarrow{\text{gb}} \qquad\qquad\qquad \downarrow{\text{gb}}$$

$$W_2 \xleftarrow{\hspace{0.5em}\text{rf}\hspace{0.5em}} r_1$$

Figure G1.3: Example: Indirect read after CSE

These rules provide the minimum requirements to guarantee order. For example, in Figure G1.3 if **CSE** is not present then r_1 might read either the old or the new value.

Arm® Architecture Reference Manual, for A-profile architecture [1] places additional requirements for certain registers, agents, etc. These are not described further here.

Chapter G2
Stages of execution

This section shows the relationship between the *stages of execution*. The terms are defined in the Glossary.

G2.1 Stages of execution without *Transactional Memory Extension* (TME)

Figure G2.1 shows the stages of execution in a *Processing Element* (PE) that does not implement TME.
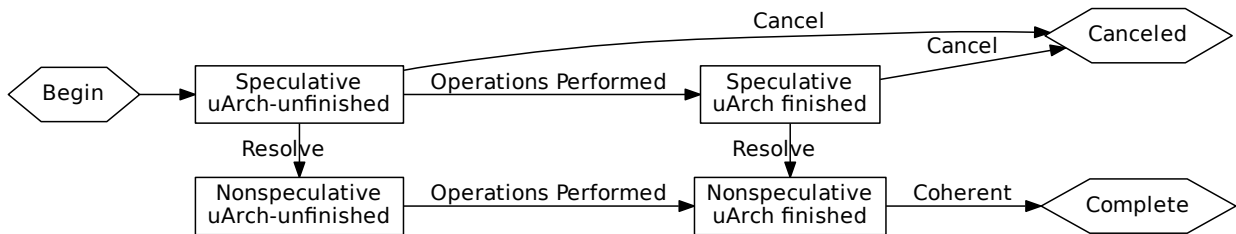

Figure G2.1: Stages of execution without TME

G2.2 Stages of execution with TME

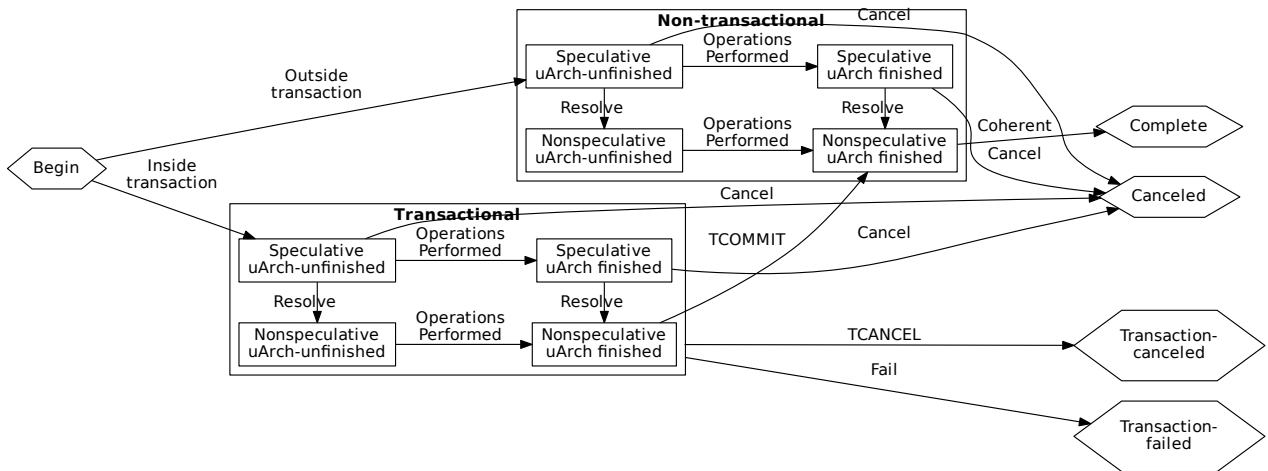Figure G2.2 shows the stages of execution in a PE that does implement TME.

Figure G2.2: Stages of execution with TME

Chapter G3
Additional Trace Buffer Extension software usage notes

G3.1 Context switching

S_{VKHHY} When switching out a process being traced, to save the current trace context and ensure all Trace operations are written to the correct context:

    (1) Prohibit program-flow trace using TRFCR_ELx. In many cases this is done before the process is traced. For example if all of the following are true:

- TRFCR_EL1.E0TRE is 1 to allow tracing of a process executed at EL0.

- TRFCR_EL1.E1TRE is 0 to prohibit tracing of the Operating System performing the context switch executed at EL1.

