Arm® Embedded Trace Macrocell Architecture Specification

ETMv4.0 to ETMv4.5

arm

ARM IHI0064G.b (ID120519)

Arm Embedded Trace Macrocell Architecture Specification
ETMv4.0 to ETMv4.5

Release Information

The *Change history* table lists the changes that have been made to this document.

Change history

Date	Issue	Confidentiality	Change
28 March 2012	A	Confidential Beta	First Beta Release.
22 May 2013	B (B.a)	Non-Confidential	Final Release.
21 March 2014	B (B.b)	Non-Confidential	Minor updates to Issue B.a.
30 June 2015	C	Non-Confidential	Incorporation of ETMv4.1 architecture.
8 February 2016	D	Non-Confidential	Incorporation of ETMv4.2 architecture.
25 April 2017	E	Non-Confidential	Incorporation of ETMv4.3 architecture.
30 April 2018	F	Non-Confidential	Incorporation of ETMv4.4 architecture.
02 October 2019	G (G.a)	Non-Confidential	Incorporation of ETMv4.5 architecture.
12 December 2019	G (G.b)	Non-Confidential	Added support for Armv8.1-M to ETMv4.5 architecture.

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
Arm Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.5

Preface

This preface introduces the *Embedded Trace Macrocell (ETM) Architecture Specification*. It contains the following sections:

About this document

This document describes version 4.0 to version 4.5 of the architecture for the Arm *Embedded Trace Macrocell* (ETM).

Some features of the architecture are IMPLEMENTATION DEFINED. For more information, see the relevant ETM *Technical Reference Manual* (TRM).

Intended audience

This document targets the following audiences:

- Designers of development tools supporting ETMv4 functionality.
- Advanced users of development tools supporting ETMv4 functionality.
- Designers of trace analyzers for use with ETMv4 trace units.
- Designers of an Arm-based product that includes an ETMv4 trace unit.
- Engineers specifying, designing, or implementing an ETM conforming to the Arm ETMv4 architecture.

Hardware engineers incorporating an Arm ETM in their design must consult the relevant ETM *Technical Reference Manual*. Arm recommends that all users of this specification have experience of the Arm architecture.

Using this document

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for a brief introduction to tracing, and to Arm ETMv4 architecture.

Chapter 2 *About the Trace Streams*

Read this chapter for a description of the trace streams the trace unit generates. This includes information about the instruction and data trace streams, in addition to how they synchronize with each other and a trace analyzer. It also lists optional features that can be implemented in an ETMv4 trace unit.

Chapter 3 *About the Trace Unit*

Read this chapter for an overview of the trace unit and its behavior. This includes information about possible trace unit power domain implementations and powerdown support.

Chapter 4 *Programming the Trace Unit*

Read this chapter for a description of the internal structure of the ETMv4 trace unit architecture, and a guide on programming the unit. This includes information about the filtering and resource selection logic, and information about accessing the trace unit, either from an external debugger or from the core.

Chapter 5 *Descriptions of Trace Elements*

Read this chapter for a description of the elements that the trace unit generates to indicate the flow of the traced program. There is also information about the elements that comprise the instruction and data trace streams.

Chapter 6 *Descriptions of Trace Protocols*

Read this chapter for a description of the output packets that indicate the elements in the instruction and data trace streams, in addition to the necessary state information that must be retained between packets to correctly interpret the trace stream.

Chapter 7 *Register Descriptions*

Read this chapter for a description of the registers in the ETMv4 trace unit architecture.

Appendix A *Examples of Trace*

Read this appendix for a set of examples of trace using an ETMv4 trace unit.

Appendix B *Required Architecture Versions*

Read this appendix for a list of the required version of ETM architecture for each version of the Arm architecture.

Appendix C *Recommended Configurations*

Read this appendix for a set of recommended configurations for trace unit implementations.

Appendix D *Filtering Examples*

Read this appendix for examples of instruction address range filtering, and the typical trace output for each example.

Appendix E *Resource Selection Examples*

Read this appendix for examples of programming the ETMv4 resource selectors.

Appendix F *Instruction Categories*

Read this appendix for a list of instructions that are classified as branch, load and store, conditional, or flag setting instructions for the trace generation and analysis.

Appendix G *Standard Layout of the External Inputs*

Read this appendix for recommendations on the number and type of inputs available to a trace unit.

Appendix H *Pseudocode Definition*

Read this appendix for a guide to the pseudocode used in this document.

Appendix I *Architecture Revisions*

Read this appendix for information on the architectural changes between different revisions of the ETMv4 architecture.

Appendix J *Revisions*

Read this appendix for information on the changes between issues of this document.

Conventions

The following sections describe conventions that this document can use:

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

Typographic conventions

The typographical conventions are:

italic Introduces special terminology, denotes internal cross-references and citations, or highlights an important note.

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 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://infocenter.arm.com.
- A cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, *Pseudocode descriptions*.
- A link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example *Trace unit behavior* on page 3-98 or TRCTSCTLR.

Signals

In general, this document does not define 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`.

Pseudocode descriptions

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

Additional reading

This section lists relevant publications from Arm and third parties.

See the Infocenter http://infocenter.arm.com, for access to Arm documentation.

Arm publications

This document contains information that is specific to this specification. See the following documents for other relevant information:

- *AMBA® APB Protocol Specification* (ARM IHI 0024).
- *AMBA® 4 ATB Protocol Specification* (ARM IHI 0032).
- *Arm® CoreSight™ Architecture Specification* (ARM IHI 0029).
- *Arm® Debug Interface v5 Architecture Specification* (ARM IHI 0031)
- *Arm® Debug Interface v6 Architecture Specification* (ARM IHI 0074).
- *Arm®v6-M Architecture Reference Manual* (ARM DDI 0419).
- *Arm® Architecture Reference Manual Armv7-A and Armv7-R edition* (ARM DDI 0406).
- *Arm®v7-M Architecture Reference Manual* (ARM DDI 0403).
- *Arm®v8-A Architecture Reference Manual* (ARM DDI 0487).
- *Arm®v8-R Architecture Reference Manual* (ARM DDI 0568).
- *Arm®v8-M Architecture Reference Manual* (ARM DDI 0553).

Other publications

This section lists relevant documents published by third parties:

- JEDEC, *Standard Manufacturers Identification Code*, JEP106 http://www.jedec.org.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this document

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

- The title.
- The number, ARM IHI0064G.b.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

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

Chapter 1
Introduction

This chapter contains a brief introduction to tracing, and to version 4.0 to version 4.5 of the architecture for the Arm® *Embedded Trace Macrocell* (ETM). It contains the following sections:

1.1 Introduction to processing element tracing

In the context of the ETMv4 architecture, the term *tracing* refers to:

- The tracing of instruction execution.
- The tracing of data movements.
- The tracing of events in a PE.

Note

This document describes a *Processing element (PE)* and trace elements. Trace elements describe the execution of a PE. A PE is not a trace element.

A trace unit performs these tracing functions. The trace unit is the hardware implementation of a particular functional configuration of an Arm trace architecture. A trace unit might be implemented as part of a full debug solution inside a *System-on-Chip* (SoC). The trace unit traces instructions and data by monitoring the instruction and data buses. A trace unit has the following interfaces:

- A PE interface, providing visibility of instruction execution and data movements within a PE.
- One or more programming interfaces:
  - A system instruction interface for direct programming from a PE.
  - A memory-mapped interface for programming from a PE or other masters in the system.
  - An external debugger interface which is connected to a debug port on the chip, such as an ADI *Debug Access Port* (DAP).
- A trace output interface, such as a parallel data interface.

Figure 1-1 shows an example of a trace unit that is implemented in an SoC.

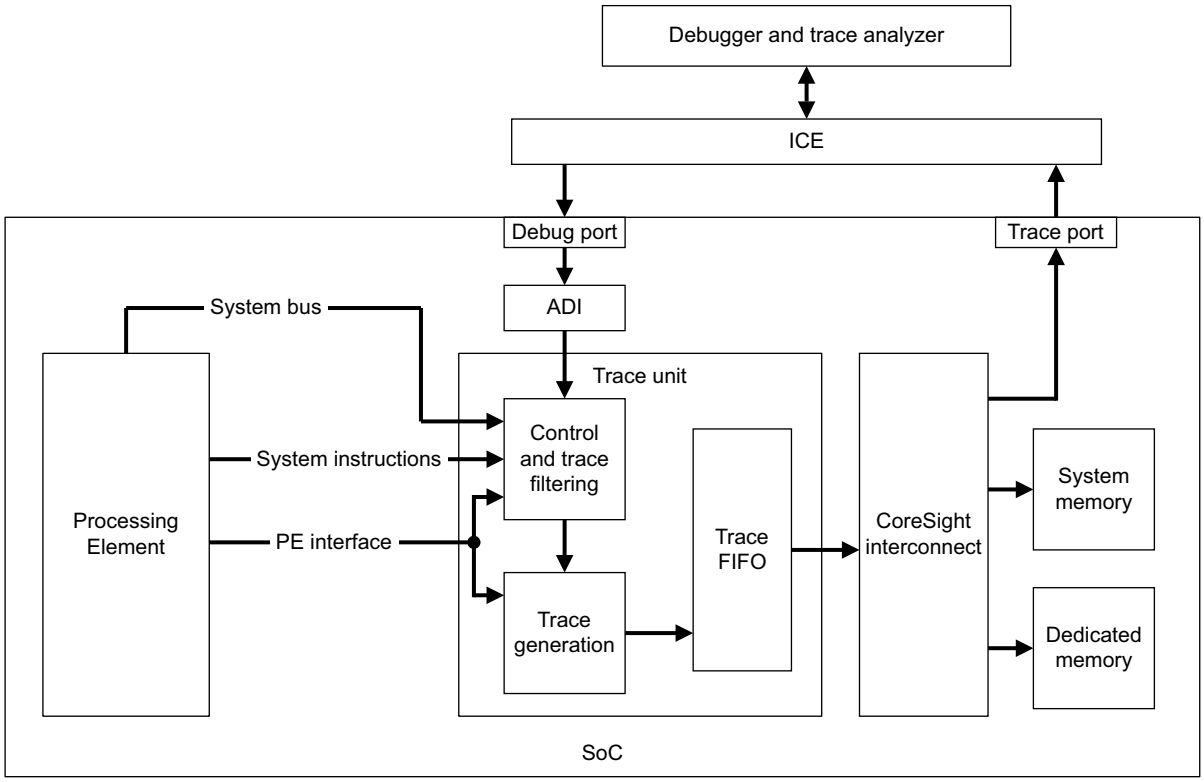


Figure 1-1 Example SoC with a trace unit

The trace output from a trace unit has several uses. It can be analyzed for:

- System development purposes, such as examining timing issues.
- Diagnosing and fixing bugs.

- PE profiling or performance analysis.

The following sections describe:

- *The attributes of PE tracing*.
- *External debug and self-hosted debug*.

1.1.1 The attributes of PE tracing

The attributes of PE tracing are:

- It is done in real time, which means that the operation of the PE is observable while it is running. For diagnostic purposes, this is useful because some types of bug and instances of erroneous behavior are only solvable by observing the system during runtime. In addition, because the PE trace can include cycle counts, it can be used for PE profiling purposes.
- It provides a method of debugging PEs that are deeply embedded within an SoC.
- Usually, it has no effect on the 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.

1.1.2 External debug and self-hosted debug

Figure 1-1 on page 1-18 shows a system that supports both external debug and self-hosted debug. Either methodology can be adopted.

External debug

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

This can be done using either:

- A dedicated trace port capable of sustaining the bandwidth of the trace, as shown in Figure 1-1 on page 1-18.
- 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

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.

Figure 1-2 shows an example of short-term on-chip capture with subsequent low speed export in a system that uses an Arm CoreSight ETB and a JTAG-DP.

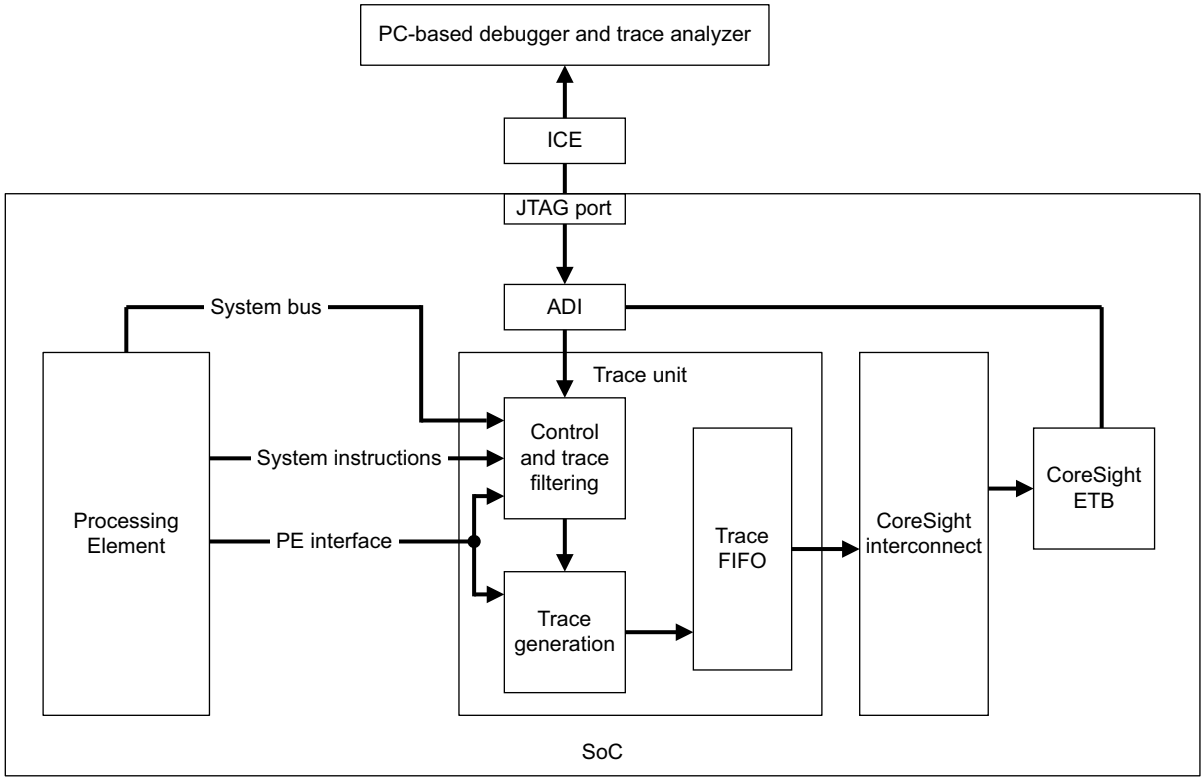


Figure 1-2 Example SoC with a trace unit and a dedicated trace buffer

Self-hosted debug

Self-hosted debug 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 usually involves either:

- Use of a dedicated on-chip buffer, such as the ETB offered by Arm CoreSight technology. If dedicated memory is used, a dedicated bus is also usually implemented between the trace unit and the dedicated memory. This means that PE tracing can be performed with zero or minimal effect on system behavior.
- 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. This means that tracing might affect system behavior, because the trace contends for system bus bandwidth with normal bus traffic.

1.2 Introduction to trace units

This section provides information about different aspects of trace unit operation:

- *Trace stream generation and compression techniques*.
- *Programming a trace unit before a trace run*.
- *Filtering of the trace on page 1-22*.
- *Tracing a PE on page 1-22*.
- *Trace unit resources on page 1-23*.
- *Trace unit powerdown support and low-power state on page 1-23*.
- *Sharing a trace unit between multiple PEs on page 1-23*.

1.2.1 Trace stream generation and compression techniques

A trace unit compresses the information that it obtains at its PE interface and outputs it as one or more trace streams that comprise multiple packets of encoded data.

A trace stream might be output from a trace unit over either a parallel data interface or a serial data interface. This depends on the implementation of the trace unit.

Compression techniques that are used include:

- The instruction trace stream does not contain an element for every executed instruction. Instead, the trace unit generates P0 elements in the instruction trace stream when certain types of instruction are executed. A P0 element acts as a signpost in the program flow, indicating that execution is proceeding along a given branch. 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 P0 elements can be encoded into a single P0 packet. See *Atom instruction trace packets on page 6-298*.
- 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 branch instructions, where the target address is encoded in the instruction itself.
- The trace unit can include a return stack. This contains information about the return address for particular instances of certain types of branch instruction. The trace analyzer maintains an independent copy of the return stack, which is based on the branch instructions it observes in the instruction trace stream. As a result, a trace analyzer can infer some return addresses, and it is possible for the trace unit to avoid generating some Address packets. See *Use of the return stack on page 5-222*

1.2.2 Programming a trace unit before a trace run

A trace unit includes facilities that can be programmed before a trace run, including:

- Filtering of the trace. See *Filtering of the trace on page 1-22*.
- Selecting and programming any *trace unit resources* that are required for the trace run, such as counters, comparators, and external inputs. See *Trace unit resources on page 1-23*. These resources can be selected and used to trigger filtering of the trace, or to signal to a trace analyzer that a particular event has occurred in the program the PE is executing.
- Turning on data tracing. Some trace applications require only the tracing of instructions, termed *instruction tracing*, whereas others require the tracing of both instructions and data transfers, such as data loads and stores. The tracing of data transfers is termed *data tracing*. If a trace unit implementation supports data tracing, it can be turned on if required.
- Selecting which types of instructions are traced explicitly. All trace unit implementations trace certain instruction types explicitly, such as branch instructions and `ISB` instructions, and the execution of other instruction types can be inferred from these instructions. However, if required, an implementation can include support for explicit tracing of other instruction types, such as data load and store instructions.

If the external debug model is adopted, the PC-based debugger provides the user interface to the trace unit, and can be used to program the trace unit facilities before each trace run. The debugger also decodes, analyzes, and post-merges the trace data with the program source code, to display the captured trace information.

Depending on the trace unit implementation, if the self-hosted debug model is adopted, there are two options for programming the trace unit. The first option is using memory-mapped accesses, and the second option is using system instructions, which are also known as *coprocessor accesses*.

1.2.3 Filtering of the trace

A trace unit is programmable to filter trace, providing two benefits:

- Only functions, data transfers, or sequences of code of interest are traced.
- The bandwidth of output trace is reduced.

1.2.4 Tracing a PE

A trace unit traces a PE by generating *trace elements*. The trace elements are then encoded into trace packets and output from the trace unit.

Some elements carry information that a trace analyzer requires to enable it to analyze the trace successfully, such as:

- Elements that contain information about which instructions, data, or events the trace unit is programmed to trace.
- Elements that show the context in which instructions are being executed.
- Elements that signal to a trace analyzer when there is a gap in the trace, and other elements that indicate why the gap has occurred. These enable the trace analyzer to take the appropriate action to maintain the integrity of the trace.
- Elements that enable a trace analyzer to synchronize trace streams if more than one trace stream is output.

Other elements either directly indicate program execution, or carry information about program execution, such as:

- Elements that indicate which branches are executed.
- Elements that indicate the execution of other instruction types, for example, if data tracing is implemented and enabled, elements that indicate load or store instructions.
- Elements that indicate exceptions, and returns from exceptions.
- Elements that indicate the addresses of instructions, and if data tracing is implemented and enabled, elements that contain data values, and elements that indicate addresses that data is transferred to or from.
- Speculation resolution elements that show whether traced instructions are:
  — Canceled because they were executed speculatively and the speculation was incorrect.
  — Committed for execution. A trace analyzer must only infer execution when a traced instruction has been committed for execution.
- Elements that show the results of condition code checks, to show whether traced conditional non-branch instructions have been executed, if tracing of conditional non-branch instructions is implemented and enabled.
- Elements that signal to a trace analyzer that a particular event has occurred in the program that the PE is executing.

  Program events are represented by *trace unit events*, that are activated by *trace unit resources*. For example, a trace unit might be programmed to signal when one of its address comparators becomes active as a result of the PE accessing a particular instruction address. In this case:
  — The address comparator is the trace unit resource.
  — The access that is performed is the program event.
  — The address comparator matching is the trace unit event.

In addition, there are elements that contain timing information, such as:

- Timestamp elements that contain global timestamp values.
- Cycle Count elements, that show counts of processor clock cycles.

1.2.5 Trace unit resources

A trace unit provides a range of resources that can be implemented and used to trigger tracing, or that can be used to program the trace unit to signal to a trace analyzer the occurrence of particular program events. A trace unit implementation might contain any, or all, of the following:

- Counters.
- A sequencer.
- External inputs.
- External outputs.
- Single instruction or data address comparators.
- Instruction or data address range comparators.
- Data value comparators.
- *Context identifier* (Context ID) comparators.
- *Virtual context identifier* comparators.
- PE comparator inputs.
- Single-shot comparator controls.

The architecture provides the option to implement a certain number of each resource type. For example, a simple design of a trace unit implementation might contain one counter, four PE comparator inputs, and two external outputs.