    (2) Execute a Context synchronization event to guarantee no new program-flow trace is generated. In the common case, this Context synchronization event is an exception taken to an Exception level where tracing is prohibited.

- If the trace unit is an ETE and the ETE is enabled, this also pauses the ETE Resources.

    (3) Execute a `TSB CSYNC` instruction to ensure the program-flow trace is flushed.

    (4) If necessary, disable the trace unit.

- For an ETE this is necessary if context is being switched. Software must set TRCPRGCTLR.EN to zero. This is necessary as:

    - The ETE must be disabled if saving the ETE state, as the ETE System registers can only be read when the ETE is disabled.

    - ETE trace compression logic is stateful, and disabling the ETE resets this compression state.

(5) Disable the Trace Buffer Unit. Set TRBLIMITR_EL1.E to zero.

- This must be done before changing the VMSA System registers to prevent the Trace Buffer Unit from speculatively accessing translation table entries.

(6) Execute a `DSB` operation.

- This is required if software will be reading the trace buffer contents, to ensure the writes to memory are Complete.

(7) Execute a further Context synchronization event.

- This is required to synchronize the effects of any System register writes since the previous Context synchronization event.

- This is also required if software will be reading the Trace Buffer Unit or trace unit System registers as part of the context switch, to capture indirect writes to those registers by Trace operations synchronized by the `TSB CSYNC`.

- For a subsequent direct read to capture the indirect write to TRBSR_EL1 resulting from an External Abort on a completed write, this Context synchronization event must follow the `DSB` above.

(8) Save and/or change the context. For example, save the MDCR_EL3, MDCR_EL2 (if applicable), Trace Buffer Unit, trace unit, and TRFCR_ELx System registers, and update the VMSA System registers for the new process.

S_{FMBCL} In other uses cases, tracing is not prohibited when software wants to save the trace context. For this case, if using an ETE, the sequence is slightly different:

(1) Disable the ETE. Set TRCPRGCTLR.EN to zero.

(2) Execute a Context synchronization event to guarantee no new program-flow trace is generated.

(3) Execute a `TSB CSYNC` instruction to ensure the program-flow trace is flushed.

(4) Execute a `DSB` and/or Context synchronization event as required by the previous example.

(5) Save and/or change the context.

For an ETE this sequence does not guarantee that all instructions before disabling the ETE are traced. The ETE might discard trace for preceding instructions when it is disabled.

S_{PKLXF} To restore the state of the Trace Buffer Unit and trace unit for switching in a process being traced, while tracing is prohibited:

(1) Restore the context. For example:

- Restore MDCR_EL3, MDCR_EL2 (if applicable), and the Trace Buffer Unit System registers, other than TRBLIMITR_EL1.
- Restore the trace unit System registers, other than enabling the trace unit.
- Ensure the TRFCR_ELx System registers are correct for the process being traced.
- Update VMSA System registers for the process being returned to.

(2) Execute a Context synchronization event to guarantee the trace unit and Trace Buffer Unit will observe the new values of the System registers written by the previous step.

(3) Enable the Trace Buffer Unit by setting TRBLIMITR_EL1.E to 1. This must be done after setting up the correct VMSA System registers for the trace buffer, as the Trace Buffer Unit might now speculatively prefetch and cache address translations. See R_{BSMLW} and S_{YKCND}.

(4) If necessary, enable the trace unit. If using an ETE, software must set TRCPRGCTLR.EN to one.

(5) Execute a Context synchronization event to guarantee tracing is allowed. In the common case, this is an `ERET` instruction that returns to a different Exception level where tracing is allowed.

This must be done after saving the state from the previous process, if applicable.

S_{YKCND} Because the Trace Buffer Unit can prefetch and cache address translations when the Trace Buffer Unit is Enabled:

- Software must not enable the Trace Buffer Unit before programming the System registers for the owning translation regime. In particular, during a context switch operation:
  - If switching from a context using the Trace Buffer Unit, the Trace Buffer Unit must be disabled before modifying the System registers for the owning translation regime being switched from.
  - If switching to a context using the Trace Buffer Unit, the Trace Buffer Unit must not be enabled until after modifying the System registers for the owning translation regime being switched to.

- The Trace Buffer Unit must not be enabled when the *Processing Element* (PE) is not executing in the owning Security state or when executing at EL3 and SCR_EL3.NS does not indicate the owning Security state.

- In normal conditions, enabling the Trace Buffer Unit early before returning to the context being traced might be advantageous if the implementation does prefetch address translations.

G3.2 Controlling generation of trace buffer management events

S_{NFQZJ} The Trace Buffer Extension does not include a direct capability to program the Trace Buffer Unit to generate a maintenance interrupt when the trace buffer reaches a programmed level below the Limit pointer, and continue collecting trace until either the interrupt is serviced or (possibly) the trace buffer fills (whichever comes first). This allows an almost lossless collection of trace.

However, the Trace Buffer Unit can be programmed to give similar behavior in one of the following ways:

(1) Using Wrap mode. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, but set the trace buffer mode to Wrap mode the current write pointer to point part way through the trace buffer, such that the remaining space in the trace buffer is the watermark level. When the amount of trace collected reaches the watermark level, the current write pointer is wrapped and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:

- The trace buffer management event is generated and the trace unit receives the TRB_WRAP event at the watermark level.

- The oldest trace in the trace buffer will be lost if more trace is generated than fits in the trace buffer, because it is overwritten by newer trace. Note that some loss of trace is inevitable if more trace is generated than fits in the trace buffer.

- The trace history does not start at the start of the trace buffer, and must be aligned by software.

(2) Use a Trigger Event. At the start of a trace session, configure the Base pointer and Limit pointer for the trace buffer as normal, and set the trace buffer mode to Fill mode and the current write pointer to the start of the trace buffer. Set the trigger mode to IRQ on trigger, the Trigger Counter to the watermark level, and TRBSR_EL1.TRG to 1. When the amount of trace collected reaches the watermark level, a Trigger Event occurs and a trace buffer management event is generated, but trace continues to be collected. This approach has the following advantages and disadvantages:

- The trace buffer management event is generated and the trace unit receives the TRB_TRIG event at the watermark level.

- The newest trace in the trace buffer will be discarded if more trace is generated than fits in the trace buffer. To overwrite the oldest trace instead, set the trace buffer mode to Circular Buffer mode.

- This method cannot be used if also searching for a Detected Trigger event from the trace unit.

- The current write pointer does not have to be set to the start of the trace buffer. If the trace buffer already contains data that software does not want to be overwritten, the current write pointer can be set to point to after this data. In this case using Circular Buffer mode or Stop on trigger can also be used to control when Collection is stopped and what data is overwritten.

Chapter G4
Transactional Memory Extension (TME) Litmus tests

This appendix is to help understand, via examples, how transactions extend the Armv8 memory model.

See also Section C1.3 *Memory model*, *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Definition of the Armv8 memory model*, and *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Barrier Litmus Tests*.

G4.1 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences, using the same conventions as *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Load-Acquire Exclusive, Store-Release Exclusive and barriers.* In addition, we define the following constructs.

The construct `TX{<code>}` describes the following sequence:

`loop:`

`TSTART X12` ; attempt to start a new transaction

`CBNZ X12, loop` ; retry forever

`<code>`

`TCOMMIT`

Note: This construct is unsafe in the general case because a transaction is permitted to never commit and should be avoided. But, for the simple examples that are presented in this section it is expected that an implementation will be able to commit the transaction eventually.

G4.2 Transaction strong isolation

TME transactions are *strongly isolated*. Strongly isolated transactions require both *non-interference* and *containment* from other transactions as well as from non-transactional code executing concurrently.

G4.2.1 Containment

The containment property of transactions means that only the last write to a Location is observable outside of the transaction:

P1	P2
LDR W5, [X1]	TX { STR W5, [X1] STR W6, [X1] }

In this example, the result of P1:W5==0x55 is not permissible.

G4.2.2 Non-interference

The non-interference property of transactions means that multiple reads to the same memory Location inside a transaction should return the same value:

P1	P2
STR W5, [X1]	TX { LDR W5, [X1] LDR W6, [X1] }

In this example, it is required for P2:W5 and P2:W6 to contain the same value.

The non-interference property of transactions also means that a read to Location following a write to the same Location inside a transaction should return the value of the write:

P1	P2
STR W5, [X1]	TX { STR W6, [X1] LDR W5, [X1] }

In this example, it is required that P2:W5==0x66.

G4.3 Transactions and barriers

The following sections show that most of the examples in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Simple ordering and barrier cases* can be achieved using transactions without the need for additional barriers.

G4.3.1 Simple weakly consistent ordering

The *simple weakly consistent ordering* example in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Simple ordering and barrier cases* can be solved by the use of transactions in various ways. In the following examples, the result of P1:W6==0, P2:W5==0 is not permissible.