1.2.6 Trace unit powerdown support and low-power state

A trace unit might include powerdown support. Trace unit state is held in the trace unit registers, and can be saved before powering down the trace unit. See *Trace unit powerdown support on page 3-94*.

In addition, a trace unit implementation might include support for entering a low-power state. If a trace unit does support low-power state, then the low-power state is usually invoked whenever the PE being traced enters a low-power state.

1.2.7 Sharing a trace unit between multiple PEs

A trace unit might be shared between multiple PEs. This can provide an opportunity to reduce costs. If a trace unit is shared, one PE at a time can be selected to be traced. The trace unit must be disabled when changing the selected PE.

If the PE implements Armv8.4-Trace, the trace unit must not be shared between multiple PEs.

1.3 Introduction to the ETMv4 architecture

The ETMv4 architecture introduces the following changes from previous trace architectures from Arm:

- It supports addresses up to 64 bits wide.
- It supports the Armv8 architecture.
- In addition to instruction tracing, it provides optional support for:
  - Data tracing.
  - Event tracing.
  - Tracing of conditional non-branch instructions.
- It provides better compression than previous trace architectures from Arm.

The following sections describe:

- *Supported instruction sets*.
- *Impact on PE behavior*.
- *Trace unit resources*.
- *Possible functional configurations of an ETMv4 trace unit on page 1-25*.

1.3.1 Supported instruction sets

The ETMv4 architecture supports the following instruction sets:

- A64 in AArch64 state.
- A32 and T32 in AArch32 state.

Note

A32 and T32 are new names for what were the Arm and Thumb instruction sets in Armv7-A, Armv7-R, Armv7-M, and Armv6-M.

1.3.2 Impact on PE behavior

The ETMv4 architecture places no requirements on the impact that trace generation has on the 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 ETMv4 architecture does not prohibit this.

1.3.3 Trace unit resources

A trace unit provides resource selectors that are used to choose one or more of the trace unit resources. Up to 32 resource selectors are implemented, see *Selecting trace unit resources on page 4-171*.

The ETMv4 architecture provides the resources that are shown in Table 1-1.

Table 1-1 Resources that are provided by the ETMv4 architecture

Resource type	Number available	Notes
Counters	0-4	-
Sequencer state machine	0-1	-
External inputs	0-256	-
External input selectors	0-4	Each of these can select from up to 256 external inputs to be a trace unit resource.
External outputs	0-4	External outputs are used for event tracing and for signaling to a trace analyzer that a particular trace unit event has occurred. As mentioned in *Tracing a PE on page 1-22*, a trace unit event might represent a program event.

Table 1-1 Resources that are provided by the ETMv4 architecture (continued)

Resource type	Number available	Notes
Single address comparators	0-16	Single address comparators are implemented in pairs. One pair of single address comparators can be programmed to comprise one *address range comparator*. If data tracing is implemented, a single address comparator can be programmed to match on an instruction address or on a data address.[a]
Address range comparators	0-8	See *single address comparators* in this table. An address range comparator is programmed with an address range, so that it matches on any address within that range. If data tracing is implemented, the address range might be an instruction address range or a data address range.[a]
Data value comparators	0-8	These are used with *data address comparators*.
Context ID comparators	0-8	Each comparator can be one of the following: • Associated with one or more single address comparators. • Associated with one or more address range comparators. • Used on its own as a trace unit resource.
Virtual context identifier comparators	0-8	Each comparator can be one of the following: • Associated with one or more address comparators. • Associated with one or more address range comparators. • Used on its own as a trace unit resource.
PE comparator inputs	0-8	-
Single-shot comparator controls	0-8	Each control can be used with one or more address comparators to signal to a trace analyzer when an accessed instruction or data transfer is nonspeculative.

a. Single address comparators and address range comparators that are programmed to match on instruction addresses are called *instruction address comparators*. Single address comparators and address range comparators that are programmed to match on data addresses are called *data address comparators*.

1.3.4 Possible functional configurations of an ETMv4 trace unit

An ETMv4 *trace unit* is the hardware implementation of a particular functional configuration of the ETMv4 architecture.

An ETMv4 trace unit might support one of several different functional configurations that the ETMv4 architecture permits, giving a trade-off between trace unit functionality and trace unit cost. An implementation might contain all available options that the ETMv4 architecture offers, providing full instruction and data trace and including all resources and support for a trace unit low-power state. Alternatively, a trace unit might be implemented with only minimal functionality, providing only basic program flow trace. Between these two extremes, several intermediate functional configurations are possible. Appendix C *Recommended Configurations* contains some example functional configurations. All implementations support certain features but support for other features is optional. Table 1-2 on page 1-26 provides a summary.

Table 1-2 A summary of the features of an ETMv4 trace unit

Function	Always implemented	Optional	For more information:
Trace stream generation	The instruction trace stream.	For Arm R and M profile PEs, the data trace stream is optional. For Arm A profile PEs, the data trace stream is not permitted.	See *Separate instruction and data trace streams on page 2-33*.
Filtering	The ViewInst function, that is used to filter the instruction trace stream. If data tracing is implemented, the ViewData function is also implemented. The ViewData function is used to filter the data trace stream.	-	See: • *The instruction-based filtering model on page 4-118*. • *The data-based filtering model on page 4-131*.
Event tracing and external outputs	The number of external outputs that is implemented for indicating a trace unit event to a trace analyzer depends on the architecture version: • For ETMv4.2 or older, at least one external output is mandatory. • From ETMv4.3, external outputs are optional. There might be no external outputs.	The number of optional external outputs that can be implemented for indicating a trace unit event to a trace analyzer depends on the architecture version: • For ETMv4.2 or older, up to three optional external outputs can be implemented. • From ETMv4.3, up to four optional external outputs can be implemented.	See: • *External outputs on page 4-146*. • *Selecting trace unit resources on page 4-171*. • *Event instruction trace element on page 5-217*. • *Event data trace element on page 5-231*.
Powerdown support	• The TRCPDCR. • The TRCPDSR. In addition, the trace unit state can be saved before the trace unit is powered down, so that it can be restored when the trace unit is powered up again.	-	See: • *Trace unit powerdown support on page 3-94*. • *TRCPDCR, PowerDown Control Register on page 7-391*. • *TRCPDSR, PowerDown Status Register on page 7-392*.
Trace unit low-power state	-	Whether the trace unit supports low-power state.	See *Trace unit behavior on a PE low-power state on page 3-105*.

1.4 Terminology that is used in this document

This section contains:

1.4.1 General terms that are used in this document

Table 1-3 lists the general terminology this document uses.

Table 1-3 General terms that are used in this document

Term	Meaning
Trace unit	The hardware implementation that is used to generate the trace.
Instruction trace	PE trace that indicates program execution, such as branches taken, the execution of instructions, and exceptions and exception returns. Instruction trace might also contain timing information. Instruction trace contains information that a trace analyzer requires to enable it to analyze the trace.
Data trace	PE trace that carries information about data transfers that the PE performs. Data trace might also contain timing information. Data trace contains information that a trace analyzer requires to enable it to analyze the trace.
Event trace	PE trace that indicates certain events in the program that the PE is executing. The program events to be indicated are selected before a trace run.
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. If data tracing is implemented and enabled, the trace unit might also be generating data trace. In addition, the trace unit might also be generating event 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 filtering function is prohibiting instruction tracing. However, the trace unit might be generating event trace in the instruction trace stream. In addition, if data tracing is implemented and enabled, the trace unit might be generating data trace and event trace in the data trace stream. **Note** A trace unit can only generate data trace for instructions that are traced.
ViewData active	All the following are true: • The trace unit implementation supports data tracing. • The trace unit has been programmed and is enabled. As part of this process, data tracing has been enabled. • The ViewData data trace filtering function is permitting data tracing, therefore the trace unit is generating data trace. If ViewInst is active, the trace unit is also generating instruction trace and in addition, the trace unit might also be generating event trace in both trace streams. **Note** A trace unit can only generate data trace for instructions that are traced.

Table 1-3 General terms that are used in this document (continued)

Term	Meaning
ViewData inactive	All the following are true: • The trace unit implementation supports data tracing. • The trace unit has been programmed and is enabled. As part of this process, data tracing has been enabled. • The trace unit is not generating any data trace because the ViewData filtering function is prohibiting data tracing. If ViewInst is active, the trace unit might be generating instruction trace. In addition, the trace unit might be generating event trace in both trace streams.
Trace buffer overflow	Buffering inside the trace unit is unable to capture more trace data.
Trace analyzer	A tool that takes the trace streams and analyzes them to determine PE execution. This tool can be part of a self-hosted debug environment, or an external debug tool.
Trace run	When the trace unit is enabled, it starts a trace run.
Element stream	A stream of trace elements that a trace unit generates. Trace elements are encoded into trace packets.
Packet stream	A stream of trace packets output by a trace unit.
Analysis of the trace 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 stream might also contain trace elements that contain timing information. This term is distinct from *analysis of program execution*.
Analysis of program execution	A trace analyzer contains a program image for the program that the PE is executing. When a trace analyzer analyzes trace elements that directly indicate program execution, and elements that carry information about program execution, it uses the program image to ascertain the instructions being executed. This term refers to that process.
Speculation depth	The number of traced P0 elements that are uncommitted. When a P0 element is traced, it remains speculative until it is either canceled or committed for execution. For more information, see *About instruction trace P0 elements on page 2-35*.
Program image	A copy of the compiled executable that is being executed on the PE being traced.

1.4.2 Terms that are used to describe ETMv4 architectural features

Table 1-4 lists the architectural terminology this document uses.

Table 1-4 Terms that are used to describe ETMv4 architectural features

Term	Meaning
Implemented	The feature is included in the implementation.
Not implemented	The feature is not included in the implementation.
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.

Table 1-4 Terms that are used to describe ETMv4 architectural features (continued)

Term	Meaning
Disabled	The feature is either not implemented, or is implemented but has been programmed to be disabled during the trace run.
Active	The feature is implemented and enabled, and the trace unit is in a state that the feature is programmed to operate in.
Inactive	The feature is either not implemented or is disabled, or the trace unit is in a state that the feature is programmed not to operate in.

Some usage examples of these terms are:

Implemented and not implemented

For example, the ETMv4 architecture supports the implementation of up to eight pairs of single address comparators, pairs 0-7. If an implementation contains only four pairs of single address comparators, then pairs 0-3 are implemented and pairs 4-7 are not implemented.

Implemented but disabled

For example, cycle-counting might be included in an implementation but it might not be required for a particular trace run. Therefore, if the trace unit is programmed not to use cycle counting during that trace run, the feature is implemented but disabled.

Implemented and enabled, and active or inactive

For example, branch broadcasting might be included in an implementation and might be required for a particular trace run, but only when the PE executes instructions from a particular memory region. In this case, if the trace unit is programmed to use branch broadcasting for these memory regions during a trace run, then:

- When the program is executing from inside the memory region, branch broadcasting is implemented, enabled, and active.
- When the program is executing from outside the memory region, branch broadcasting is implemented and enabled, but inactive.

1.4.3 Terms that are used to describe resets

Table 1-5 lists the reset terminology this document uses.

Table 1-5 Terms that are used to describe resets

Term	Meaning
PE reset	The PE has been through a reset procedure and has restarted execution from its reset state. This does not reset any trace unit registers, unless one of the following occurs at the same time: • A *trace unit reset*. • An *external trace reset*.
Trace unit reset	This resets all trace unit registers that are located in the trace unit core power domain. These registers include: • All trace unit *trace registers*. • Some trace unit *management registers*. These are TRCOSLAR and TRCOSLSR. *Register map overview on page 4-164* shows which registers are trace registers and which are management registers. This reset is usually only applied on a trace unit core power domain powerup. For more information, see *Trace unit behavior on a trace unit reset on page 3-98*.
External trace reset	This resets all trace unit registers that are located in the trace unit debug power domain. These include: • All trace unit management registers except TRCOSLAR and TRCOSLSR. *Register map overview on page 4-164* shows which registers are trace registers and which are management registers. This reset is usually only applied on a trace unit debug power domain powerup. For more information, see *Trace unit behavior on a trace unit reset on page 3-98*.

Chapter 2
About the Trace Streams

This chapter describes the trace streams that the trace unit generates. It contains the following sections:

2.1 The tracing flow

A ETMv4 trace unit traces *processing element*, or PE, execution by generating *trace elements*. The ETMv4 architecture defines the generation of these trace elements from the execution of the PE.

An ETMv4 trace unit can generate two trace element streams:

- An instruction trace element stream.
- A data trace element stream, if data tracing is implemented and enabled.

These two streams of elements are then encoded into two streams of trace packets:

- A stream of instruction trace packets.
- A stream of data trace packets.

The encoding process uses compression techniques to reduce the amount of trace that is generated, so that there is improved transmission efficiency and a reduced trace storage requirement. For more information, see *Trace stream generation and compression techniques on page 1-21*.

On receiving a trace stream, an analyzer decodes the data and then analyzes each trace element to infer program execution.

Figure 2-1 shows the tracing flow.

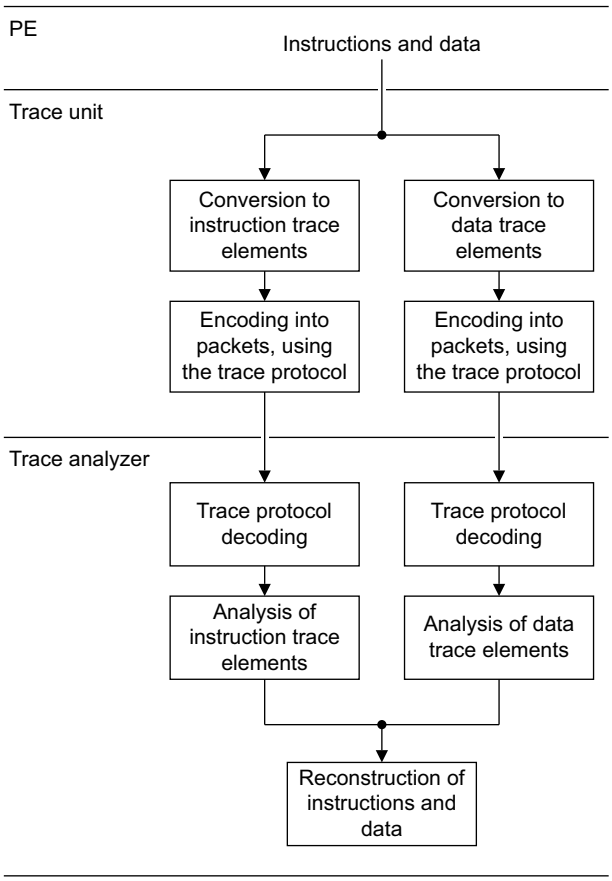


Figure 2-1 The tracing flow

2.2 Separate instruction and data trace streams

An ETMv4 trace unit outputs an instruction trace stream and, if implemented and programmed to do so, a data trace stream. This is shown in Figure 2-2:

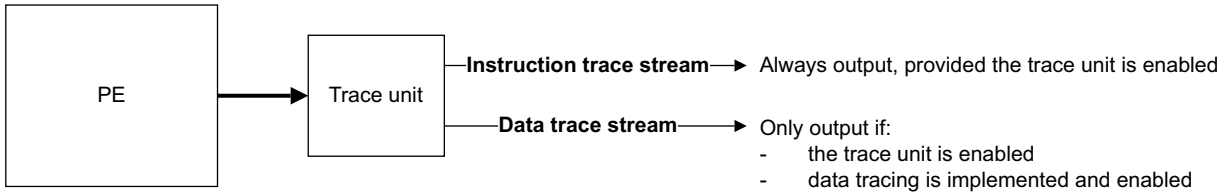


Figure 2-2 Separate instruction and data trace streams

Each trace stream can be filtered:

- The trace unit includes a ViewInst function that can be used to filter the instruction trace stream.
- If data tracing is implemented, the trace unit includes a ViewData function that can be used to filter the data trace stream.

If data tracing is implemented, the data trace stream is enabled by setting either or both of the following to 1:

- TRCCONFIGR.DA. When this bit is set to 1, whenever the PE initiates a data load or store transfer and if ViewData permits it to be traced, the address of that data transfer is output in the data trace stream:
  - If the transfer is a data load, the address that is output is the address that the data is loaded from.
  - If the transfer is a data store, the address that is output is the address that the data is stored to.
- TRCCONFIGR.DV. When this bit is set to 1, whenever the PE initiates a data load or store transfer and if ViewData permits it to be traced, the data value of that data transfer is output in the data trace stream.

Note

The data value output is the view of the register in the PE, not the view that is loaded from or provided to the memory system.

If TRCCONFIGR.INSTP0 is set to 0 and TRCCONFIGR.DA or TRCCONFIGR.DV are non-zero, the behavior of the trace unit is CONSTRAINED UNPREDICTABLE:

- Data trace might or might not be generated.
- Event tracing in the data trace stream might or might not occur.
- ATB triggers in the data trace stream might or might not occur.

If both TRCCONFIGR.DA and TRCCONFIGR.DV are set to 1, the data trace stream contains both the address value and data value of each data transfer that ViewData permits to be traced. If these bits are both set to 0, then data tracing is disabled and the data trace stream is not output.

For more information, see *Data address tracing on page 2-84* and *Data value tracing on page 2-84*.

The instruction trace stream comprises instruction trace elements, which are grouped into the following categories:

- P0 elements.
- All other instruction trace elements.

The data trace stream comprises data trace elements, which are grouped into the following categories:

- P1 elements.
- P2 elements.
- All other data trace elements.

Figure 2-3 on page 2-34 shows the element types that are included in each of the categories, for each trace stream.

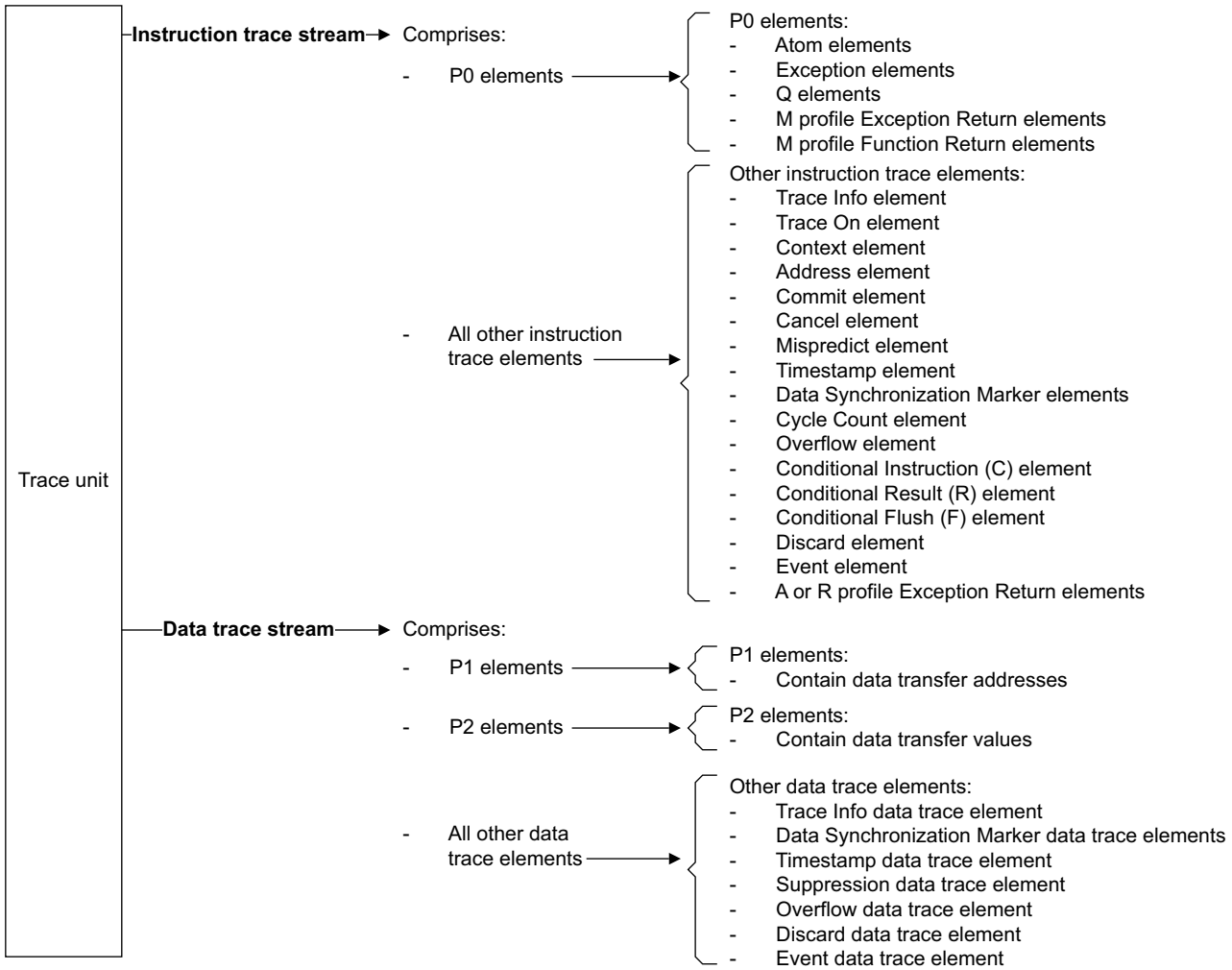


Figure 2-3 Elements that comprise each trace stream

The ETMv4 architecture does not require the instruction and data trace streams to be captured by the same trace capture device, neither does the architecture require any ordering of the capture of the instruction and data trace streams. Each stream is permitted to be transported and captured independently.