Memory accesses after the transaction cannot be observed by other observers before the transaction:

P1	P2
TX { STR W5, [X1] } LDR W6, [X2]	TX { STR W6, [X2] } LDR W5, [X1]

Memory accesses before the transaction cannot be observed by other observers after the transaction:

P1	P2
STR W5, [X1] TX { LDR W6, [X2] }	STR W6, [X2] TX { LDR W5, [X1] }

An empty transaction behaves like a barrier instruction:

P1	P2
STR W5, [X1] TX {} LDR W6, [X2]	STR W6, [X2] TX {} LDR W5, [X1]

G4.3.2 Message passing

The *weakly-ordered message passing* problem in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Simple ordering and barrier cases* can be solved by the use of transactions in various ways. In the following examples, the result of P2:W5==0 is not permissible.

Using a transaction when accessing the data:

P1	P2
TX { STR W5, [X1] } STR W0, [X2]	WAIT([X2]==1) AND X12, X12, #0 LDR W5, [X1, X12]

An empty transaction behaves like a barrier instruction:

P1	P2
STR W5, [X1] TX {} STR W0, [X2]	WAIT([X2]==1) AND X12, X12, #0 LDR W5, [X1, X12]

These approaches also work with multiple observers, viz, with extra observers running the same sequence as P2.

Chapter G5
Transactional Memory Extension (TME) Transactional Lock Elision

G5.1 Overview

Contended locks can lead to software scalability problems on multicore systems. Hardware Transactional Memory may improve the scalability of contended locks by implementing transactional lock elision.

With transactional lock elision, critical regions are converted into transactions and multiple threads can execute their critical regions in parallel as long as there are no transactional conflicts. When such conflicts occur, the implementation resolves them by failing transactions as necessary.

Failed transactions must re-execute the critical region for the application to progress, but since transactions are best effort, a fallback execution path is necessary. In the case of transactional lock elision, typically the fallback path acquires a lock and executes the critical region non-transactionally.

The most popular implementations of transactional lock elision use the same programming model as locks, so they can be applied to existing programs.

Arm notes however that not all locking libraries are equal with respect to lock elision. Certain existing libraries cannot be elided soundly (see below for an example) and will need reviewing or perhaps revisiting entirely (e.g. by adding extra barriers or using atomic operations) if they are intended to be used in this context. This is the case both for the surrounding locks on other threads, and the lock used as the fallback path.

G5.2 Conventions

Many of the examples are written in a stylized extension to Arm assembler, to avoid confusing the examples with unnecessary code sequences, using the same conventions as the *Conventions* topic in the *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Barrier Litmus Tests* chapter. In addition, we define the following constructs.

The construct `LOCK(Xx)` describes the following sequence from *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Acquiring a lock*:

`PRFM PSTL1KEEP, [Xx]` ; preload into cache in unique state

`loop:`

`LDAXR W5, [Xx]` ; read lock with acquire

`CBNZ W5, loop` ; check if 0

`STXR W5, W0, [Xx]` ; attempt to store new value

`CBNZ W5, loop` ; test if store succeeded and retry if not

The construct `UNLOCK(Xx)` describes the following sequence from *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Releasing a lock*:

`STLR WZR, [Xx]` ; clear the lock with release semantics

The construct `CHECK(Xx)` describes the following sequence:

`LDR W5, [Xx]` ; read lock

The construct `CHECK_ACQ(Xx)` describes the following sequence:

`LDAR W5, [Xx]` ; read lock with acquire

The construct `WAIT_ACQ(Xx==0)` describes the following sequence:

`loop:`

`LDAR W5, [Xx]` ; load acquire ensures it is ordered before subsequent loads/stores

`CBNZ W5, loop`

In the rest of this chapter, the `LOCK(Xx)` construct is used as one example of lock acquisition, the `UNLOCK(Xx)` construct is used as one example of lock release, the `CHECK(Xx)` and `CHECK_ACQ(Xx)` constructs are used as one example of reading the status of a lock, and the `WAIT_ACQ(Xx==0)` construct is used as one example of waiting until the lock is free. Unless otherwise stated, the examples where these constructs are used would work the same if these constructs were mapped to different locking primitives, such as but not limited to the ones presented in *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Ticket locks* and *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Use of Wait For Event (WFE) and Send Event (SEV) with locks*.

G5.3 Acquiring a lock

The recommended instruction sequence for acquiring a lock using transactional lock elision is as follows (where `W6` contains a retry count for the transaction):

`loop:`

`TSTART X5` ; attempt to start a new transaction

`CBNZ X5, fallback` ; check if `TSTART` succeeded

`CHECK_ACQ(X1)` ; add the fallback lock to the transactional read set ; and set `W5` to 0 if the fallback lock is free.