For most normal usage scenarios, the instruction stream must be able to successfully interpret the data information in the data trace stream.

The remainder of this section is organized as follows:

- *About instruction trace elements*.
- *About data trace elements on page 2-38*.
- *Associating data trace elements with instruction trace elements on page 2-38*.

2.2.1 About instruction trace elements

Elements in the instruction trace stream contain the following information:

- Full instruction execution information, including conditional branch instructions.

Note

Some trace units include support for tracing conditional non-branch instructions. If the tracing of conditional non-branch instructions is implemented and enabled, then instruction trace elements also contain execution information about conditional non-branch instructions. For more information, see *Conditional instructions tracing on page 2-84*.

- Indications of when the PE takes an exception, and returns from an exception. Exceptions reported in the trace include:
  - Architectural exceptions, which are defined by the PE architecture.
  - Exception occurrences that are microarchitecture-specific.

The following information is provided about each instruction executed:

- The virtual address (VA).
- The instruction set.
- The Exception level for AArch64 execution, or the privilege level for Armv7 or AArch32 execution.
- The Security state.
- The *context identifier*, Context ID, if enabled.
- The *Virtual context identifier*, if enabled.
- The condition code check result for conditional branch instructions.
- The condition code check result for other types of conditional instructions, if tracing of these instructions is implemented and enabled.

About instruction trace P0 elements

A trace unit generates a P0 element in the instruction trace stream whenever any of the following occurs:

- The PE takes an exception or enters Debug state.
- The PE returns from an exception, for Armv6-M, Armv7-M, and Armv8-M PEs.
- The PE executes a function return instruction, for Armv8-M PEs for which Data trace is implemented and enabled.
- The PE executes one of the following types of instruction:
  - A direct branch instruction.
  - An indirect branch instruction.
  - An Instruction Synchronization Barrier, `ISB`.
  - A `WFI` or `WFE` instruction, if TRCIDR2.WFXMODE is `0b1`.

  These instruction types generate a P0 element regardless of whether:
  - They pass or fail their condition code check.
  - They are part of an IT block.

Note

Appendix F *Instruction Categories* lists the instructions for each of the instruction types that are mentioned in this section.

In addition, if an implementation includes support for data tracing, the trace unit can be programmed to generate a P0 element whenever the PE executes:

- A data load instruction.
- A data store instruction.

This option is enabled by programming TRCCONFIGR.INSTP0 so that either:

- Load instructions also generate P0 elements.
- Store instructions also generate P0 elements.
- Both load instructions and store instructions generate P0 elements.

Tracing load or store instructions as P0 elements is termed *explicit tracing of load or store instructions*. When load or store instructions are traced explicitly, and if data tracing is implemented and enabled, a trace unit uses a key system to show the relationships between data transfers and their parent load or store instructions. See *Relationships between P0, P1, and P2 elements on page 2-38*.

This specification defines the following terms for describing sequences of instructions:

Batch of instructions

A batch of instructions is a sequence of one or more instructions that can start on any instruction and ends on any instruction, but always ends if a P0 instruction is encountered. All of the instructions in a batch of instructions are at consecutive addresses. The size of a batch of instructions is chosen by the PE and might vary at run-time. The simplest batch of instructions is just a single instruction.

Block of instructions

A block of instructions is a sequence of instructions from the target of a P0 instruction or exception, up to and including the next P0 instruction or exception. All of the instructions in a block of instructions are at consecutive addresses.

A block of instructions consists of one or more complete batches of instructions.

Sequential block of instructions

A sequential block of instructions is a sequence of instructions from the target of a P0 instruction or exception, up to and including the next P0 instruction or exception, where the bounding P0 instructions can only be load or store instructions if they are also a branch instruction.

All of the instructions in a sequential block of instructions are at consecutive addresses.

A sequential block of instructions can only include at most one branch instruction, and that branch instruction is always the last instruction.

A sequential block of instructions might contain zero or more load or store instructions, including when those loads and stores are defined as P0 instructions.

A sequential block of instructions consists of one or more complete blocks of instructions.

When load and store instructions are not selected as P0 instructions, a sequential block of instructions is the same as a single block of instructions.

If load and store instructions are not traced explicitly, then they are traced *implicitly*, as part of a block of instructions as shown in Figure 2-5 on page 2-37. In this case, a trace analyzer can infer the execution of load or store instructions from other P0 elements, but it cannot associate any data transfers with the traced load and store instructions, which means that:

- A trace unit that implements data tracing must also implement explicit tracing of load and store instructions.
- On enabling data tracing, the trace unit must also be programmed to trace either data load instructions, data store instructions, or both, explicitly.

Note

A trace unit might implement explicit tracing of load and store instructions but not implement data tracing. Data load and store instructions can be traced explicitly even if the data addresses and data values of associated data transfers are not required.

For more information, see *Explicit tracing of data load and store instructions on page 2-84*.

To minimize the quantity of trace that is generated, the trace unit does not generate P0 elements for any other instruction types. Each P0 element implies the execution of all instructions from the target of the previous P0 element, up to and including the instruction indicated by the present P0 element. Therefore, a P0 element can indicate a block of instructions, as shown in Figure 2-4 on page 2-37 and Figure 2-5 on page 2-37.

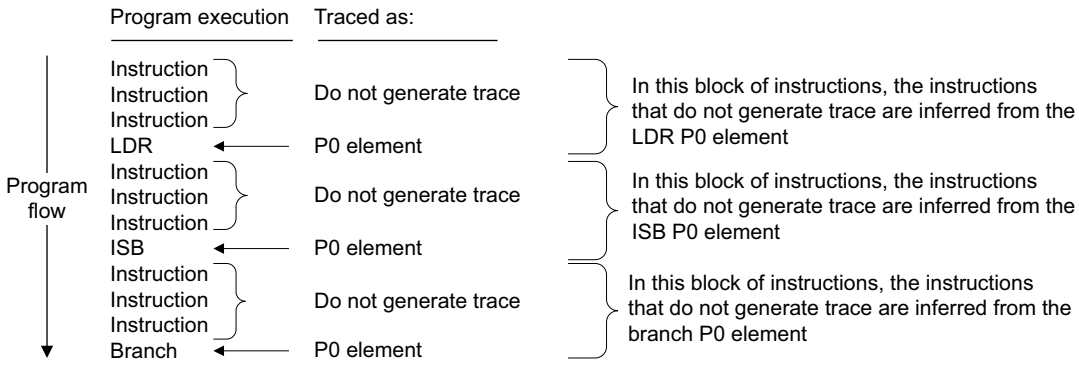


Figure 2-4 How P0 elements can indicate blocks of instructions

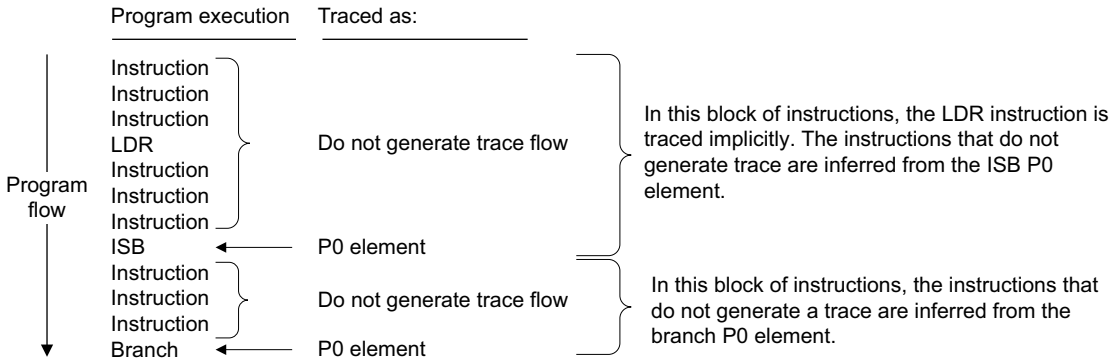


Figure 2-5 The same trace flow when data tracing is not implemented or not enabled

All P0 elements are always traced speculatively, and are then explicitly committed or canceled by subsequent Commit or Cancel elements.

If a trace unit is exposed to speculative execution, when it generates a P0 element, that P0 element might represent either speculative execution or nonspeculative execution, because a trace unit traces instructions that have been executed speculatively in the same way as all other instructions. However, when the status of an instruction is known, that is, when it is known whether the instruction has been committed for execution or canceled because of mis-speculation, the trace unit generates an element to indicate that status. For more information, see *Trace behavior on speculative execution on page 2-70*.

A PE might execute instructions out-of-order. When a trace unit is tracing a PE that can perform out-of-order execution, instructions and exceptions are always traced in program order.

Note

Nonspeculative execution is also referred to as *architectural* execution.

Branch Future

Armv8.1-M includes branch future instructions as part of the LOB extension. Branch future instructions are used to give the PE advanced notification of a branch instruction that will be executed. The details of the branch are used to initialise dedicated hardware to perform the branch operation just before the real branch instruction is reached. As the PE is notified in advance, the use of branch future instructions can eliminate the pipeline bubbles associated with the branch and therefore increase performance.

For example, consider the execution of the following simple subroutine:

```
start:
      BFX b_label, LR        // Set up BF at b_label
```

```
      ADD r0, r0, r1
      ADD r0, r0, r2
      ADD r0, r0, r3
      // This is the BF branch point
b_label:
      BX LR                  // Executed if LO_BRANCH_INFO invalid
```

Here the effect of the BFX instruction is to add a implicit branch operation immediately before the BX LR, and the BX LR instruction is not necessarily even fetched from memory. This brings challenges to tracing the PE operation, requiring changes in both trace analyzer synchronization and the operation of analyzing trace.

When tracing of branch future is implemented, the trace unit generates trace elements for both the execution of the branch future instruction and the subsequent implicit branch.

2.2.2 About data trace elements

If data tracing is supported and enabled, the trace unit outputs a data trace stream. Elements in the data trace stream contain the following information:

- The data address and data value of each data transfer instruction.

The following information is provided about each data transfer:

- The virtual address (VA), if data address tracing is enabled.
- The data value, if data value tracing is enabled.
- The endianness, if data address tracing is enabled.
- For instructions that perform multiple data transfers, a transfer index that indicates the transfer that is performed.

Note

- As documented in *Separate instruction and data trace streams on page 2-33*, the data value that is provided is the view of the register in the PE, not the view that is provided to the memory system.
- Information about the architectural size of a data transfer is not provided, because this can usually be inferred from the parent data transfer instruction.
- ETMv4 does not support data trace on Armv7-A, Armv8-A and Armv8-M PEs.

The elements that provide data transfer information in the data trace stream can be associated, by using a key mechanism, with their parent elements in the instruction trace stream. For more information, see *Associating data trace elements with instruction trace elements*.

2.2.3 Associating data trace elements with instruction trace elements

This section describes how certain elements in the data trace stream can be associated with their parent elements in the instruction trace stream. It contains the following subsections:

- *Relationships between P0, P1, and P2 elements*.
- *About P0, P1, and P2 keys on page 2-40*.

Relationships between P0, P1, and P2 elements

As shown in Figure 2-3 on page 2-34:

- The instruction trace stream comprises the following element types:
  - P0 elements.
  - All other instruction trace elements.
- The data trace stream comprises the following element types:
  - P1 elements.
  - P2 elements.
  - All other data trace elements.

P0 elements show that the PE has executed a certain type of instruction. See *About instruction trace P0 elements on page 2-35*.

P1 elements are only output if both:

- Data tracing is enabled, that is, if either TRCCONFIGR.{DA,DV} are set to 1, or both are set to 1. See *Separate instruction and data trace streams on page 2-33*.
- The data transfer instruction is traced explicitly. See *Explicit tracing of data load and store instructions on page 2-84*.

If data address tracing is enabled, that is, if TRCCONFIGR.DA is set to 1, the P1 element contains the data address of the transfer.

P2 elements contain the data values of data transfers, and are only output if both:

- Tracing of the data values of data transfers is enabled, that is, if TRCCONFIGR.DV is set to 1.
- If the P1 element was traced.

A relationship exists between a P0, a P1, and a P2 element. For example, if the PE executes a load instruction that results in multiple data transfers, then the load instruction is traced as a P0 element, the addresses of the data transfers are traced as P1 elements, and the data values of the data transfers are traced as P2 elements. P0 elements in the instruction trace stream are associated with P1 and P2 elements in the data trace stream by using keys, as shown in Figure 2-6:

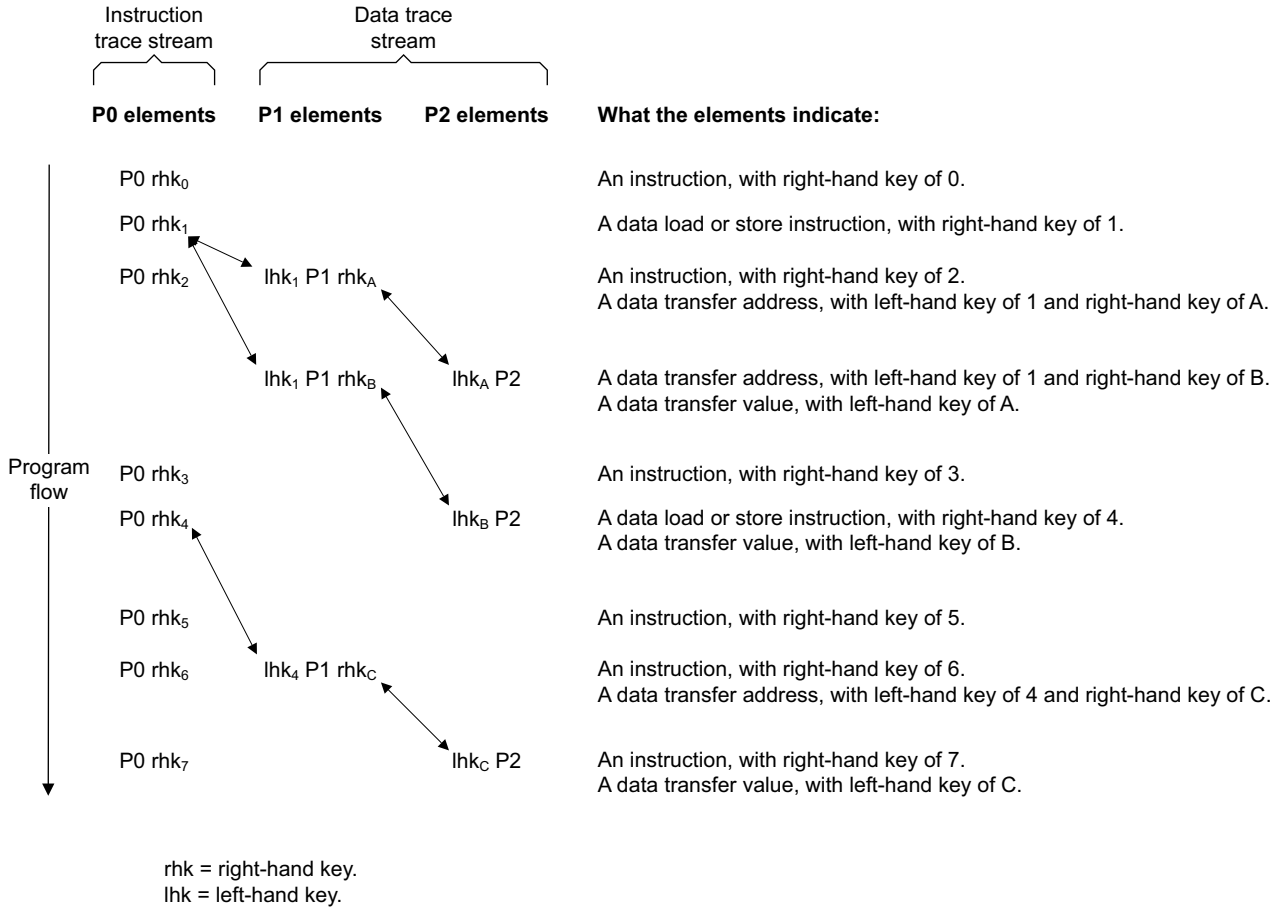


Figure 2-6 Association of P0 instruction trace elements with P1 and P2 data trace elements

As Figure 2-6 shows:

- P0 elements have only right-hand keys.
- P1 elements have both right-hand and left-hand keys.

- P2 elements have only left-hand keys.

The keys define parent-child relationships between elements. The right-hand key of a parent P0 element matches the left-hand key of a child P1 element. Similarly, the right-hand key of a parent P1 element matches the left-hand key of a child P2 element. This means that:

- A P0 element can only be a parent element.
- A P1 element can simultaneously be a child element to a P0 element, and a parent element to a P2 element.
- A P2 element can only be a child element.

A P0 element can have multiple child P1 elements. However, a P1 element can only have one child P2 element. Each child element, regardless of whether it is a P1 child element or a P2 child element, only has one parent element. For example, a P0 element might have three child P1 elements, and each of those child P1 elements might have one child P2 element, but each P2 element only has one parent P1 element, and each P1 element only has one parent P0 element.

A child element is only traced if its parent element is traced. This means that for a P2 element to be traced, the parent P1 element must also be traced. Similarly, for a P1 element to be traced, the parent P0 element must also be traced.

A P2 element is always traced after its parent P1 element in the trace stream.

About P0, P1, and P2 keys

A PE that can speculatively execute instructions might execute instructions out-of-order. When a trace unit is tracing a PE that can perform out-of-order execution:

- Instructions and exceptions are traced as P0 elements and are always traced in program order.
- The addresses of data transfers are traced as P1 elements, and can be traced out of program order, and out of order relative to their parent P0 elements.
- The data values of data transfers are traced as P2 elements, and can be traced out of program order, and not in the same order as their parent P1 elements.

The key system means that the data addresses of data transfers, and the data values of data transfers, can be associated with the correct instructions.

The keys that are used to associate P0 and P1 elements are independent of the keys that are used to associate P1 and P2 elements.

The value of the right-hand key for a P0 element is one more than the value of the right-hand key of the previous P0 element. For example, if a P0 element is generated with a key value of six, then the next new P0 element to be generated has a key value of seven. This new key value applies even if the new P0 element has no child elements. For P1 elements, the value of the right-hand key for a P1 element is usually one more than the value of the right-hand key for the previous P1 element. However, this is not mandatory and an implementation might not follow this rule. Left-hand keys, for P1 and P2 elements, match the key values of the parent P0 and P1 elements.

The maximum number of keys available between each Pn stage is IMPLEMENTATION DEFINED. When a key value reaches the maximum key value, it wraps around to begin again at the first key value.

In the case of P1 elements:

- A right-hand key cannot be reused until all child P2 elements have been output.

In the case of P0 elements:

- A right-hand key cannot be reused until all child P1 elements have been output.
- If all child P1 elements have been output but some grandchild P2 elements remain, the right-hand key can be reused. For example, a P0 element that has a right-hand key value of five might have two child P1 elements and two grandchild P2 elements. If both of the child P1 elements have been output, the right-hand key value of five can be reused, even if one or both of the grandchild elements have not been output.

  The reason for this is that the keys used to associate P1 and P2 elements are independent of the keys that are used to associate P0 and P1 elements. If one or more of the grandchild elements have not been output, the left-hand keys of those elements associate each P2 element only with a parent P1 element, not with a

grandparent P0 element. This means that the correct relationships between the grandparent P0 element, the parent P1 elements, and the grandchild P2 elements can still be ascertained, even if the right-hand key of the P0 element is reused.

2.3 Handling the trace streams

The trace streams can be handled separately. For example, one trace stream might be exported off-chip over a trace port, while the other trace stream is stored in an on-chip buffer for low-speed export later on.

This has the advantage that, if the amount of trace that is produced instantaneously exceeds the bandwidth of the trace port, some of the trace can be stored temporarily on-chip in a buffer, and then exported separately.

Figure 2-7 shows an example of how the trace streams might be handled.