`CBZ W5, enter` ; if the fallback lock is free enter the critical region

`TCANCEL #0xFFFF` ; otherwise cancel the transaction with RTRY set to 1

`fallback:`

`TBZ X5, #15, lock` ; if `RTRY` is 0 take the fallback lock

`SUB W6, W6, #1` ; decrement the retry count

`CBZ W6, lock` ; take the lock if 0

`WAIT_ACQ(X1==0)` ; wait until the lock is free

`B loop` ; retry the transaction

`lock:`

`LOCK(X1)` ; elision failed, acquire the fallback lock

`DMB ISH` ; block loads/stores from the critical region

`enter:`

G5.3.1 Checking the lock inside the transaction

When eliding a lock, it is required to check the status of the lock inside the transaction because the memory accesses of a thread that executes the critical region non-transactionally are not tracked by hardware. In the following example, mutual exclusion cannot be guaranteed:

P1	P2
LOCK(X1) LDR W5, [X2]	TSTART X5 CBNZ X5, fallback
ADD W5, W5, #1 STR W5, [X2]	LDR W5, [X2] ADD W5, W5, #1
UNLOCK(X1)	STR W5, [X2] TCOMMIT

To ensure mutual exclusion when the transaction by P2 commits after the load from the address in `X2` by P1 and before the store to the address in `X2` by P1, P2 must ensure that the lock variable is contained within the transactional read set, which occurs as a side effect of adding the `CHECK_ACQ(Xx)` construct inside the transaction.

To ensure mutual exclusion when the transaction by P2 starts after the `LOCK(Xx)` construct by P1 and the transaction in P2 commits before the store to the address in `X2` by P1, P2 must test the value of the lock returned by the `CHECK_ACQ(Xx)` construct and cancel the transaction as appropriate.

Using `CHECK_ACQ(Xx)` instead of `CHECK(Xx)` ensures that read from a critical region executing in a transaction do not take their values from writes from a mutually excluded critical region that acquires a lock, including when the acquisition of the lock generates a conflict that fails the transaction.

G5.3.2 Checking the lock at the fallback path

When eliding a lock, it is recommended to use the `WAIT_ACQ(Xx==0)` construct in the fallback handler to avoid the Lemming effect, in which one thread acquiring the lock causes all other concurrent threads to do so too because they retry too many times while the first thread holds the lock.

G5.3.3 Synchronization between transactions and the fallback path

When transactional lock elision fails, and the lock is acquired, it is required that loads and stores from the critical region are not observable before the lock is acquired.

In the following example, mutual exclusion cannot be guaranteed:

P1	P2
LOCK(X1) LDR W5, [X2]	TSTART X5 CBNZ X5, fallback
ADD W5, W5, #1 STR W5, [X2]	CHECK_ACQ(X1) CBNZ W5, cancel
UNLOCK(X1)	LDR W5, [X2] ADD W5, W5, #1
	STR W5, [X2] TCOMMIT

The following architecturally permissible ordering violates mutual exclusion:

- P1 performs the Load-Exclusive of the `LOCK(Xx)` construct.

- P1 performs the load of the shared variable from the critical region (reading 0).

- P2 enters the transaction, executes the critical region writing 1 to the shared variable, and commits the transaction (because the lock is not acquired yet).

- P1 performs the Store-Exclusive of the `LOCK(Xx)` construct. The Store-Exclusive does not fail because there are no intervening writes to the lock variable (P2 only reads the lock)

- P1 performs the store of the shared variable from the critical region (writing 1).

To ensure mutual exclusion when the fallback lock acquisition implementation permits reads or writes from the critical region to be observable before the lock variable is updated, a DMB is added before the first load or store of the critical region.

All the recommended locking acquisition sequences from *Arm® Architecture Reference Manual, for A-profile architecture* [1] *Load-Acquire Exclusive, Store-Release Exclusive and barriers* that use Load-Exclusive and Store-Exclusive to acquire the lock are affected.

Lock implementations that use an atomic operation with Acquire or Acquire-Release semantics (such as `LDADDA`, `SWPA`, etc.) to update the lock variable are not affected.

G5.4 Releasing a lock

The recommended instruction sequence for releasing a lock using transactional lock elision is as follows:

`CHECK(X1)` ; set `W5` to 0 if the fallback lock is free

`CBNZ W5, unlock` ; check if 0

`TCOMMIT` ; the lock was elided, exit the transaction

`B exit`

`unlock:`

`UNLOCK(X1)` ; elision failed, release the fallback lock

`exit:`

G5.4.1 Elision and nesting

When releasing a lock that has potentially been elided it is advisable to use the `CHECK(Xx)` construct to check if the lock is acquired instead of using `TTEST` to check if the *Processing Element* (PE) is in Transactional state, because inside a nested transaction using `TTEST` is not sufficient to distinguish if the lock was elided or not.

Chapter G6
Transactional Memory Extension (TME) Implementation recommendations

G6.1 Permitted architectural difference between PEs

The architecture does not support implementations where the value of ID_AA64ISAR0_EL1.TME differs between PEs in a single system.

G6.2 Individual operation latency

In order to not affect single-thread performance when using transactional lock elision, Arm recommends that the latency of starting and committing a transaction is not higher than the latency of the illustrative code sequence for acquiring and releasing a spinlock.

In an application that successfully employs transactional lock elision, it is expected that most transactions will not fail, so it acceptable that failing or canceling a transaction is a slower operation than committing a transaction. Even so, in order to not affect single-thread performance, Arm recommends that the latency of failing or canceling a transaction is not unreasonably high compared to the latency of committing a transaction.

G6.3 Read and write set capacity

Arm recommends that, for adequate performance of applications written in Java and C/C++, hardware supports a read set size of at least 512 objects and a write set size of at least 300 objects – assuming average object size to be 128 bytes.

G6.4 State tracking

The properties of the transactional read set and the transactional write set imply that the implementation tracks the addresses of transactional reads and writes and buffers the data values of transactional writes throughout the execution of a transaction.

Arm expects a typical transaction to execute between a few tens to a few thousand instructions and to access up to several hundreds or even thousands of distinct transactional reservation granules.

Arm expects the transactional write set to contain a significantly smaller number of transactional reservation granules compared to the transactional read set of a typical transaction.

Arm expects the capacity of a typical Level 1 data or unified cache to be enough to hold the transactional write set, but not enough to hold the transactional read set in many cases.

Arm expects the associativity of a typical Level 1 data or unified cache to not be enough to hold the transactional write set or the transactional read set in many cases.

Arm considers a typical Level 1 data or unified cache to have a capacity between 32KB and 64KB with an associativity between 2 to 4.

Arm recommends that implementations take these expectations into consideration in order to avoid frequent transactional failures due to insufficient hardware resources.

For holding the transactional write set, Arm recommends the use of hardware structures in addition to the Level 1 data or unified cache that can provide the illusion of high associativity, such as a small fully associative cache.

For holding the transactional read set, Arm recommends the use of hardware structures in addition or instead of the Level 1 data or unified cache that are capable of holding the addresses of tens of thousands of transactional reservation granules, such as higher-level caches, Bloom filters, Signatures, or other similar structures.

G6.5 Transactional conflicts

Arm recommends that the hardware cache coherency facilities of the processor be used to detect transactional conflicts. This is also known as *eager conflict detection* because conflicts are detected when the read or write requests are generated. The alternative, *lazy conflict detection*, defers the detection of conflicts until the transaction attempts to commit.

Arm recommends that implementations do not generate a transactional conflict when a read generated by a PRFM instruction or by hardware prefetching accesses a Location within the transactional write set of a transaction.

Part H
Glossary

Chapter H1
Glossary

Active

> The feature is implemented and enabled, and the trace unit is in a state in which the feature is programmed to operate.

Analysis of the trace element stream

> This term refers to the process of:
>
> • Tracing elements that carry information that a trace analyzer requires to enable it to analyze the trace successfully.
>
> • Tracing elements that either directly indicate program execution, or carry information about program execution.
>
> A trace element stream might also contain trace elements that contain timing information.
>
> This term is distinct from analysis of *program execution*.

ARC

> Address Range Comparator

ARCs

> Address Range Comparators

AST

> Abstract Syntax Tree

Atom element

> *P0 element* to imply the execution of instructions

Bit replacement

> Only bits that have changed are encoded.

Cancel

> Operation to indicate that an instruction has not architecturally executed. The architecturally state of the PE is rolled back to before the instruction.

Cancel element

> Element to indicate that instructions inferred by *Atom elements* or *Exception elements* did not architecturally execute.

Canceled

> An operation on an incorrectly predicted execution path.

CATU

> CoreSight Address Translation Unit

Commit element

> Element to indicate that instructions implied by *Atom elements* or *Exception elements* did or will architecturally execute.

Commit window

> The Commit window defines the range of P0 elements which are committed by a Commit element. The oldest P0 element in the Commit window is the first P0 element committed when a Commit element occurs. By default, the Commit window starts on the oldest uncommitted P0 element, and moves forward to the next uncommitted P0 element with each P0 element committed by a Commit element. The Commit Window Move element moves the start of the Commit window by a number of P0 elements, to allow a Commit element to commit P0 elements which are younger than the oldest uncommitted P0 elements, leaving these older P0 elements uncommitted.