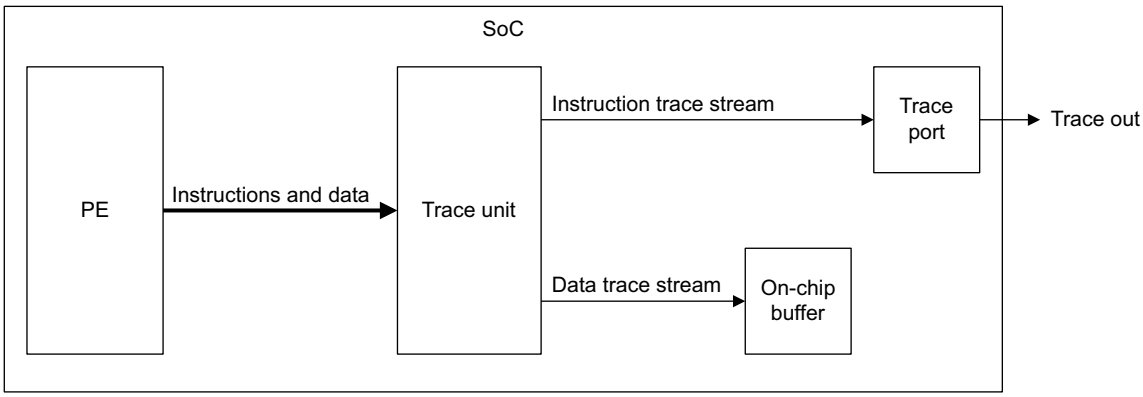


Figure 2-7 An example of how the trace streams might be handled

2.4 Synchronizing the instruction and data trace streams

This section describes how to synchronize the instruction and data trace streams. It contains the following subsections:

- *Trace analyzer operation*.
- *Aligning Data Synchronization Markers on page 2-45*.
- *Trace analyzer pseudocode on page 2-46*.
- *Trace unit operation on page 2-50*.
- *Examples on page 2-51*.

2.4.1 Trace analyzer operation

To match P1 elements successfully with their parent P0 element, the trace analyzer must follow a specific algorithm that defines how to search for the matching P0 element using the Data Synchronization Marker elements in both the instruction and the data trace stream, together with the P0 right-hand keys and P1 left-hand keys. This algorithm requires the trace analyzer to remember the most recent successfully matched P0 element, LME, the Last Matched Element. For some search operations, the algorithm defines a search space after the LME limits where a matching P0 element exists.

The end of the search space is defined as the P0 element after the LME which has a P0 right-hand key with the value ((LME.rhkey + RoundDown(TRCIDR9.NUMP0KEY/2)) modulo TRCIDR9.NUMP0KEY).

The first step in the algorithm aligns the data and instruction trace streams using global timestamps and the *Numbered Data Synchronization Markers*, NDSM. This procedure is described in the `InitialDataAlignment()` function. For details of the `InitialDataAlignment()` function, see *Trace analyzer pseudocode on page 2-46*.

It is possible that, as a result of Trace buffer overflows, not all Data Synchronization Markers can be paired. A data trace buffer overflow might mean that one or more Data Synchronization Markers in the instruction trace stream do not have a corresponding marker in the data trace stream. Similarly, an instruction trace buffer overflow might mean that one or more Data Synchronization Markers in the data trace stream might not have a corresponding marker in the instruction trace stream. In some scenarios, this means that it might not be possible to associate some P1 elements with their parent P0 element.

On recovery from a data trace buffer overflow, the first Data Synchronization Marker must be a Numbered Data Synchronization Marker to ensure that P1 elements can be associated with their parent P0 elements as soon as possible after the overflow recovery.

After the Data Synchronization Markers have been aligned, the `MatchData()` function iterates over the data trace stream to match all P1 elements to their parent P0 elements. For details of the `MatchData()` function, see *Trace analyzer pseudocode on page 2-46*.

For more information about the Data Synchronization Marker elements, see the appropriate section in *Descriptions of instruction trace elements on page 5-188*.

For each P1 element in the data trace stream, the algorithm to determine the parent P0 element in the instruction stream is as follows:

1. If the P1 element in the data trace stream is the first P1 element after a Data Synchronization Marker:
   a. Find the corresponding Data Synchronization Marker in the instruction trace stream.
   b. Search backwards from the Data Synchronization Marker in the instruction trace stream until a P0 element is found where the right-hand key matches the left-hand key of the P1 element.
   c. When a match is found:
      - If the match is the first P1 element that is analyzed, remember the position of the P0 element. This is now the LME.
      - If the match is not the first P1 element that is analyzed, and the P0 element occurs after the existing LME, then the LME is set to the P0 element.

2. If this is not the first P1 element after a Data Synchronization Marker:
   a. Starting at the LME, search forwards and look at each instruction trace element until the trace analyzer:
      - Finds a Cancel element. In this case, proceed to step 2b.
      - Finds a P0 element that has a right-hand key with a value that is after the end of the search space. In this case, proceed to step 2b.
      - Finds a P0 element whose right-hand key matches the left-hand key of the P1 element. If this occurs, set the LME to this new P0 element. A match has been found so the algorithm is complete for this P1 element.
      - Finds an Overflow or Discard element. If this occurs, the P1 element cannot be matched to a P0 element and this P1 element must be discarded.
      - Reaches the end of the captured trace. If this occurs, the P1 element cannot be matched to a P0 element and this P1 element must be discarded.
   b. If no matching P0 element is found while searching forwards, then, starting at the element before the LME, search backwards until the trace analyzer finds a P0 element whose right-hand key matches the left-hand key of the P1 element.

      Note

      - This does not update the LME.
      - This step must not be performed if one of the following is encountered while searching forwards in step 2a:
        — An Overflow element.
        — A Discard element.
        — The end of the captured trace.

When searching backwards in step 1b and step 2b, the search must terminate if any of the following are found:
- The start of the captured trace.
- An Overflow element.
- A Discard element.

If the start of the captured trace is found while searching backwards:

- The P1 element cannot be matched with a P0 element. Assume that the P1 element is associated with a P0 element before the start of the captured trace.
- Discard the P1 element.
- Set the LME to the start of the captured trace.
- Set the end of the search space that is based on the LME having the right-hand key value of the left-hand key of the P1 element.

If an Overflow or Discard element is found while searching backwards:

- This P1 element cannot be matched with a P0 element. Assume that the P1 element is associated with a P0 element before the Overflow or Discard element.
- Discard the P1 element.
- The LME is UNKNOWN.
- Following these steps, start the algorithm again from the Numbered Data Synchronization Marker, NDSM, in the data trace stream.

To perform this algorithm, a Data Synchronization Marker must be present in the data trace stream before the first P1 element. In addition, analysis of the P1 elements cannot begin until the first P1 element after the Data Synchronization Marker.

These rules mean that the LME only progresses forwards down the instruction trace element stream.

2.4.2 Aligning Data Synchronization Markers

To align Data Synchronization Markers, it is necessary to insert global timestamps into both the instruction and data trace streams.

A *Numbered Data Synchronization Marker*, NDSM, is inserted into both streams after each trace synchronization point, and the trace unit requests a timestamp to be inserted into both streams after each trace synchronization point. This enables a trace analyzer to uniquely pair an NDSM in the data stream with the corresponding NDSM in the instruction trace stream.

To align Data Synchronization Markers, an NDSM in the data trace stream must be paired with corresponding NDSM in the instruction trace stream using the timestamps in both streams. After this, all other Data Synchronization Markers, including NDSMs, are trivially aligned because, as a result of this first mapping, there is then a direct one-to-one mapping of Data Synchronization Markers between both streams. If a trace buffer overflow occurs in either the instruction or data trace stream, the Data Synchronization Marker alignment must be performed again because one or more Data Synchronization Markers might have been lost in either stream.

An example of a procedure for aligning Data Synchronization Markers involves:

1. Finding the first NDSM in the data trace stream, and remember the number N.
2. Determining the range of timestamps in which the data trace NDSM was generated:
   - This can be done by searching forwards in the data trace stream for the first global timestamp after the NDSM. The value of this timestamp is Y.

     Note

     This timestamp might not exist if the NDSM occurs very close to the end of the captured trace.

   - This can be done by searching backwards in the data trace stream for the first global timestamp before the NDSM. The value of this timestamp is X.

     Note

     This timestamp might not exist if the NDSM occurs very close to the start of the captured trace.

   - The NDSM was generated between timestamp values X and Y. If X is not known, the NDSM occurred before Y. If Y is not known, the NDSM occurred after X. If both X and Y are not known, it might not be possible to align the Data Synchronization Markers.
3. In the instruction trace stream, find an NDSM that has the following properties:
   - The value N.
   - The first timestamp element after the NDSM has the value B, where $B \geq X$.

     Note

     There might not be a timestamp after the NDSM if the NDSM occurs close to the end of the captured trace.

   - The first timestamp element before the NDSM has the value A, where $A \leq Y$.

     Note

     There might not be a timestamp before the NDSM if the NDSM occurs very close to the start of the captured trace.
4. The next Data Synchronization Marker in the data trace stream is paired with the next Data Synchronization Marker in the instruction trace stream. This step is repeated for all subsequent Data Synchronization Markers in the data trace stream:
   - If an Overflow element is found in the data trace stream, then this whole procedure must be restarted, starting at the first NDSM in the data trace stream after the Overflow.

- When searching for a Data Synchronization Marker in the instruction trace stream, if an Overflow element is found in the instruction trace stream, then this whole procedure must be restarted, starting at the next NDSM.

Step 3 of this procedure might not always produce a pair of NDSMs, for example because one of the following has occurred:

- The instruction trace containing the corresponding NDSM has not been captured.
- An overflow has occurred in the instruction trace stream and the corresponding NDSM was lost.
- Fewer timestamps than are required to find a unique NDSM were inserted. This is unlikely to occur, because the insertion of an NDSM is always accompanied by a request to insert a timestamp, meaning that typically there are as many timestamps as required.
- The NDSM is very close to the start or the end of the captured trace and there is no timestamp before or after the NDSM in one or more of the trace streams, making a unique match impossible.

In the case where not enough timestamps were inserted, the following techniques might be used to determine the correct match:

- Finding a matching pair for a subsequent data trace NDSM. The trace analyzer can then work backwards to the first NDSM to pair up the earlier NDSM.
- If the available timestamps indicate that one instruction trace NDSM is substantially closer to the data trace NDSM, this is likely to be the correct matching pair.

2.4.3 Trace analyzer pseudocode

The following pseudocode can be used to analyze the trace.

```
// Trace Analyzer pseudocode
// =========================

//
// Global types and enumerations
//
enumeration i_type {P0,UDSM,NDSM,OTHER};
enumeration d_type {P1,UDSM,NDSM,OTHER};

type i_element is (i_type el_type, integer rhkey, array integer p1s[]);
type d_element is (d_type el_type, integer lhkey);

//
// Global variables
//
// Arrays of elements for the instruction and data streams.
array i_element InstStream[0..n];
array d_element DataStream[0..n];

// Tracks the last known position in the data stream.
integer dpos = 0;

// Track the last matched P0 element
integer lme = UNKNOWN;
integer lme_key = UNKNOWN;

boolean first_p1_after_dsm = false;
integer data_last_dsm = UNKNOWN;



//
// InitialDataAlignment() aligns the instruction and data trace streams
// and matches the first P1 element with its parent P0 element.
//
```

```
InitialDataAlignment()

  // Align data sync marks in both streams using timestamps
  // and numbered data sync marks.
  dpos = AlignDataSyncMarks();

  // From the start of the data trace stream,
  // find the first data sync mark (numbered or un-numbered).
  dpos = DataFindDSM(dpos);

  // The next P1 will be the first after a DSM
  first_p1_after_dsm = true;
  data_last_dsm = dpos;


//
// Main function to match P1 elements to their parent P0 element
//
MatchData()
  integer p1key;

  while (dpos < SizeOf(DataStream)) do

    // Move through the DataStream until we get a P1 element.
    (p1key,dpos) = DataFindNextP1(dpos);

    if (first_p1_after_dsm) then
      integer ipos;
      boolean match;
      boolean overflow;
      // Find the matching data sync mark in the
      // instruction trace stream.
      ipos = InstFindMatchingDSM(data_last_dsm);

      // From the matching data sync mark in the instruction stream,
      // search backwards through the P0 elements until you find a P0
      // element with the same right-hand key value as the left-hand
      // key in the P1 element.
      (match,ipos,overflow) = InstSearchBackwards(ipos,p1key);

      if (match) then
        // Set the LME to the matched P0 element, but only if either:
        // - this is the first match
        // - the lme moves forward.
        // We never move the lme backwards.
        if (lme == UNKNOWN || ipos > lme) then
          lme = ipos;
          lme_key = p1key;

        // Attach the data item to the parent instruction
        InstStream[ipos].p1s[SizeOf(InstStream[ipos].p1s)] = dpos;

      // If we didn't have a match, set the LME to the
      // point where we finished the search and set the LME key to
      // the P1 left-hand key value.
      else if (!overflow && (lme == UNKNOWN || ipos > lme)) then
        lme = ipos;
        lme_key = p1key;
      else if (overflow) then
        lme = UNKNOWN;
        lme_key = UNKNOWN;
    else if (overflow) then
      lme = UNKNOWN;
      lme_key = UNKNOWN;

      first_p1_after_dsm = false;

    else
```

```
      integer ipos;
      boolean match;
      boolean overflow;
      // Calculate the end of the search space.
      integer search_end = (lme_key + RoundDown(TRCIDR9.NUMP0KEY/2))
                           MOD TRCIDR9.NUMP0KEY;

      // Search forwards from the LME for a matching P0 element.
      (match,ipos,overflow) = InstSearchForwards(lme,p1key,search_end);

      // If the forward search produced a match, attach the P1
      // element to the P0 element, and update the LME.
      if (!overflow && match) then
        lme = ipos;
        lme_key = p1key;
        InstStream[ipos].p1s[SizeOf(InstStream[ipos].p1s)] = dpos;
      // If the forward search did not produce a match,
      // search backwards from the LME until we find a match,
      // but do not update the LME unless we hit the start of the
      // captured trace without a match.
      else if (!overflow) then
        (match,ipos,overflow) = InstSearchBackwards(lme-1,p1key);
        if (match) then
          InstStream[ipos].p1s[SizeOf(InstStream[ipos].p1s)] = dpos;
        else if (!overflow && (lme == UNKNOWN || ipos > lme)) then
          lme = ipos;
          lme_key = p1key;
        else if (overflow) then
          lme = UNKNOWN;
          lme_key = UNKNOWN;
      else if (overflow) then
         lme = UNKNOWN;
         lme_key = UNKNOWN;else if (overflow)


//
// Iterate through the DataStream to find the next P1 element.
// Returns the left-hand key of the P1 element and the position
// of the P1 element in the DataStream.
//
(integer, integer) DataFindNextP1(integer pos)
  // Iterate through the DataStream
  while (pos < SizeOf(DataStream)) do
    // Break out if we find a P1 element
    if (isP1(DataStream[pos])) then
      // Return the P1 left-hand key of the P1 element,
      // and the position it was found.
      return (DataStream[pos].lhkey,pos);

    // Flag if we pass a Sync Mark
    if (isDSM(DataStream[pos])) then
      first_p1_after_dsm = true;
      data_last_dsm = pos;

    pos = pos + 1;

  return (UNKNOWN,pos);


//
// Iterate through the DataStream to find the next Data Sync Marker.
// Returns the position of the DSM in the DataStream.
//
(integer) DataFindDSM(integer pos)
  // Iterate through the DataStream
  while (pos < SizeOf(DataStream)) do
    // Break out if we find a DSM element
    if (isDSM(DataStream[pos])) then
```

```
      return (pos);
    pos = pos + 1;

  return (UNKNOWN);

//
// Method to search forwards in the Instruction stream to find a
// P0 element with a right-hand key which matches the supplied P1
// left-hand key.
// Returns:
// - a Boolean indicating whether a match was found
// - the position of the matching P0 element
// - an indication of whether an Overflow or Discard was detected
//
(boolean, integer, boolean) InstSearchForwards(integer start,
                                               integer p1key,
                                               integer end_key)
  integer pos = start;

  // Iterate forwards through the Instruction stream.
  while (pos < SizeOf(InstStream)) do
    // If we have a P0 element, check the right-hand key
    // to determine if we have a match or if we have reached the end
    // of the search space.
    if (IsP0(InstStream[pos]) then
      // If we have a key match, return the position.
      if (InstStream[pos].rhkey == p1key) then
        return (true,pos,false);
      // If this is the end of the search space,
      // return without success
      else if (InstStream[pos].rhkey == end_key) then
        return (false,UNKNOWN,false);

    // If this is a Cancel element, return without success.
    else if (IsCancel(InstStream[pos])) then
      return (false,UNKNOWN,false);

    // If this is an Overflow element, return without success and
    // indicate the overflow.
    else if (IsOverflow(InstStream[pos])) then
      return (false,UNKNOWN,true);

    // If this is a Discard element, return without success and
    // indicate a discard was found.
    else if (IsDiscard(InstStream[pos])) then
      return (false,UNKNOWN,true);

    pos = pos + 1;

  // If there are not enough items, return without success.
  return (false,UNKNOWN,false);

//
// Method to search backwards in the Instruction stream to find a
// P0 element with a right-hand key which matches the supplied P1
// left-hand key.
// Returns:
// - a Boolean indicating whether a match was found
// - the position of the matching P0 element
// - an indication of whether an Overflow or Discard was detected
//
(boolean, integer, boolean) InstSearchBackwards(integer start,
                                                integer p1key)
  integer pos = start;

  // Iterate backwards through the Instruction stream.
  while (pos >= 0) do
    // If we have a P0 element, check the right-hand key
```

```
  // to determine if we have a match.
  if (IsP0(InstStream[pos]) then
    // If we have a key match, return the position.
    if (InstStream[pos].rhkey == p1key) then
      return (true,pos,false);

  // If this is an Overflow element, return without success and
  // indicate the overflow.
  if (IsOverflow(InstStream[pos])) then
    return (false,pos+1,true);

  // If this is a Discard element, return without success and
  // indicate a discard was found.
  if (IsDiscard(InstStream[pos])) then
    return (false,pos+1,true);

  pos = pos - 1;

// If we run out of items, return without success.
return (false, 0, false);
```

2.4.4 Trace unit operation

The trace unit, like the trace analyzer, must follow a set of rules. These rules are described in the following sections:

- *Inserting Data Synchronization Markers in the data trace stream*.
- *Inserting Data Synchronization Markers in the instruction trace stream on page 2-51*.
- *Insertion of timestamps on page 2-51*.

Inserting Data Synchronization Markers in the data trace stream

In the data trace stream, whenever a P1 element is generated the trace unit must consider whether to insert a Data Synchronization Marker before the P1 element.

The trace unit must track the last matched P0 element, LME, and the search space around the LME which is defined as:

- All P0 elements after the LME up to and including the first P0 element with the right-hand key of:

  `end_key = (LME.rhkey + RoundDown(TRCIDR9.NUMP0KEY/2)) MOD TRCIDR9.NUMP0KEY`

- All P0 elements before the LME down to and including the most recent P0 element with the right-hand key of:

  `start_key = (end_key + 1) MOD TRCIDR9.NUMP0KEY`

When a P1 element is traced, the trace unit must determine whether the parent P0 element is within the search space, and this defines whether it is necessary to output a Data Synchronization Marker before a P1 element. A P0 element is often outside the search space if one of the following occurs:

- This is the first P1 element for a long time, and the last matching P0 element occurred a long time ago, meaning that the parent P0 element is not near the last matching P0 element and is after the end of the search space.
- This is a P1 element that has been delayed and is out of order with respect to other P1 elements, and the last matched P0 element is much more recent than the parent P0 element for this P1 element, and is before the beginning of the search space.

The trace unit does not update the LME on each P1 element. The LME is only updated if the parent P0 element is after the current LME.

A Data Synchronization Marker is inserted before a P1 element if any of the following are true:

- This is the first P1 element after tracing becomes active. This must be a Numbered Data Synchronization Marker, NDSM.

- This is the first P1 element after recovery from a data stream trace overflow. This must be an NDSM.
- A Trace Info element has been generated due to a trace synchronization request. Arm recommends that the Data Synchronization Marker is inserted before the first P1 element after the Trace Info element, and that this is an NDSM.
- If this P1 element is the first P1 element corresponding to an instruction after a previous instruction was canceled.
- The parent P0 element is not in the search space around the LME.

When an NDSM is inserted, it contains the next incremental value from the previous NDSM. The value of the first NDSM inserted after the trace unit is enabled is IMPLEMENTATION SPECIFIC.

Other elements might be present in the data stream between a Data Synchronization Marker and the corresponding P1 element, but none of these intervening elements are permitted to be a P1 element. That is, the Data Synchronization Marker must be inserted after the preceding P1 element and before the corresponding P1 element that caused the insertion.

A trace unit might also insert additional Data Synchronization Markers in the data trace stream, but the trace unit must consider the effect that these additional markers have on the LME and ensure that the trace analyzer algorithm functions correctly with these additional markers.

Inserting Data Synchronization Markers in the instruction trace stream

In the instruction trace stream, a Data Synchronization Marker is inserted if a Data Synchronization Marker is inserted into the data trace stream. The Data Synchronization Marker is of the same type in both streams, and if an NDSM, the number has the same value.