Complete

> An operation that has finished all its operational pseudocode, and the results of any memory accesses, including translation table walks and updates, are coherent with other observers. For more information see the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

Context element

> Element to indicate the context in which the the instructions are executing.

CoreSight Address Translation Unit

> A form of System MMU for trace streams.

Cross-trigger Interface

> See the *ARM CoreSight Architecture Specification* [5] and the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

CTI

> Cross-trigger Interface

Cycle count element

> Element to indicate passage of PE clock cycles.

Disabled

> The feature is either not implemented, or is implemented but has been programmed to be disabled during the trace session.

Discard element

> Element to indicate that no more *Commit element* or *Cancel elements* will follow for the speculative *P0 elements* in trace element stream.

EBBR

> Embedded Base Boot Requirements

Element

> Basic building block on the data model.
>
> - *Atom element*
> - *Cancel element*
> - *Commit element*
> - *Context element*
> - *Cycle Count element*
> - *Discard element*
> - *Exception element*
> - *Event element*
> - *Mispredict element*
> - *Overflow element*
> - *Source Address element*
> - *Target Address element*
> - *Timestamp element*
> - *Transaction Start element*
> - *Transaction Commit element*
> - *Transaction Failure element*
> - *Trace Info element*
> - *Trace On element*
> - *Q element*

Element stream

See Trace element stream.

Embedded Trace Extension

See Chapter D1 *Embedded Trace Extension*.

Embedded Trace Router

See the *ARM CoreSight Architecture Specification* [5].

Enabled

The feature is implemented and has been programmed to operate at runtime. However, because of other trace unit conditions, the feature might not be active.

ETB

Embedded Trace Buffer

ETE

Embedded Trace Extension

ETE trace operation

The operation that the trace unit must perform to meet the requirements of the trace operation.

ETEEvent

A feature of the trace unit that is used to generate Event elements and drive External Outputs. Each ETEEvent can programmed to be sensitive to resource events.

ETR

Embedded Trace Router

Event element

Element to indicate that an ETEEvent occurred.

Event trace

The trace uses *Event elements* that indicate certain events have occurred in the program that the PE is executing. The program events to be indicated are selected before a trace session.

Exception element

Element to indicate that an *Exceptional occurrence* occurred.

Exceptional occurrence

Events indicated by an *Exception element* by the ETE architecture, including the following:

- PE Architectural exceptions.
- ETE defined exceptions.
- IMPLEMENTATION DEFINED exceptions.

External reads

Reads of the trace unit registers via the external debugger interface.

External writes

Writes of the trace unit registers via the external debugger interface.

Filtering

The function to select what PE activity that causes trace elements to be generated.

GIC

Generic Interrupt Controller

Implemented

The feature is included in the implementation.

Inactive

The feature is either not implemented or is disabled, or the trace unit is in a state in which the feature is programmed not to operate.

Inoperative

The trace unit is unable to generate packets.

Instruction Block

Set of instructions that are executed atomically.

Instruction trace

PE trace that indicates program execution, such as branches taken, the execution of instructions, and Exceptional occurrences.

Instruction trace might also contain timing information.

Instruction trace contains information that a trace analyzer requires to enable it to analyze trace.

ISA

Instruction Set Architecture.

logical AND

$\wedge$

logical NEG

$\neg$

logical OR

$\vee$

Memory System Performance Resource and Monitoring Extension

See *Arm® Architecture Reference Manual Supplement; Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture* [6].

Memory Tagging Extension

See the "Memory Tagging Extension" chapter of the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

Microarchitecturally-finished

An operation that has finished all its operational pseudocode, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers.

Microarchitecturally-unfinished

An operation that has not completed all its operational pseudocode.

Mispredict element

Element to indicate that the state of an Atom element needs to be modified.

MPAM

Memory System Performance Resource and Monitoring Extension

MSI

Message-signaled Interrupt

Nonspeculative

An operation on a confirmed execution path.

Nonspeculative Microarchitecturally-finished

An operation that has finished all its operational pseudocode, on a confirmed execution path, although the results of any memory accesses, including translation table walks and updates, are not yet coherent with other observers and the operation is not Complete.

Nonspeculative Microarchitecturally-unfinished

An operation that is in progress on a confirmed execution path.

Not implemented

The feature is not included in the implementation.

Operative

The trace unit is able to generate packets.

Overflow element

Element to indicate that a trace unit buffer overflow has occurred and some trace might have been lost.

Packet stream

See Trace byte stream.

PE

Processing Element

POD

Plain Old Data. The data is packet specific.