For a P1 element, the Data Synchronization Marker must be inserted after the parent P0 element that has the same P0 right-hand key, and must be inserted before the next P0 element with the same P0 right-hand key or the next Data Synchronization Marker. The Data Synchronization Marker might not be inserted immediately after the P0 element, and might be many P0 elements after the parent P0 element.

The order of the Data Synchronization Markers in the instruction trace stream must be identical to the order of the data trace stream.

The trace unit might also insert additional Data Synchronization Markers in the data trace stream, and each of these additional markers requires a Data Synchronization Marker to be inserted into the instruction trace stream.

Insertion of timestamps

NDSMs are inserted:

- After trace synchronization.
- After a trace buffer overflow.
- For the first P1 element in the trace.

Typically, each of these events requests a timestamp. As a consequence, there is at least one timestamp inserted near each NDSM. However, because timestamp insertion is permitted to be delayed, in some scenarios there might be fewer than one timestamp for each NDSM. Arm strongly recommends that a trace unit ensures that there is at least one timestamp between two identically numbered NDSMs, or it might not be possible for a trace analyzer to align the Data Synchronization Markers reliably. This applies to both the instruction and data trace streams.

2.4.5 Examples

This subsection contains a number of examples of trace streams to show different scenarios of data tracing and to demonstrate when the trace unit generates Data Synchronization Markers, and how a trace analyzer uses the information in the trace streams to match P1 elements to their parent P0 elements.

In all the examples, the number of P0 right-hand keys is six.

For simplicity, the example traces only include the trace elements that are relevant to aligning the instruction and data trace streams. For example, Address and Context elements are not shown in the instruction trace streams, although these would be required to analyze the trace successfully.

Example 1 - Basic alignment and matching

In this example:

- The instruction trace stream is a sequence of simple P0 elements. The first P0 element has a right-hand key of 0, the second P0 element has a right-hand key of 1, and this sequence continues until the seventh P0 element which restarts with a right-hand key of 0.
- The data trace stream starts when the first P1 element is generated. Three P1 elements are generated, with left-hand keys of 1,2, and 4.

Figure 2-8 shows three steps of the process as each of the three P1 elements are added. Each step shows the state of the LME and the search space at the beginning of that step.

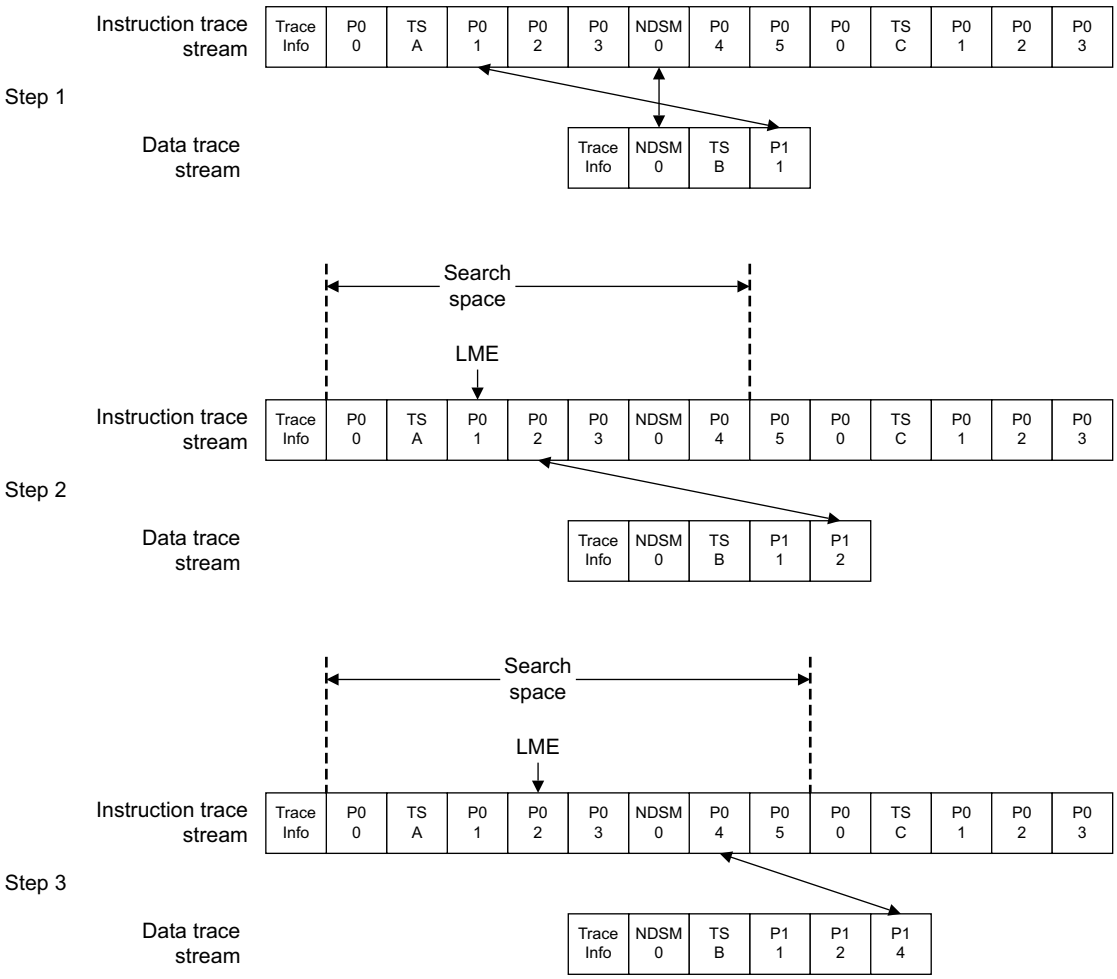


Figure 2-8 Example 1 Basic alignment and matching

Trace unit operation

This is as follows:

1. When the first P1 element must be traced:
   a. A Trace Info element is generated.

b. Because this P1 is the first P1 element, an NDSM is generated. This NDSM is generated with a value of 0.

c. A corresponding NDSM is inserted into the instruction trace stream at the same time.

d. A timestamp is also generated in the data trace stream. This has a value of B.

e. The first P1 element is then inserted into the data trace stream. This has a left-hand key of 1.

f. The LME is set to the corresponding P0 element that has a right-hand key of 1.

g. The search space is set to all P0 elements after the LME up to and including a P0 element with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

Note

This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. When the second P1 element is generated, with a left-hand key of 2:

a. The parent P0 element is within the current search space, so there is no need for a new Data Synchronization Marker.

b. The P1 element is inserted into the data trace stream and has a left-hand key of 2.

c. The LME is updated to be the corresponding P0 element with the right-hand key 2.

d. The search space is set to all P0 elements after the LME up to and including a P0 with a right-hand key of 5, and down to a P0 element with a right-hand key of 0.

3. When a third P1 element is generated, with a left-hand key of 4:

a. The parent P0 element is within the current search space, so there is no need for a new Data Synchronization Marker.

b. The P1 element is inserted into the data trace stream and has a left-hand key of 4.

c. The LME is updated to the corresponding P0 element with a right-hand key of 4.

d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 1, and down to a P0 element with a right-hand key of 2.

Trace analyzer operation

The trace analyzer must first align the Data Synchronization Markers. In the data trace stream, NDSM 0 has a timestamp value of B, provided by the Timestamp element immediately after the NDSM. There is an NDSM with a value of 0 in the instruction trace stream between timestamps A and C, where A < B < C, so this is the corresponding NDSM in the instruction trace stream.

At this point the trace analyzer begins processing P1 elements:

1. The first P1 element with a left-hand key of 1 is analyzed:

a. This is the first P1 element after an NDSM, so the trace analyzer searches backwards in the instruction trace stream from NDSM 0.

b. A P0 element with a right-hand key of 1 is found. This is the parent P0 element.

c. The LME is set to the corresponding P0 element that has a right-hand key of 1.

d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

Note

This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. The second P1 element with a left-hand key of 2 is analyzed:

a. This is not the first P1 element after a Data Synchronization Marker, so the trace analyzer searches forwards from the LME.

b. The next P0 element has a right-hand key of 2 and this is within the search space. This is the matching P0 element.

c. The LME is set to the corresponding P0 element that has a right-hand key of 2.

d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 5, and down to a P0 element with a right-hand key of 0.

3. The third P1 element with a left-hand key of 4 is analyzed:

a. This is not the first P1 element after a Data Synchronization Marker, so the trace analyzer searches forwards from the LME.

b. There is a P0 element with a right-hand key of 4 and this is within the search space. This is the matching P0 element.

c. The LME is set to the corresponding P0 element that has a right-hand key of 4.

d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 1, and down to a P0 element with a right-hand key of 2.

Example 2 - A substantial time lapse between P1 elements

This example has the same instruction trace stream as *Example 1 - Basic alignment and matching on page 2-52*. The data trace stream has three P1 elements, where the second P1 element occurs much later than the first P1 element, and this requires the insertion of a Data Synchronization Marker.

Figure 2-9 on page 2-55 shows the three steps as each of the three P1 elements is added to the process. Each step shows the state of the LME and the search space at the beginning of that step.

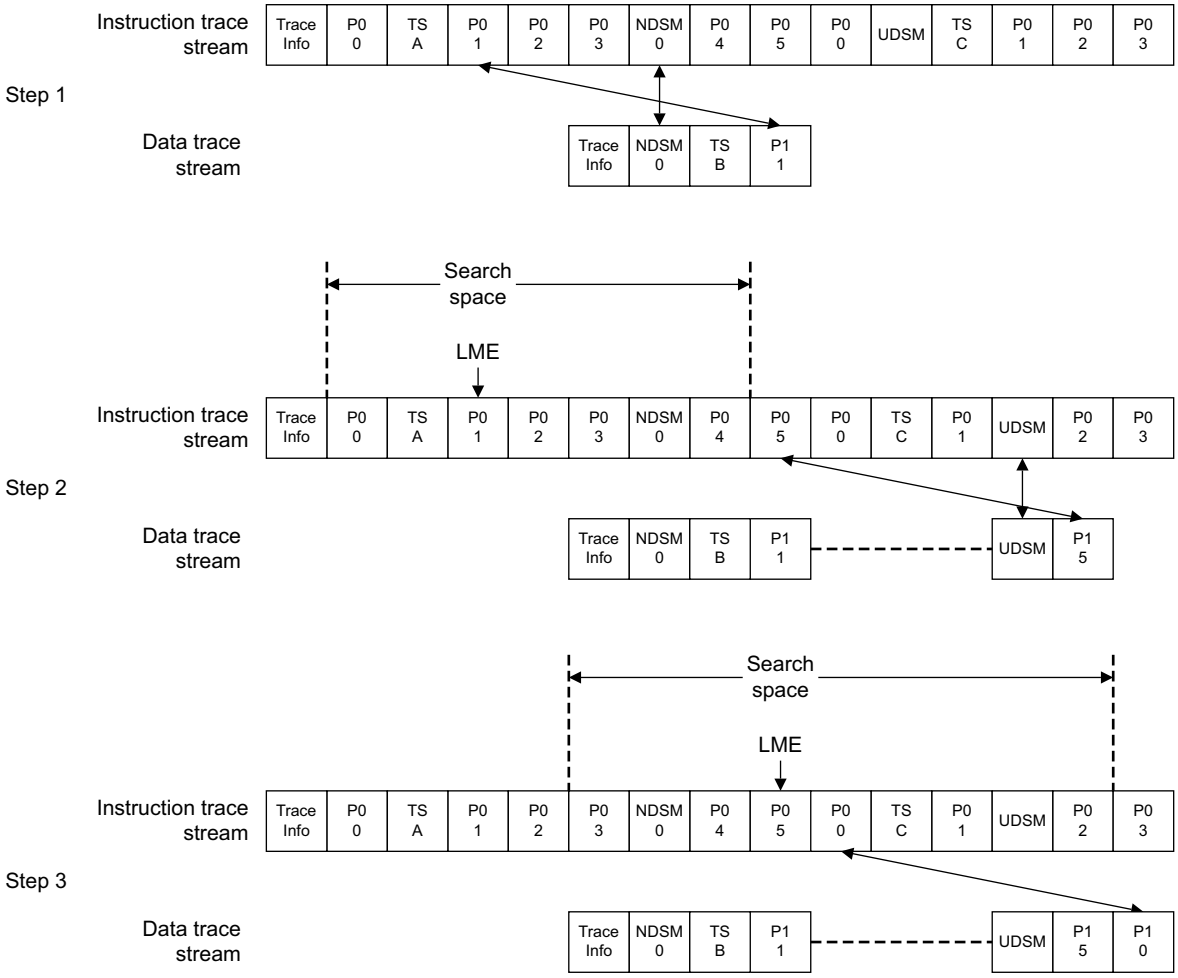


Figure 2-9 Example 2 A substantial time lapse between P1 elements

Trace unit operation

This is as follows:

1. When the first P1 element must be traced:
   a. A Trace Info element is generated.
   b. Because this P1 element is the first P1 element, an NDSM is generated. This NDSM has a value of 0.
   c. A corresponding NDSM is inserted into the instruction trace stream at the same time.
   d. A timestamp is also generated in the data trace stream. This has a value of B.
   e. The first P1 element is then inserted into the data trace stream and has a left-hand key of 1.
   f. The LME is set to the corresponding P0 element that has a right-hand key of 1.
   g. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

   Note

   This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. When the second P1 element is generated, with a left-hand key of 5:
    a. The parent P0 element is not within the current search space, so a Data Synchronization Marker must be inserted into both the data and instruction trace streams. This is an Un-numbered Data Synchronization Marker, UDSM.
    b. The P1 element is inserted into the data trace stream. This has a left-hand key of 5.
    c. The LME is updated to be the corresponding P0 element with a right-hand key of 5.
    d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 2, and down to a P0 element with a right-hand key of 3.
3. When a third P1 element is generated, with a left-hand key of 0:
    a. The parent P0 element is within the current search space, so a new Data Synchronization Marker is not required.
    b. The P1 element is inserted into the data trace stream. It has a left-hand key of 0.
    c. The LME is updated to be the corresponding P0 element with a right-hand key of 0.
    d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 3, and down to a P0 element with a right-hand key of 4.

Trace analyzer operation

The trace analyzer must first align the Data Synchronization Markers. In the data trace stream, NDSM 0 has timestamp value of B, provided by the Timestamp element immediately after the NDSM. There is an NDSM with a value of 0 in the instruction trace stream between timestamps A and C, where A < B < C, so this is the corresponding NDSM in the instruction trace stream.

Now the trace analyzer begins processing the P1 elements:

1. The first P1 element with a left-hand key of 1 is analyzed:
    a. This is the first P1 element after an NDSM, so the trace analyzer searches backwards in the instruction trace stream from NDSM 0.
    b. A P0 element with a right-hand key of 1 is found. This is the parent P0 element.
    c. The LME is set to the corresponding P0 element that has a right-hand key of 1.
    d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

    Note

    This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.
2. The second P1 element with a left-hand key of 5 is analyzed:
    a. There was a UDSM before this P1 element so the trace analyzer must search backwards from the corresponding UDSM in the instruction trace stream.
    b. There is a P0 element with a right-hand key of 5 before the UDSM. This is the matching P0 element.
    c. The LME is set to the corresponding P0 element that has a right-hand key of 5.
    d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 2, and down to a P0 element with a right-hand key of 3.
3. The third P1 element with a left-hand key of 0 is analyzed:
    a. This is not the first P1 element after a Data Synchronization Marker, so the trace analyzer searches forwards from the LME.
    b. A P0 element with a right-hand key of 0 is found. This is the parent P0 element.
    c. The LME is set to the corresponding P0 element that has a right-hand key of 0.
    d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 3, and down to a P0 element with a right-hand key of 2.

Example 3 - Simple out-of-order P1 elements

This example has the same instruction trace stream as *Example 1 - Basic alignment and matching on page 2-52*. The data trace stream has different P1 elements that are out-of-order with respect to the parent P0 elements. The P1 elements are generated with a left-hand key of 1, 0, and 5.

Figure 2-10 shows three steps as each of the three P1 elements is added to the process. Each step shows the state of the LME and the search space at the beginning of that step.

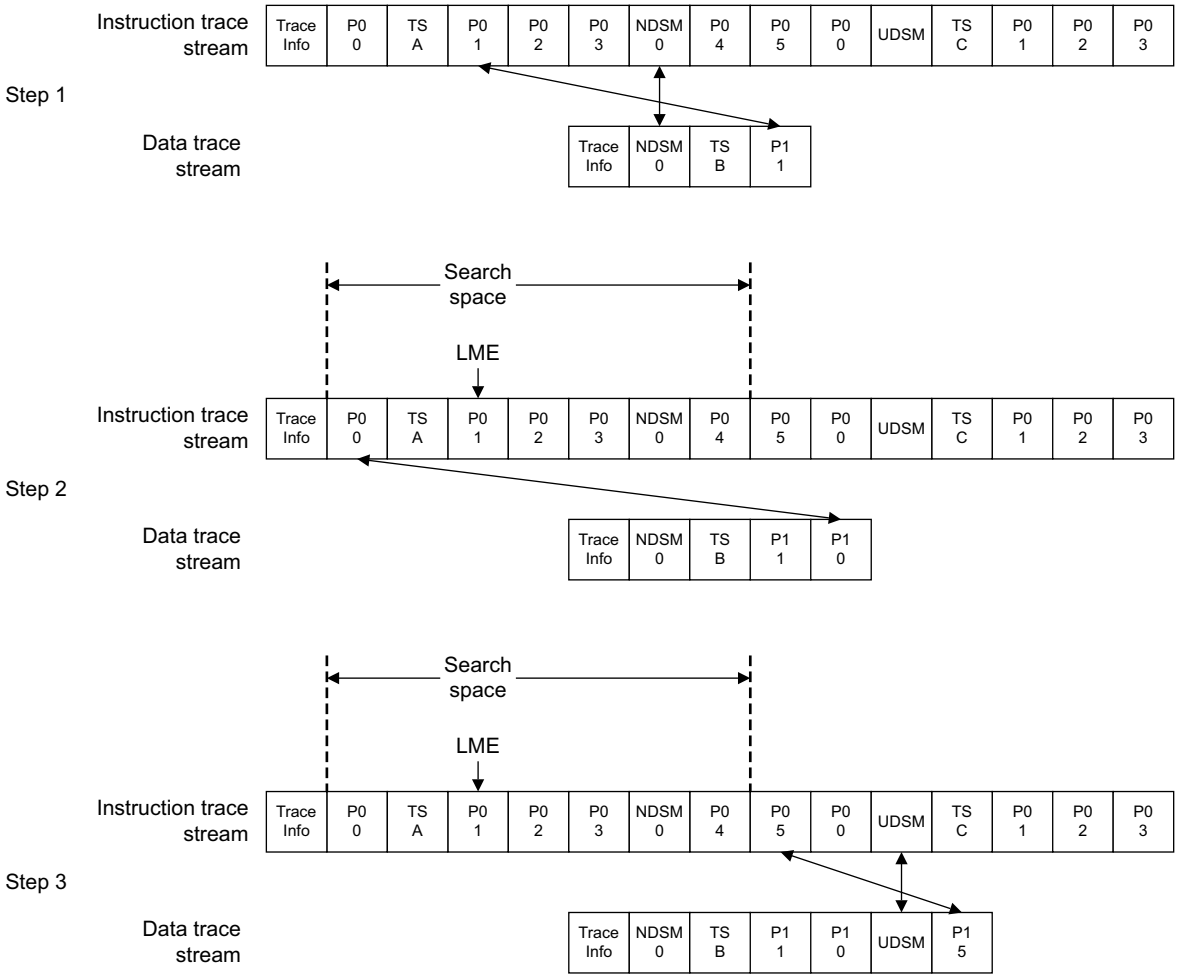


Figure 2-10 Example 3 Simple out-of-order P1 elements

Trace unit operation

This is as follows:

1. When the first P1 element must be traced:
   a. A Trace Info element is generated.
   b. Because this P1 element is the first P1 element, an NDSM is generated. This NDSM is generated with a value of 0.
   c. A corresponding NDSM is inserted into the instruction trace stream at the same time.
   d. A timestamp is also generated in the data trace stream and this has value of B.
   e. The first P1 element is then inserted into the data trace stream and has a left-hand key of 1.
   f. The LME is set to the corresponding P0 element that has a right-hand key of 1.

g. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

Note

This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. When the second P1 element is generated, with a left-hand key of 0:
   a. The parent P0 element is within the current search space, so a new Data Synchronization Marker is not required.
   b. The P1 element is inserted into the data trace stream and has a left-hand key of 0.
   c. The LME and the search space are not updated because the matching P0 element occurred before the LME.
3. When the third P1 element is generated, with a left-hand key of 5:
   a. The parent P0 element is not within the current search space, so a Data Synchronization Marker must be inserted into both the data and instruction trace streams. This is an Un-numbered Data Synchronization Marker, UDSM.
   b. The P1 element is inserted into the data trace stream and has a left-hand key of 5.
   c. The LME is updated to be the corresponding P0 element with a right-hand key of 5.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 2, and down to a P0 element with a right-hand key of 3.