PPI

Private Peripheral Interrupt

Reset

The trace unit has a Trace Unit reset

Resolve

Operation to indicate that an instruction has or will architecturally execute.

Resource event

The output of a Resource Selector, or a boolean combined pair of Resource Selectors. Resource events are selected by various trace unit features to control those features.

Rewind point

A rewind point is a point in the program flow where execution can return to if all subsequent execution is found to have been incorrectly speculatively executed.

SAC

Single Address Comparator

SACs

Single Address Comparators

SBBR

Standard Base Boot Requirements

SBSA

Server Base System Architecture

Source Address element

Element to indicate the address of a taken *P0 instruction*.

SPE

Statistical Profiling Extension

Speculative

An operation on a predicted execution path. For more information see the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

Speculative Microarchitecturally-finished

An operation that has finished all its operational pseudocode, on a predicted execution path.

Speculative Microarchitecturally-unfinished

An operation that is in progress on a predicted execution path.

SPI

Shared Peripheral Interrupt

Statistical Profiling Extension

See the *Arm® Architecture Reference Manual, for A-profile architecture* [1].

Sub isa

The alignment of Virtual Address in the trace packet. When combined with the execution state, indicates the instruction set architecture.

Target Address element

Element to indicate the instruction set and virtual address of the next instruction to be executed.

TCANCEL

The instruction which causes the PE to transition out of Transactional state, causing the transaction to fail.

TCOMMIT

The instruction which causes the PE to transition out of Transactional state, causing the transaction to succeed.

Timestamp element

Element to indicate the time the trace was captured.

TMC

Trace Memory Controller

TME

Transactional Memory Extension

Trace analyzer

A tool that takes the trace byte stream, or trace element stream, and analyzes them to determine PE execution. This tool can be part of a self-hosted debug environment, or an external debug tool.

Trace byte stream

The stream of trace packets which describe a sequence of trace elements.

Trace byte stream generation

The conversion of the trace elements stream into trace packets to form the trace byte stream.

Trace element stream

A sequence of trace elements which describe the operation of the PE.

Trace element stream generation

The conversion of the PE activity into trace elements.

Trace Info element

Element to indicate the state of the compression of the trace element stream.

Trace Memory Controller

See the *ARM CoreSight Architecture Specification* [5].

Trace On element

Element to indicate a discontinuity in the trace element stream.

Trace operation

The architectural operation that a PE, trace unit and trace buffer must perform.

Trace session

The period while the trace unit is enabled.

Trace unit

The implementation that is used to generate trace.

Trace Unit

A functional unit that generates generating program-flow trace data for use by self-hosted software. In the *Arm® Architecture Reference Manual, for A-profile architecture* [1], this is described as a *trace component*, *trace functionality*, *trace logic* or *trace macrocell*. The trace unit might be an implementation of an Arm Trace Architecture, such as the *Embedded Trace Extension* (ETE), or might be some other IMPLEMENTATION DEFINED program-flow trace functionality.

Trace unit buffer overflow

Buffering inside the trace unit is unable to capture more trace data. A trace unit buffer overflow is independent of the Trace Buffer Extension filling or wrapping a trace buffer in memory.

Trace Unit reset

trace unit Core power domain reset

Transaction atomicity

If a transaction succeeds it must appear that all the instructions executed in Transactional state have executed collectively as a single atomic operation.

Transaction-canceled

An operation that was part of a transaction that was canceled by a TCANCEL instruction.

Transaction-failed

An operation that was part of a transaction that failed.

Transactional

An operation that is part of a transaction and the transaction has not yet succeeded, failed or been canceled. The operation can be any of:

- Speculative Microarchitecturally-unfinished.
- Speculative Microarchitecturally-finished.
- Nonspeculative Microarchitecturally-unfinished.
- Nonspeculative Microarchitecturally-finished.

Transactional execution

Instructions inside a transaction execute in Transactional state. This specification uses the term transactional execution to refer to the execution of instructions in Transactional state.

TSTART

The instruction which initiates the transition into Transactional state.

Unsigned LE128n

Little endian leading zero compression.

ViewInst

The filtering function which defines which instructions are traced.

ViewInst active

Both of the following are true:

- The trace unit has been programmed and is enabled.

- The ViewInst instruction trace filtering function is permitting instruction tracing, therefore the trace unit is generating instruction trace. In addition, the trace unit might also be generating *Event elements* in the trace.

ViewInst inactive

Both of the following are true:

- The trace unit has been programmed and is enabled.

- The trace unit is not generating any instruction trace, because the ViewInst function is filtering instruction tracing. However, the trace unit might be generating *Event elements* in the trace element stream.