Trace analyzer operation

The trace analyzer must first align the Data Synchronization Markers. In the data trace stream, NDSM 0 has a timestamp value of B, provided by the Timestamp element immediately after the NDSM. There is an NDSM with a value of 0 in the instruction trace stream between timestamps A and C, where $A < B < C$, so this is the corresponding NDSM in the instruction trace stream.

Now the trace analyzer begins processing the P1 elements:

1. The first P1 element with a left-hand key of 1 is analyzed:
   a. This is the first P1 element after an NDSM, so the trace analyzer searches backwards in the instruction trace stream from NDSM to 0.
   b. A P0 element with a right-hand key of 1 is found. This is the parent P0 element.
   c. The LME is set to the corresponding P0 element that has a right-hand key of 1.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

Note

This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. The second P1 element with a left-hand key of 0 is analyzed:
   a. This is not the first P1 element after a Data Synchronization Marker, so the trace analyzer searches forwards from the LME.
   b. There is no P0 element with a right-hand key of 0 forward in the search space, so the trace analyzer must search backwards from the LME.
   c. A P0 element with a right-hand key of 0 is found. This is the parent P0 element.
   d. The LME and search space are not updated because the parent P0 element occurred before the LME.

3. The third P1 element with a left-hand key of 5 is analyzed:
    a. There was a UDSM before this P1 element so the trace analyzer must search backwards from the corresponding UDSM in the instruction trace stream.
    b. There is a P0 element with a right-hand key of 5 before the UDSM. This is the matching P0 element.
    c. The LME is set to the corresponding P0 element that has a right-hand key of 5.
    d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 2, and down to a P0 element with a right-hand key of 3.

Example 4 - Misspeculation 1

This example addresses a situation where two P0 elements are canceled because of misspeculation. Two P0 elements with right-hand keys of 3 and 4 are canceled, generating a Cancel element. Two new P0 elements with right-hand keys of 3 and 4 are generated later for the correct path of execution.

Three P1 elements are generated in this example:

- The first P1 element is generated for a corresponding P0 element that was generated before the Cancel element.
- The second P1 element is generated after the Cancel element, but corresponds to a P0 element that was generated before the Cancel element.
- The third P1 element is generated for a P0 element that was generated after the Cancel element.

Figure 2-11 on page 2-60 shows three steps as each of the three P1 elements are added to the process. Each step shows the state of the LME and the search space at the beginning of that step.

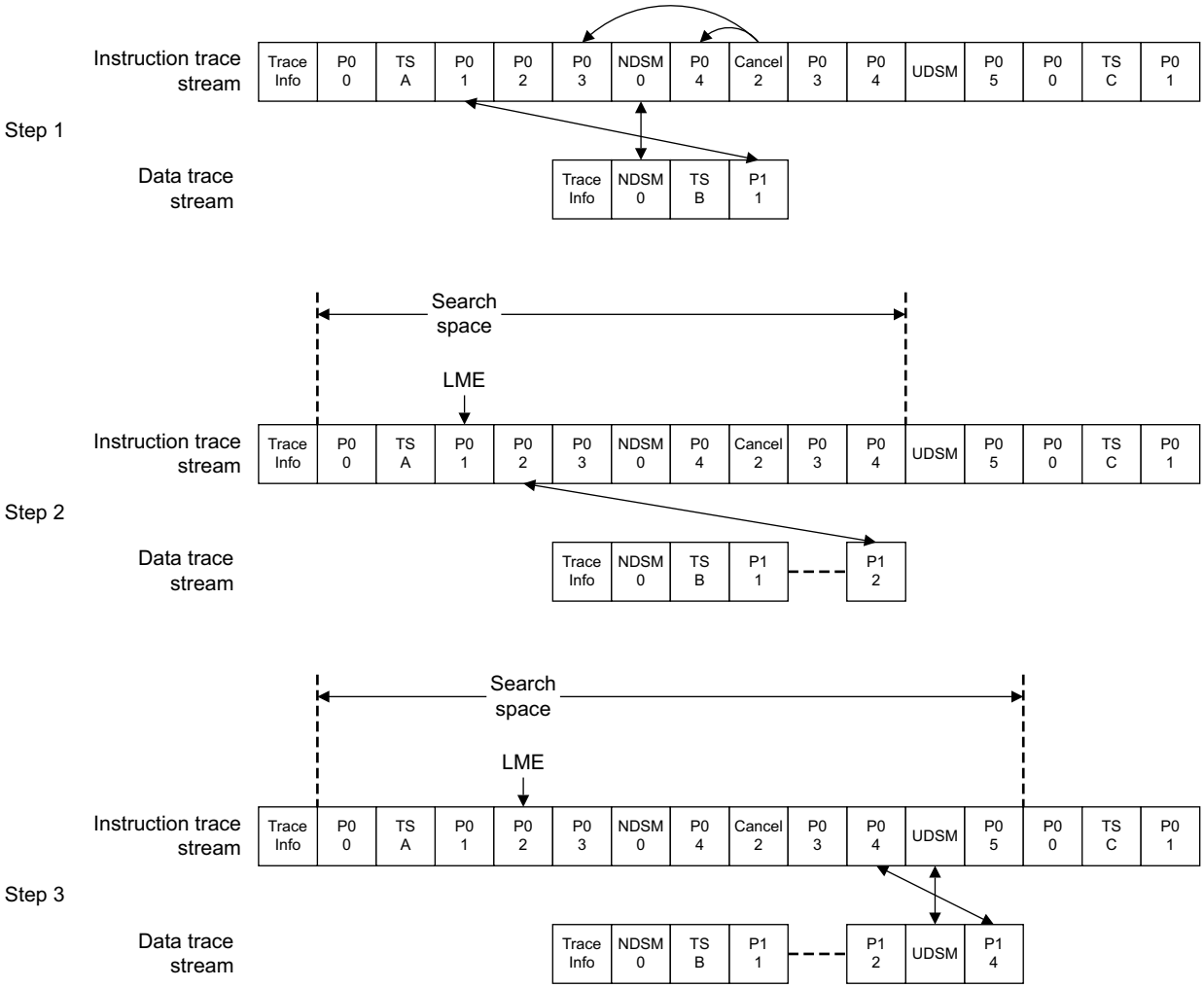


Figure 2-11 Example 4 - Misspeculation 1

Trace unit operation

Trace unit operation is as follows:

1. When the first P1 element must be traced:
   a. A Trace Info element is generated.
   b. Because this P1 element is the first P1 element, an NDSM is generated. This NDSM is generated with a value of 0.
   c. A corresponding NDSM is inserted into the instruction trace stream at the same time.
   d. A timestamp with a value of B is also generated in the data trace stream.
   e. The first P1 element is then inserted into the data trace stream and has a left-hand key of 1.
   f. The LME is set to the corresponding P0 element that has a right-hand key of 1.
   g. The search space is set to be all P0 elements after the LME up to, and including, a P0 element with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

   Note

   This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. When the second P1 element is generated, with a left-hand key of 2:
   a. The parent P0 element is within the current search space, so a new Data Synchronization Marker is not required.
   b. The P1 element is inserted into the data trace stream and has a left-hand key of 2.
   c. The LME is updated to be the corresponding P0 element with a right-hand key of 2.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 5, down to a P0 element with a right-hand key of 0.
3. When a third P1 element is generated, with a left-hand key of 4:
   a. This is the first P1 element for an instruction after the Cancel element, so a Data Synchronization Marker must be inserted into both the data and instruction trace streams. This is a UDSM.
   b. The P1 element is inserted into the data trace stream and has a left-hand key of 4.
   c. The LME is updated to be the corresponding P0 element with a right-hand key of 4.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 1, and down to a P0 element with a right-hand key of 2.

Trace analyzer operation

The trace analyzer must first align the Data Synchronization Markers. In the data trace stream, NDSM 0 has a timestamp value of B, provided by the Timestamp element immediately after the NDSM. There is an NDSM with a value of 0 in the instruction trace stream between timestamps A and C, where $A < B < C$, so this is the corresponding NDSM in the instruction trace stream.

Now the trace analyzer begins processing the P1 elements:

1. The first P1 element with a left-hand key of 1 is analyzed:
   a. This is the first P1 element after an NDSM, so the trace analyzer searches backwards in the instruction trace stream from NDSM to 0.
   b. A P0 element with a right-hand key of 1 is found. This is the parent P0 element.
   c. The LME is set to the corresponding P0 element that has a right-hand key of 1.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

      Note

      This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.
2. The second P1 element with a left-hand key of 2 is analyzed:
   a. This is not the first P1 element after a Data Synchronization Marker, so the trace analyzer searches forwards from the LME.
   b. A P0 element with a right-hand key of 2 is found. This is the parent P0 element.
   c. The LME is set to the corresponding P0 element that has a right-hand key of 2.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 5, and down to a P0 element with a right-hand key of 0.
3. The third P1 element with a left-hand key of 4 is analyzed:
   a. There was a UDSM before this P1 element so the trace analyzer must search backwards from the corresponding UDSM in the instruction trace stream.
   b. There is a P0 element with a right-hand key of 4 before the UDSM. This is the matching P0 element.

      Note

      This is not related to the canceled P0 element with a right-hand key of 4.
   c. The LME is set to the corresponding P0 element that has a right-hand key of 4.

d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 1, and down to a P0 element with a right-hand key of 2.

Example 5- Misspeculation 2

Like *Example 4 - Misspeculation 1 on page 2-60*, this example addresses a situation where two P0 elements are canceled because of misspeculation. Two P0 elements with right-hand keys of 3 and 4 are canceled, generating a Cancel element. Two new P0 elements with right-hand keys of 3 and 4 are generated later for the correct path of execution.

Three P1 elements are generated in this example:

- The first P1 element is generated for a corresponding P0 element that was generated before the Cancel element.
- The second P1 element corresponds to a P0 element that was generated before the Cancel element.
- The third P1 element has the same left-hand key as the P1 element that preceded it but corresponds to a P0 element after the Cancel element.

Figure 2-12 shows three steps as each of the three P1 elements are added to the process. Each step shows the state of the LME and the search space at the beginning of that step.

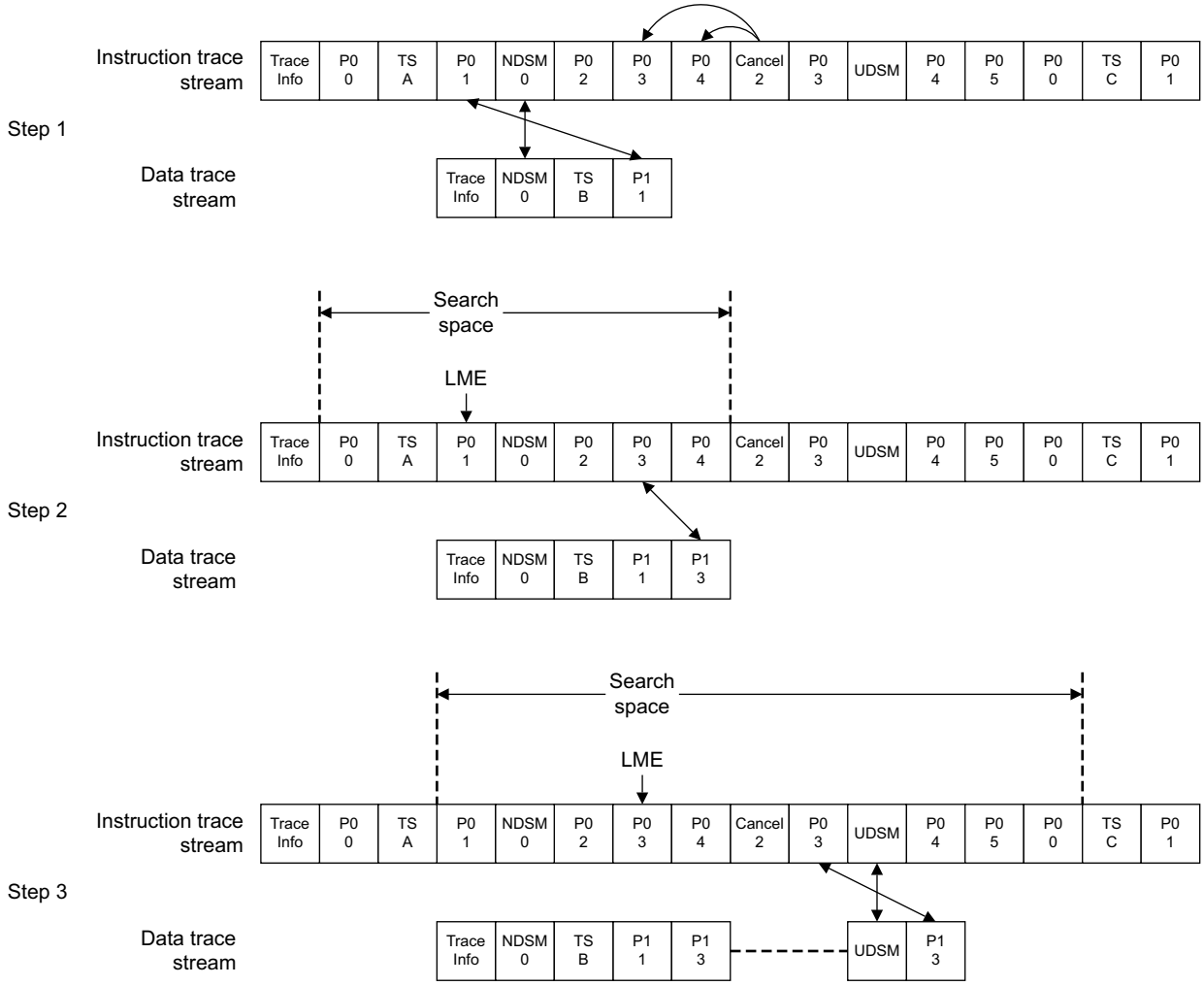


Figure 2-12 Example 5 - Misspeculation 2

Trace unit operation

This is as follows:

1. When the first P1 element must be traced:
   a. A Trace Info element is generated.
   b. Because this is the first P1 element, an NDSM is generated. This NDSM is generated with a value of 0.
   c. A corresponding NDSM is inserted into the instruction trace stream at the same time.
   d. A timestamp is also generated in the data trace stream, and this has a value of B.
   e. The first P1 element is then inserted into the data trace stream and has a left-hand key of 1.
   f. The LME is set to the corresponding P0 element that has a right-hand key of 1.
   g. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

      Note

      This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. When the second P1 element is generated, with a left-hand key of 3:
   a. The parent P0 element is within the current search space, so a new Data Synchronization Marker is not required.
   b. The P1 element is inserted into the data trace stream and has a left-hand key of 3.
   c. The LME is updated to be the corresponding P0 element with a right-hand key of 3.

      Note

      This is the first P0 element with a right-hand key of 3.

   d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 0, and down to a P0 element with a right-hand key of 1.

      Note

      This includes multiple P0 elements with the same right-hand key value.

3. When the third P1 element is generated, with a left-hand key of 3:
   a. This is the first P1 element for an instruction after the Cancel element, so a Data Synchronization Marker must be inserted into both the data and instruction trace streams. This is a UDSM.
   b. The P1 element is inserted into the data trace stream and has a left-hand key of 3.
   c. The LME is updated to be the corresponding P0 element with a right-hand key of 3.

      Note

      This is the second P0 element with a right-hand key of 3.

   d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 0, and down to a P0 element with a right-hand key of 1. The search space is unchanged, although the LME has changed.

Trace analyzer operation

The trace analyzer must first align the Data Synchronization Markers. In the data trace stream, NDSM 0 has a timestamp value of B, provided by the Timestamp element immediately after the NDSM. There is an NDSM with a value of 0 in the instruction trace stream between timestamps A and C, where $A < B < C$, so this is the corresponding NDSM in the instruction trace stream.

Now the trace analyzer begins processing the P1 elements:

1. The first P1 element with a left-hand key of 1 is analyzed:
   a. This is the first P1 element after an NDSM, so the trace analyzer searches backwards in the instruction trace stream from NDSM 0.
   b. A P0 element with a right-hand key of 1 is found. This is the parent P0 element.
   c. The LME is set to the corresponding P0 element that has a right-hand key of 1.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 4, and down to a P0 element with a right-hand key of 5.

      Note

      This example only goes backwards to a P0 element with a right-hand key of 0, because there is no earlier instruction trace.

2. The second P1 element with a left-hand key of 3 is analyzed:
   a. This is not the first P1 element after a Data Synchronization Marker, so the trace analyzer searches forwards from the LME.
   b. A P0 element with a right-hand key of 3 is found. This is the parent P0 element.

      Note

      - This is the first P0 element with a right-hand key value of 3.
      - This instruction is canceled.

   c. The LME is set to the first P0 element that has a right-hand key of 3.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 with a right-hand key of 0, and down to a P0 element with a right-hand key of 1.
3. The third P1 element with a left-hand key of 3 is analyzed:
   a. There was a UDSM before this P1 element so the trace analyzer must search backwards from the corresponding UDSM in the instruction trace stream.
   b. There is a P0 element with a right-hand key of 3 before the UDSM. This is the matching P0 element.

      Note

      This is the second P0 element with a right-hand key of 3.

   c. The LME is set to the second P0 element that has a right-hand key of 3.
   d. The search space is set to be all P0 elements after the LME up to and including a P0 element with a right-hand key of 0, and down to a P0 element with a right-hand key of 1. The search space is unchanged.

2.5 Synchronization with a trace analyzer

An ETMv4 trace unit can output two trace streams that can be handled independently. For example, one trace stream might be exported off-chip using a debug port, and the other might be stored on-chip for low-speed export later on. If trace is stored on-chip, it 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 it has been stored in circular buffer, a trace unit must periodically generate trace synchronization points in each trace stream. The trace unit generates these trace synchronization points whenever a trace synchronization request occurs.

Trace synchronization requests might come from:

- The trace unit itself. The trace unit can be programmed to generate trace synchronization requests on a periodic basis. The number of bytes of trace that are output between trace synchronization requests can be specified by programming TRCSYNCPR.PERIOD.
- Outside the trace unit, for example, from a trace analyzer or from another on-chip component.

In addition, trace synchronization requests automatically occur:

- When the trace unit is first enabled.
- Whenever a trace unit buffer overflow occurs.

On receiving a trace synchronization request, the trace unit generates a trace synchronization point in the instruction trace stream and, if data tracing is implemented and enabled, also in the data trace stream. However, these trace synchronization points might not occur at the same point in each stream. For example, if there is a risk that one of the trace unit buffers might overflow, the trace unit might wait for a short time before generating a trace synchronization point in that stream.

This also means that, if there is a risk of an overflow of both trace buffers, the trace unit might not generate trace synchronization points in either stream until that risk has passed for at least one of the trace buffers.

The remainder of this section is organized as follows:

- *Synchronizing with the instruction trace stream*.
- *Synchronizing with the data trace stream on page 2-68*.

2.5.1 Synchronizing with the instruction trace stream

Trace synchronization points in the instruction trace stream are identified by an A-Sync packet.

Whenever a trace synchronization request occurs, the trace unit generates the following packets in the instruction trace stream:

1. An A-Sync packet. This enables a trace analyzer to determine where another packet starts. For more information, see *Alignment Synchronization (A-Sync) instruction trace packet on page 6-253*.
2. A Trace Info packet. This is generated after the A-Sync packet, and serves two purposes:
   - Provides a trace analyzer with information about the setup of the trace, such as whether load or store instructions are traced explicitly, whether cycle counting is enabled, and the right-hand key value for the next P0 element.
   - Provides a point in the trace stream where analysis of the trace stream can begin.

   For more information, see *Trace Info instruction trace packet on page 6-254* and *Trace Info instruction trace element on page 5-188*.

Other packets can occur between the A-Sync and Trace Info packets. However, Arm recommends that the Trace Info packet appears in the trace stream soon after the A-Sync packet.

After generating the A-Sync and Trace Info packets, the trace unit must generate:

- An Address packet, to provide a trace analyzer with an address from where analysis of program execution can begin. For more information, see *Address and Context tracing packets on page 6-285* and *Address instruction trace element on page 5-209*.

- A Context packet, to provide a trace analyzer with information about the context in which instructions are being executed. For more information, see *Address and Context tracing packets on page 6-285* and *Context instruction trace element on page 5-211*.

If the tracing of conditional non-branch instructions is implemented and enabled, then a Conditional Flush element is also required after the Trace Info packet. Arm recommends that this is generated immediately before or after the Address packet.

Arm recommends that the Address and Context packets are generated as soon as possible after the A-Sync and Trace Info packets.

From ETMv4.5 the instruction trace stream might include a Resynchronization element, which indicates that the trace analyzer might not have enough information to properly analyze the trace. When a trace analyzer receives a Resynchronization element, in some scenarios it must not infer program execution until after a subsequent Address element has been received. For more information, see *Resynchronization element on page 5-213*.

If global timestamping is enabled, the trace unit must also generate a Timestamp packet soon after the Trace Info packet. The timestamp value that is contained in the Timestamp packet corresponds to whichever one of a particular group of elements was most recently generated. See *Timestamp instruction trace element on page 5-215* for a list of these elements.

The most recent of these might have occurred either:

- Before the A-Sync packet.
- Between the A-Sync packet and the Trace Info packet.
- After the A-Sync packet and Trace Info packets.

For more information, see:

- *Global timestamping on page 2-82*.
- *Timestamp instruction trace element on page 5-215*.
- *Timestamp instruction trace packet on page 6-259*.

Interpreting the information that is contained in the Address and Context packets

When a trace synchronization request occurs, the rules for interpreting the information that is contained in the Address and Context instruction trace packets vary depending on whether ViewInst is active or inactive when the Trace Info element is generated.

If the Trace Info element is generated while ViewInst is active, and as a result of a trace synchronization request, then the address and context information is for the target of the most recent P0 element as shown in Figure 2-13 and Figure 2-14 on page 2-67.

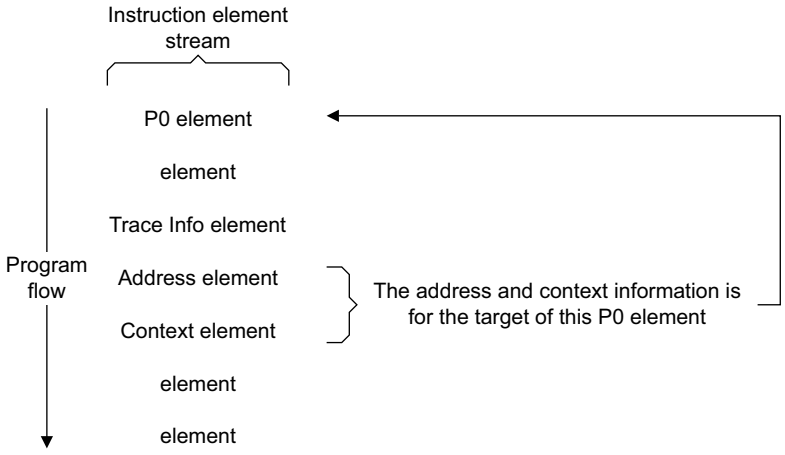


Figure 2-13 Interpreting the address and context information when ViewInst is active, example one

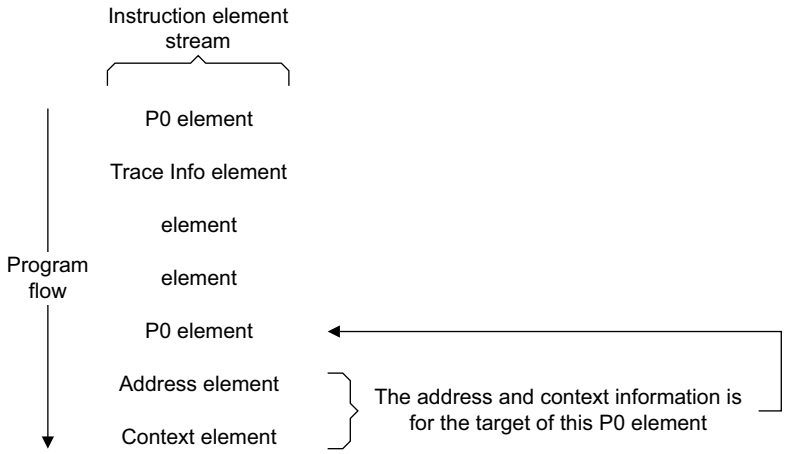


Figure 2-14 Interpreting the address and context information when ViewInst is active, example two

Sometimes, the trace unit might generate a Trace Info element when ViewInst is inactive. For example, if ViewInst is only active for a particular program function or section of code, a trace synchronization request might occur at a time when ViewInst is inactive. In this case:

1. The trace unit generates a Trace Info element while ViewInst is inactive.
2. When ViewInst becomes active again, the trace unit generates a Trace On element to indicate a gap in the trace stream.

   Note

   This is not a special case, that is, a Trace On element is normally generated after a gap in the trace stream. See *Trace On instruction trace element on page 5-190*.

3. An Address and Context element must be generated before the next P0 element is generated, so that the trace analyzer knows where to restart analysis of program execution. This is shown in Figure 2-15.

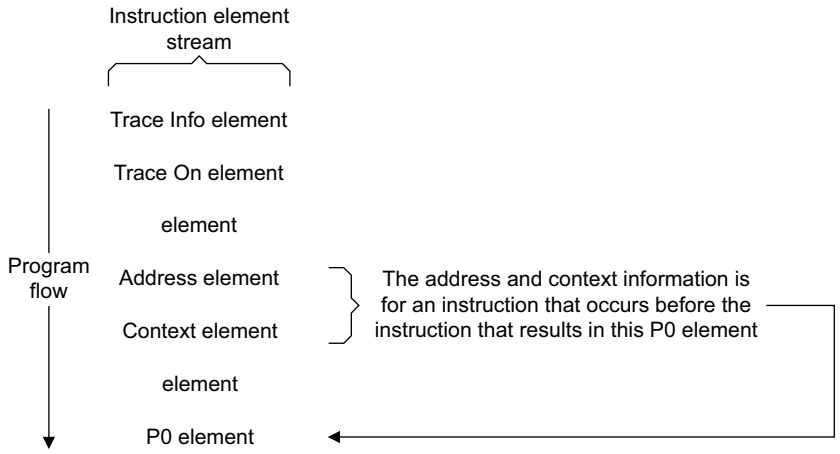


Figure 2-15 Interpreting the address and context information when ViewInst is inactive

Note

- Arm recommends that if a trace synchronization request occurs while ViewInst is inactive, the A-Sync packet is not output in the instruction trace stream until just before either:
  - ViewInst becomes active.
  - An Event tracing instruction trace packet is output.

This is because if the instruction trace stream is stored in a circular buffer, then if the buffer fills between the time when the A-Sync packet is output, and the time when either ViewInst becomes active or an Event tracing instruction trace packet is output, then the A-Sync packet might be overwritten.

- Arm recommends that an A-Sync packet is only output in the instruction trace stream if other trace packets have been output since the previous A-Sync packet. This strategy reduces the risk of a circular buffer filling and overwriting trace.
- If two or more synchronization requests occur, and no trace is generated between these two requests, then Arm recommends that full trace synchronization occurs before any further trace is generated. Full trace synchronization involves generating both the A-Sync and Trace Info packets. 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.

2.5.2 Synchronizing with the data trace stream

Like trace synchronization points in the instruction trace stream, trace synchronization points in the data trace stream are identified by an A-Sync packet.

Whenever a trace synchronization request occurs, the trace unit generates the following packets in the data trace stream:

1. An A-Sync packet. This enables a trace analyzer to determine where another packet starts. For more information, see *Alignment Synchronization (A-Sync) data trace packet on page 6-309*.
2. A Trace Info packet. This is generated after the A-Sync packet. It provides a point in the trace stream where analysis of the trace stream can begin.

   For more information, see: *Trace Info data trace packet on page 6-309* and *Trace Info data trace element on page 5-226*.

Note

Unlike Trace Info packets in the instruction trace stream, Trace Info packets in the data trace stream do not provide any information about the setup of the trace.

Other packets might occur between the A-Sync and Trace Info packets. However, Arm recommends that the Trace Info packet appears in the trace stream soon after the A-Sync packet.

If global timestamping is enabled, the trace unit must also generate a Timestamp packet soon after the Trace Info packet.

Following the Trace Info packet, a Numbered Data Synchronization Marker (NDSM) must be inserted into the trace streams. See *Trace unit operation on page 2-50* for more details on when an NDSM must be inserted into the trace streams.

For more information, see:

- *Global timestamping on page 2-82*.
- *Timestamp data trace packet on page 6-314*.
- *Timestamp data trace element on page 5-230*.

Note

- Arm recommends that, if a trace synchronization request occurs while ViewData is inactive, the A-Sync packet is not output in the data trace stream until just before either:
  - ViewData becomes active.
  - An Event tracing data trace packet is output.

  This is because otherwise, if the data trace stream is stored in a circular buffer, then if the buffer fills between the time when the A-Sync packet is output, and the time when either ViewData becomes active or an Event tracing data trace packet is output, then the A-Sync packet might be overwritten.

- Arm recommends that an A-Sync packet is only output in the data trace stream if other trace packets have been output since the previous A-Sync packet. This strategy reduces the risk of a circular buffer filling and overwriting trace.
- If two or more synchronization requests occur, and no trace is generated between these two requests, then Arm recommends that full trace synchronization occurs before any further trace is generated. Full trace synchronization involves generating both the A-Sync and Trace Info packets. 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.

2.6 Trace behavior

The following sections describe trace behavior:

- *Trace behavior on speculative execution*.
- *Trace behavior on tracing conditional instructions on page 2-71*.
- *Trace behavior on tracing Jazelle execution on page 2-77*.
- *Trace behavior on tracing ThumbEE instructions on page 2-77*.
- *Data trace behavior on tracing store-exclusive instructions on page 2-77*.

2.6.1 Trace behavior on speculative execution

The ETMv4 architecture supports the tracing of speculative execution of instructions by a PE.

An ETMv4 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 stream. Speculative data transfers might also be traced in the data trace stream, if the trace unit is programmed to generate them.

This means that some of the program execution information that is shown in the trace streams 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.

The trace unit resolves this issue by generating elements to confirm the status of each instruction in the instruction trace stream. That is, it 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 it receives an element that indicates whether the instruction has been committed for execution, or canceled because it was mis-speculated.

Therefore, whenever instructions are traced, later on in the instruction trace stream elements appear that show whether those instructions have been executed or canceled. A trace analyzer must then take the appropriate action, that might involve canceling some trace elements, to establish what the actual program execution is.

Elements that resolve the status of a traced instruction are called *speculation resolution elements*. These elements are:

- The Cancel element. This indicates that one or more P0 elements are canceled. If any load or store instructions are represented by the canceled P0 elements, then all data transfers associated with those load or store instructions are also canceled. A Cancel element must never cancel more P0 elements than are currently speculative.
- The Commit element. This indicates that one or more P0 elements are committed for execution. A Commit element must never commit more P0 elements than are currently speculative.
- The Mispredict element. This indicates that the most recent *Atom element* has the incorrect E or N status. This means that the predicted outcome of a traced conditional branch instruction is incorrect.

The maximum speculation depth, that is, the maximum permitted number of P0 elements that can be speculative at any instance is IMPLEMENTATION DEFINED. TRCIDR8.MAXSPEC shows the maximum speculation depth. The trace unit must never output more speculative P0 elements than the maximum speculation depth.

If an implementation is not exposed to any speculative execution, then Arm recommends that the implementation has 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 committed 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.

If trace is generated for speculative execution, then the trace for any mis-speculated execution must not contain information that cannot be accessed by software executing at the current or a lower level of privilege that the mis-speculated execution was executed at. For instruction trace, pages with executable permissions are considered accessible, even if those pages do not have read or write permissions.

If a Context synchronization event is speculated as being taken or executed, trace must not be generated for any execution after the Context synchronization event until the Context synchronization event is resolved as being taken or executed. If the Context synchronization event is resolved as not taken or not executed, no trace is generated for any mis-speculated execution after the Context synchronization event.

If an exit from a prohibited region is speculated as being taken, trace must not be generated for any execution after the prohibited region exit until the prohibited region exit is resolved as being taken. If the prohibited region exit is resolved as not taken, no trace is generated for any mis-speculated execution after the prohibited region exit.

2.6.2 Trace behavior on tracing conditional instructions

All conditional branch instructions are traced using *Atom elements*, that have an E or N status.

If tracing of conditional non-branch instructions is implemented and enabled, all conditional non-branch instructions are traced using Conditional Instruction (C) elements, Conditional Result (R) elements, and Conditional Flush (F) elements.

This section is split into subsections, as follows:

- *Conditional branch instructions*.
- *Conditional non-branch instructions*.
- *About the generation of Conditional Instruction (C) elements on page 2-73*:
  - *The algorithm for tracing the APSR condition flag values on page 2-73*.
  - *The algorithm for tracing the pass or fail result on page 2-74*.
- *About the ordering of C, R, and F elements in relation to other elements on page 2-75*.
- *About analyzing C, R, and F elements on page 2-75*.

Conditional branch instructions

Note

Conditional branch instructions are always traced.

Conditional branches are traced using *Atom elements*. *Atom elements* are a subset of P0 elements, as shown in Figure 2-3 on page 2-34.

An *Atom element* contains either an E or an N status:

- If a conditional branch is taken, it is traced using an E Atom.
- If a conditional branch is not taken, it is traced using an N Atom.

If the branch is taken, execution continues to the target of that branch.

Whether the branch is taken might be a prediction. If it is a prediction, and it is later discovered to be incorrect, then the E or N status of the most recent *Atom element* that is generated can be corrected by a Mispredict element. If it is necessary to correct the prediction of an earlier *Atom element*, the more recent *Atom element* must first be canceled. See *Atom instruction trace element on page 5-192*.

Appendix F *Instruction Categories* shows which instructions are classified as branch instructions.

Conditional non-branch instructions

Note

- Conditional non-branch instructions are only traced if tracing of conditional non-branch instructions is implemented and enabled. See *Conditional instructions tracing on page 2-84*.
- Support for tracing conditional non-branch instructions is required if data tracing is implemented.

- ETMv4 does not support the tracing of conditional non-branch instructions on Armv7-A and Armv8-A PEs.

TRCIDR0.TRCCOND indicates if the tracing of conditional non-branch instructions is implemented. Conditional non-branch instructions are traced using the following element types:

Conditional Instruction (C) elements

These are generated when a conditional non-branch instruction is executed.

Conditional Result (R) elements

These are generated when the result of a conditional non-branch instruction is known.

Conditional Flush (F) elements

These are used to manage mis-speculation of conditional non-branch instructions and are also required at trace synchronization points.

On receiving a C element, a trace analyzer knows that a conditional instruction has been executed. However, the trace analyzer does not know the result of that conditional instruction until it receives an R element to associate with the C element. An R element contains one of the following:

- The values of those APSR condition flags that are required to compute whether the instruction passed or failed its condition check code.
- An indication of whether the instruction passed or failed its condition code check.

Note

Whether an R element contains an indication of the pass or fail result, or a copy of the required APSR condition flag values, is IMPLEMENTATION DEFINED. TRCIDR0.CONDTYPE shows which method is used.

A C element has a right-hand key that associates a subsequent R element to the C element. A C element can only be associated with one R element. However, an R element might be associated with more than one C element. This is because, for example, two different C elements might each require the status of a different APSR condition flag, but a single R element might contain values for the condition flags required by both C elements.

Figure 2-16 shows this for the case of three C elements and one R element.

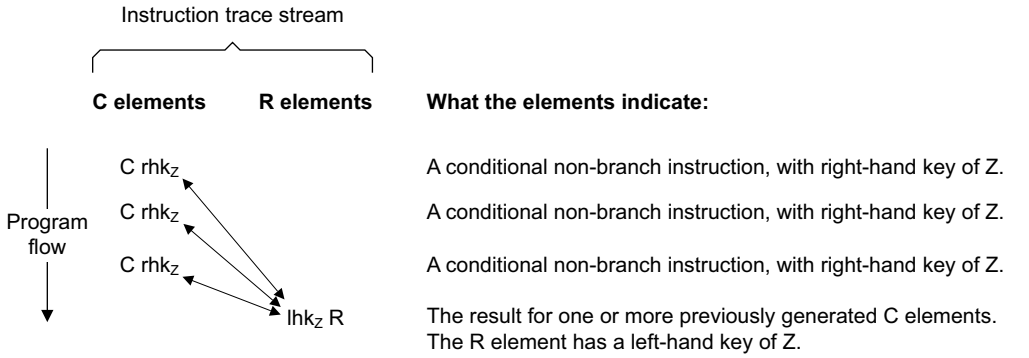


Figure 2-16 Association of C elements with an R element

The keys that are used to associate C and R elements are independent to those used to associate P0, P1, and P2 elements. That is:

- The keys between P0 and P1 elements use a particular namespace.
- The keys between P1 and P2 elements use a different namespace.
- The keys between C and R elements use another different namespace.

About P0, P1, and P2 keys on page 2-40 describes the keys that P0, P1, and P2 elements use.

F elements are generated when C elements might no longer be relevant. For example, if the PE cancels some speculative instructions because of mis-speculation, it might be necessary to discard any remaining C elements that indicate those canceled speculative instructions.

Note

When the trace unit generates an F element, that F element must occur after the most recent P0 element that indicates a batch of instructions that contains conditional instructions that might require C elements. For more information, see *About analyzing C, R, and F elements on page 2-75*.

TRCCONFIGR.COND controls the tracing of conditional non-branch instructions. Depending on the trace requirements, and to minimize trace bandwidth, a trace analyzer can choose to trace one of the following:

- No conditional non-branch instructions.
- Conditional load instructions.
- Conditional store instructions.
- Conditional load and conditional store instructions.
- All conditional non-branch instructions.

Table 2-1 shows some example scenarios.

Table 2-1 Example scenarios for conditional non-branch instruction tracing

Conditional instructions traced			Usage model
Loads	**Stores**	**All**	
N	N	N	Tracing program flow only
Y	N	N	Tracing conditional data load transfers, for example for register reconstruction
N	Y	N	Tracing conditional data store transfers, for example for memory reconstruction
Y	Y	N	Tracing all conditional data transfers
Y	Y	Y	Full code coverage analysis

About the generation of Conditional Instruction (C) elements

C elements are not necessarily generated every time the PE executes a conditional instruction. A C element is only generated if, when a conditional instruction is executed, the result of that instruction cannot be determined from previous results.

The trace unit uses one of two possible algorithms to determine whether a conditional instruction causes a C element to be generated. A trace analyzer must follow the same algorithm so that it can associate C elements with the correct conditional instructions.

The possible algorithms are:

- Tracing of the APSR condition flag values, see *The algorithm for tracing the APSR condition flag values*.
- Tracing the pass or fail result, see *The algorithm for tracing the pass or fail result on page 2-74*.

TRCIDR0.CONDTYPE indicates which algorithm is implemented.

The algorithm for tracing the APSR condition flag values

For an Arm architecture PE, four markers are required, one for each APSR condition flag, N, Z, C, and V. The algorithm operates as follows:

- Initially, after the trace unit is first enabled and a Trace Info element is generated, these flag markers are all cleared to 0.

- After analysis of program execution starts, when the first conditional instruction is executed, a C element is generated and those flags whose status is required by that instruction have their markers set to 1.
- For subsequent conditional instructions:
  - if an instruction requires only the status of flags whose markers are already set, then no C element is generated.
  - if an instruction requires the status of flags whose markers are not set, then a C element is generated and the markers for those flags are set to 1.
- The markers are cleared to 0, indicating that no C elements have been generated, whenever any of the following occur:
  - Instruction execution updates the APSR condition flags.
  - A Conditional Flush element is generated.

In this way, each flag marker indicates whether a conditional instruction that requires the status of that particular flag has had a C element generated. This means that each flag marker indicates whether the value of the flag it represents has been sent to the trace analyzer.

The `TracingAPSRValues()` function is:

```
//TracingAPSRValues()
//===================

TracingAPSRValues()
    If instruction is conditional:
        If any markers for flags required by this instruction are clear then:
            Set markers for required flags
            Trace C element
    If instruction updates the APSR:
        Clear all markers
    If Conditional Flush element is generated then
        Clear all markers
```

For interrupt continuable or exception continuable instructions on an Armv6-M, Armv7-M, or an Armv8-M PE, each attempt to execute any part of the conditional instruction is treated as an attempt to execute a separate conditional instruction. This means that a C element might be traced each time there is an attempt to execute the instruction.

If using this algorithm, the corresponding R element for a C element is generated after the C element. A single R element might be associated with multiple C elements, as shown in Figure 2-16 on page 2-72.

It is not a requirement for an R element to contain correct values for all four APSR condition flags. An R element:

- Must contain correct values for those condition flags that are required to perform the condition code checks for all of the instructions that are indicated by the associated C elements.
- Might contain incorrect values for those flags that are not required.

The algorithm for tracing the pass or fail result

In this algorithm:

- A C element is generated for every conditional instruction that the PE executes.
- The corresponding R element for a C element is generated after the C element, and that R element contains a single pass or fail result.

For interrupt continuable or exception continuable instructions on an Armv6-M, Armv7-M, or an Armv8-M PE, each attempt to execute any part of the conditional instruction is treated as an attempt to execute a separate conditional instruction. This means that a C element is traced each time there is an attempt to execute the instruction.

A single R element might be associated with multiple C elements, as shown in Figure 2-16 on page 2-72

About the ordering of C, R, and F elements in relation to other elements

Note

As mentioned in *The tracing flow on page 2-32*, an ETMv4 trace unit generates two streams of trace elements that are then encoded into two streams of trace packets.

This section relates to the ordering of the elements within the element streams.

For reducing trace bandwidth, the ETMv4 architecture permits C elements in the element stream to be out of order with respect to *Atom Elements* and *Exception elements*. However, C elements must be in order with respect to F elements and other C elements, because the analysis of C elements operates using a FIFO strategy.

The ETMv4 architecture also permits R elements to be out of order with respect to *Atom Elements* and *Exception elements*, again for reducing trace bandwidth. However, R elements must occur after the C elements that they are associated with. In addition, R elements must occur before a future C element is generated that reuses the left-hand key value of the R element. This is to avoid the possibility of associating an R element with the incorrect C elements.

About analyzing C, R, and F elements

After decoding the trace protocol, a trace analyzer analyzes the trace elements. At this stage, the following process is required to successfully analyze Conditional Instruction elements:

1. Associate R elements with their parent C elements. An R element contains a left-hand key, whose value matches a right-hand key that belongs to one or more C elements. If an R element is associated with more than one C element, the right-hand key values of all those C elements are the same. See Figure 2-16 on page 2-72.

   When associating R elements with C elements:

   a. Examine the left-hand key value of the R element.
   b. Find all the previous C elements that have a right-hand key that matches this value.
   c. Ignore any C elements that have already been associated with a previous R element.

   After the R elements have been associated with their parent C elements, the combined C-R element pairs can be removed from the main instruction element stream and pushed onto the back of a separate queue. Any F elements must also be put in the queue, and:

   - The original order of C elements with respect to other C elements must be maintained. This means that it is the C elements, in the C-R element pairs, that dictate the ordering of the C-R element pairs in the queue.
   - The original order of F elements with respect to C elements must be maintained. For example, if an F element occurs in the main instruction element stream after two C elements, then when those C elements have each been paired and removed to the queue, the F element must also be put in the queue and must appear after the two C-R element pairs.

   Note

   Even though F elements must be put in the queue along with C-R element pairs, either:

   - The trace analyzer must remember the position of F elements in the main instruction element stream.
   - A copy of the F elements must remain in the main instruction element stream.

   This is because when analyzing the main instruction element stream, whenever the trace analyzer encounters an F element, it must flush the separate queue of all C-R element pairs up to and including the F element. This keeps the C-R element pair queue that is synchronized with the main instruction element stream.

2. Analyze all types of trace elements for the purpose of removing any that are mis-speculated. However, do not remove any speculative C elements or their associated R elements at this stage.

3. Analyze all elements for the purpose of reconstructing program execution. At this stage, when a trace analyzer is analyzing blocks of instructions, it might infer that conditional non-branch instructions exist within those blocks of instructions, as shown in Figure 2-17 on page 2-76.

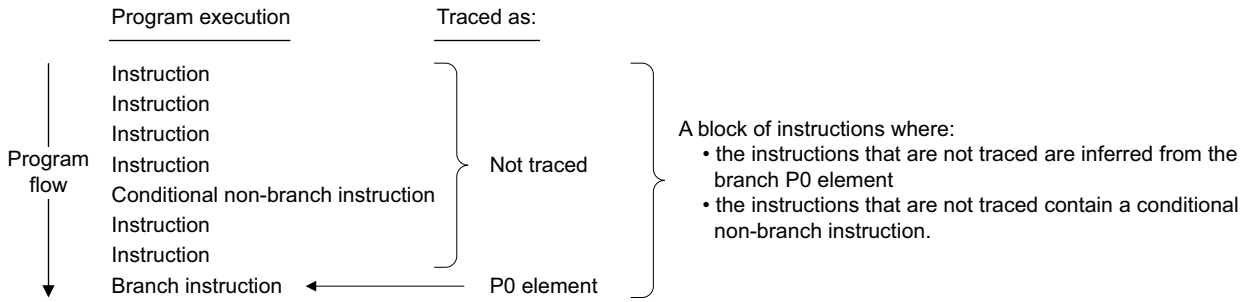


Figure 2-17 A block of instructions containing a conditional non-branch instruction

When a conditional non-branch instruction is encountered, the trace analyzer might either:

- Require a C and R element for that instruction.
- Not require a C and R element for that instruction, because the result of the condition code check for the instruction can be determined from a previous R element. For more information about how the results of a conditional non-branch instruction might be determined from previous R elements, see *About the generation of Conditional Instruction (C) elements on page 2-73*.

If a C and R element are required, then the next C-R element pair must be popped from the front of the queue. Because the queue operates using a FIFO strategy, the C-R pair is the correct one, meaning that the C element represents the correct conditional non-branch instruction. Because the C element has already been combined with the required R element, the trace analyzer can determine the result of the conditional instruction.

Analysis of Conditional Instruction (C) elements when using the algorithm for tracing the APSR condition flag values

Whenever a C-R element pair is popped from the queue, the R element contains a copy of the APSR condition flags that the trace analyzer can then use to compute the pass or fail result of the conditional non-branch instruction that the C element indicates.

Note

An R element does not necessarily contain correct values for all four condition flags in the APSR. An R element contains correct values for those flags that are required to perform the condition code check, but for the other flags, that is, those flags whose value is not required, the values might be reported as either 0 or 1.

A trace analyzer must store the values of the flags that the C elements require, because these flag values might be used for future C elements. For example, if a conditional non-branch instruction that requires the value of the Z flag is executed, then the trace unit generates a C element, followed by an associated R element that contains the correct value of the Z flag. If a second non-branch instruction is then executed, and this instruction requires the value of both the Z flag and the N flag, then the trace unit generates a C element, followed by an associated R element. In this case however, the associated R element contains the correct value of only the N flag, because the value of the Z flag is already known.

If an F element is encountered in the instruction element stream, then the C-R queue must be flushed of all C elements up to and including the next F element in the queue.

The pseudocode for the analysis of C elements when using the algorithm for tracing the APSR condition flag values is:

```
//AnalyzingCelements()
//====================

AnalyzingCelements()
    If instruction is conditional:
        If any markers for flags required by this instruction are clear:
            Set markers for required flags
            Pop next C element from the queue
            Use values from R element associated with C element to determine flags
        Use known flag values to determine whether the instruction passes its condition code check
```

```
    If instruction updates the APSR:
        Clear all markers
    If Conditional Flush element is analyzed:
        Clear all markers
```

The pseudocode for the analysis of F elements when using the algorithm for tracing the APSR condition flag values is:

```
//AnalyzingFelements()
//====================

AnalyzingFelements()
    Clear all markers
    Flush queue of all C-R element pairs up to the next F element
```

Analysis of Conditional Instruction (C) elements when using the algorithm for tracing the pass or fail result

If an F element is encountered in the instruction element stream, then the C-R queue must be flushed of all C elements up to and including the next F element in the queue.

The pseudocode for the analysis of C elements when using the algorithm for tracing the pass or fail result is:

```
//AnalyzingCelements()
//====================

AnalyzingCelements()
    If instruction is conditional:
        Pop next C element from the queue
        Use result from R element associated with the C element to determine result
```

The pseudocode for the analysis of F elements when using the algorithm for tracing the pass or fail result is:

```
//AnalyzingFelements()
//====================

AnalyzingFelements()
    Flush queue of all C-R element pairs up to the next F element
```

2.6.3 Trace behavior on tracing Jazelle execution

ETMv4 does not support tracing of execution in Jazelle state. If the PE enters Jazelle state, ViewInst becomes inactive until the PE leaves Jazelle state.

In addition, the entry to Jazelle state is not explicitly traced as an entry to Jazelle state. An instruction that causes entry to Jazelle state is always traced with an *Atom element*. This is because the instruction is always one that results in a P0 element.

There is no requirement for the target address in Jazelle state to be traced.

2.6.4 Trace behavior on tracing ThumbEE instructions

The ETMv4 architecture does not support the tracing of PEs that implement ThumbEE. If you are implementing ThumbEE on a PE and using an ETMv4 trace unit to trace that PE, please contact Arm.

2.6.5 Data trace behavior on tracing store-exclusive instructions

All store-exclusive instruction types comprise two parts:

- The data that is stored to memory.
- An indication of whether the data store is successful.

To trace the data that is stored to memory, the trace unit generates at least one P1 element, and each P1 element has an associated P2 element:

- The addresses of the data stores are traced using P1 elements.

- The data values are traced using P2 elements.

Data stores that result from store-exclusive instructions are therefore traced in the same way that data transfers from all other types of instruction are traced.

In addition, an ETMv4 trace unit treats success indicators as data transfers so that they can also be traced. The attributes of a success indicator as a data transfer are considered to be as follows:

- The data transfer is a write access.
- The access address is the same as the address of the lowest byte of the memory access. For example, for a store-exclusive instruction that performs a 32-bit data transfer to address `0x1000`:
  - The lowest byte of the memory access is at `0x1000`.
  - The second byte of the memory access is at `0x1001`.
  - The third byte of the memory access is at `0x1002`.
  - The highest byte of the memory access is at `0x1003`.

  The access address of the success indicator therefore, is `0x1000`.

  The reason that the access address of a success indicator is always considered to be the same as the lowest address byte of the memory access is because it ensures that the success indicator is always traced, even when address-based filtering is applied.
- The access size is considered to be the same as the size of the data transfer performed.

To trace a success indicator, the trace unit generates a P1 and a P2 element:

- The success indicator value is traced using a P2 element.
- The P1 element provides keys so that the P2 element can be associated with the P0 element, that is, so that the success indicator can be associated with the correct store-exclusive instruction. However, there is no requirement for the P1 element to contain the access address of the success indicator, because the access address is the same as the lowest address byte of the memory access. That is, the access address is the same as the address given in the P1 element that traced the data store.

  Note

  There are several scenarios that might result in P1 elements that do not contain addresses. For more information, see *Occasions when P1 elements are traced without the address or endianness of the data transfer on page 5-229*.

Table 2-2 gives examples of how P1 elements and P2 elements are used to trace data transfers initiated by store-exclusive instructions.

Table 2-2 Example data transfers for A32 and T32 store-exclusive instructions

Instruction	Accesses performed	Type of access	P1 element transfer index	P2 element contains:
`STREXB` to `0x1000`	Byte write at `0x1000`	Data store	0	Data value
	Byte write at `0x1000`	Success indicator	1	Success indicator value
`STREXH` to `0x1000`	Halfword write at `0x1000`	Value stored	0	Data value
	Halfword write at `0x1000`	Success indicator	1	Success indicator value
`STREX` to `0x1000`	Word write at `0x1000`	Value stored	0	Data value
	Word write at `0x1000`	Success indicator	1	Success indicator value

Table 2-2 Example data transfers for A32 and T32 store-exclusive instructions (continued)

Instruction	Accesses performed	Type of access	P1 element transfer index	P2 element contains:
STREXD to 0x1000	Word write at 0x1000	Value stored [31:0]	0	Data value
	Word write at 0x1004	Value stored [63:32]	1	Data value
	Word write at 0x1000[a]	Success indicator	2	Success indicator value

a. For STREXD, the success indicator is word-sized, not doubleword-sized.

In Table 2-2 on page 2-78, the meaning of the P1 element transfer index value depends on whether the P1 element is for a data store or a success indicator:

- If the P1 element is for a data store, the transfer index indicates the address of the data store as an offset from the base address that is accessed by the instruction.
- If the P1 element is for a success indicator, the transfer index indicates that the value given in the associated P2 element is the value of the success indicator.

For more information, see *P1 element transfer index meanings on page F-477*.

2.7 Optional features

The ETMv4 architecture includes the following optional features that can be implemented:

Context ID tracing

If a trace unit implementation includes support for Context ID tracing, it can be programmed to output the Context ID of the process that the PE is executing. For more information, see *Context ID tracing on page 2-81*.

Virtual context identifier tracing

If a trace unit implementation includes support for virtual context identifier tracing, it can be programmed to output the identifier of a virtual machine that the PE is executing. For more information, see *Virtual context identifier tracing on page 2-81*.

Cycle counting

If a trace unit implementation includes support for cycle counting, it can be programmed to count and report the number of processor clock cycles that occur between two Commit elements. For more information, see *Cycle counting on page 2-81*.

Global timestamping

If a trace unit implementation includes support for global timestamping, and if a timestamp source is available in the system, the trace unit can be programmed to periodically output the timestamp value into the trace streams. For more information, see *Global timestamping on page 2-82*.

Branch broadcasting

If a trace unit implementation includes support for branch broadcasting, it can be programmed to explicitly trace the target addresses of direct branch and ISB instructions. For more information, see *Branch broadcasting on page 2-83*.

Conditional instructions tracing

All ETMv4 trace unit implementations always trace conditional branch instructions. However, an implementation might also include support for tracing conditional non-branch instructions. If an implementation includes this support, it can be programmed to also trace either:

- No conditional non-branch instructions.
- Conditional load instructions only.
- Conditional store instructions only.
- Conditional load and conditional store instructions.
- All conditional non-branch instructions.

For more information, see *Conditional instructions tracing on page 2-84*.

Explicit tracing of data load and store instructions

As described in *About instruction trace P0 elements on page 2-35*, whether a trace unit supports the explicit tracing of load and store instructions is IMPLEMENTATION DEFINED. If it does, it can be programmed so that either:

- No data load or store instructions are traced explicitly.
- Data load instructions are traced explicitly.
- Data store instructions are traced explicitly.
- Both data load and data store instructions are traced explicitly.

For more information, see *Explicit tracing of data load and store instructions on page 2-84*.

Data tracing If a trace unit implementation includes support for data tracing, it can be programmed so that whenever the PE performs a data transfer, for example as a result of a load or store instruction, it traces either or both of the following:

- The data address of the data transfer.
- The data value of the data transfer.

For more information, see:

- *Data address tracing on page 2-84*.
- *Data value tracing on page 2-84*.

Note

As mentioned in *About instruction trace P0 elements on page 2-35*, an implementation that includes support for data tracing must also include support for explicitly tracing data load and store instructions.

Q elements Whether an implementation supports Q elements is IMPLEMENTATION DEFINED. If they are supported, then the trace unit can be programmed so that an individual P0 element is not necessarily generated for each of the instructions that are described in *About instruction trace P0 elements on page 2-35*. Instead, a Q element is generated to indicate the execution of multiple instructions that would otherwise be traced as P0 elements.

For more information, see *Q elements on page 2-85*.

The following sections describe each of these optional features.

2.7.1 Context ID tracing

Whether an implementation supports Context ID tracing is IMPLEMENTATION DEFINED. If it does, the trace unit can be programmed to output the Context ID of the process that the PE is executing.

This option is enabled by setting TRCCONFIGR.CID to 1.

TRCIDR2.CIDSIZE indicates if support for tracing the Context ID is implemented. If it is, then TRCCONFIGR.CID is a RW field.

For an Armv7 PE, or an Armv8 PE in AArch32 state, the value of the Context ID is the value of the current Context ID Register, CONTEXTIDR.

For an Armv8 PE in AArch64 state, the value of the Context ID is the value of the current Context ID Register, CONTEXTIDR_EL1.

2.7.2 Virtual context identifier tracing

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.

This option is enabled by setting TRCCONFIGR.VMID to 1.

TRCIDR2.VMIDSIZE indicates if support for tracing a Virtual context identifier is implemented. If it is, then TRCCONFIGR.VMID is a RW field.

In ETMv4.0, the value of the Virtual context identifier is the value that is stored in the VMID field of the VTTBR.

From ETMv4.1, for Armv8-A, the value of the Virtual context identifier is one of:

- The value that is stored in the VMID field of the VTTBR.
- The value that is stored in the CONTEXTIDR_EL2 register.

Use TRCCONFIGR.VMIDOPT to choose which value is used by the trace unit.

From ETMv4.2, for Armv8-R, the value of the Virtual context identifier is the value that is stored in the VMID field of the VSCTLR.

2.7.3 Cycle counting

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.

Whether an implementation supports cycle counting is IMPLEMENTATION DEFINED. If it does, the trace unit can be programmed to generate Cycle Count elements. Cycle Count elements are associated with Commit elements, so that when a Commit element is generated, a Cycle Count element might also be generated. A Cycle Count element indicates the number of processor cycles between the two most recent Commit elements that both had a cycle count value that is associated with them. Some Commit elements do not have a cycle count value that is associated with them.

When cycle counting is enabled and when a cycle count is traced, the cycle count must correctly represent the number of processor cycles, except where:

- The cycle counter is permitted to stop counting, as described in *Trace unit behavior on a PE low-power state on page 3-105* and *Trace unit behavior when tracing is prohibited on page 3-108*.
- The cycle counter value is permitted to be UNKNOWN, as described in *Cycle Count instruction trace element on page 5-216*.

To reduce trace bandwidth, the ETMv4 architecture only requires a Cycle Count element to be generated if the cycle count value exceeds a minimum threshold value at the time when a Commit element is generated. For example, if the minimum threshold value is set to 16 cycles, and the trace unit generates a Commit element:

- If the cycle count is less than 16, the trace unit does not generate a Cycle Count element.
- If the cycle count is 16 or more, then the trace unit generates a Cycle Count element that contains the value of the cycle count, and the cycle counter is reset.

If it is supported, cycle counting is enabled by performing both of the following:

- Setting TRCCONFIGR.CCI to1.
- Programming the TRCCCCTLR. This sets the cycle count threshold value.

TRCIDR0.TRCCCI indicates if support for cycle counting is implemented. If it is, then the TRCCCCTLR is implemented and TRCCONFIGR.CCI is a RW field.

2.7.4 Global timestamping

The ETMv4 architecture provides optional support for global timestamping. Whether this support is included in an implementation is IMPLEMENTATION DEFINED. If it is, then the trace unit has a mechanism where a timestamp value that is global to the system can be inserted into the trace streams periodically.

These timestamps can be used to achieve:

- Approximate correlation of the data trace stream with the instruction trace stream.
- Correlation of multiple independent trace sources in a system, for example, multiple trace units in an environment with multiple PEs.
- Simple analysis of code performance, with a coarse granularity.
- Faster searching of large trace buffers when multiple streams are output and related pieces of code in each stream are required.

To use this feature, the system must contain a timestamp source, and must simultaneously broadcast the same timestamp value to all trace sources in the system. Each independent trace unit can then sample the timestamp value on request and insert it as an absolute value into their respective trace streams.

For implementations that support global timestamping, the ETMv4 architecture permits maximum timestamp values of either 48 bits or 64 bits. Where the PE implements Armv8.4-Trace, global timestamping must be implemented, and the maximum timestamp value must be 64 bits. Whether an implementation supports a maximum timestamp value of 48 or 64 bits is IMPLEMENTATION DEFINED.

When global timestamping is enabled, the trace unit automatically inserts global timestamps into the trace streams at points where they are likely to be useful, such as:

- After the trace unit:
  - Has generated a Trace Info element. This is true for the instruction trace stream and, if it is supported and enabled, the data trace stream.

  - Has recovered from a trace buffer overflow.

- Whenever the PE:
  - Takes an exception.
  - Returns from an exception handler.
  - Executes an `ISB` instruction.

- Whenever a flush of the trace unit is requested.

In addition, the TRCTSCTLR is provided so that the trace unit can be programmed to insert global timestamps into the trace streams at specified points, which are based on events that occur in the trace unit.

If it is supported, global timestamping is enabled by setting TRCCONFIGR.TS to 1.

TRCIDR0.TSSIZE indicates if support for global timestamping is implemented. If it is, then:

- TRCIDR0.TSSIZE also indicates what maximum timestamp value size is implemented.
- The TRCTSCTLR is implemented.
- TRCCONFIGR.TS is a RW field.

When tracing a PE that implements Armv8.4-Trace, global timestamping is mandatory, with a 64-bit timestamp size.

Note

Support for global timestamping is always implemented when support for data tracing is implemented.

The global timestamp source must be tolerant of low-power or powerdown scenarios. The global timestamp value must not be reset in these scenarios if tracing is expected to continue after the low-power or powerdown scenario finishes.

Arm recommends that the global timestamp source increments at a constant rate relative to real-time, and that the update frequency of the timestamp is no less than 10% of the clock speed of the PE being traced.

2.7.5 Branch broadcasting

Whether an implementation supports branch broadcasting is IMPLEMENTATION DEFINED. If it does, the trace unit can be programmed so that it explicitly traces the target addresses of direct branch and ISB instructions that the PE executes. The target addresses are traced using Address elements in the instruction trace stream.

Branch broadcasting is enabled by performing both of the following:

- Setting TRCCONFIGR.BB to 1.

- Programming the TRCBBCTLR to specify how branch broadcasting behaves. When programming this register, particular memory ranges can be selected by providing the addresses for those ranges. The trace unit can then be programmed so that either:
  - Branch broadcasting is active for all the branch instruction addresses inside those ranges. This is known as *include mode*.
  - Branch broadcasting is active for all the branch instruction addresses outside of those ranges. This is known as *exclude mode*.

TRCIDR0.TRCBB indicates if support for branch broadcasting is implemented. If TRCIDR0.TRCBB is 1, then TRCCONFIGR.BB is a RW field. If TRCIDR0.TRCBB is 1 and TRCIDR4.NUMACPAIRS is greater than zero, then TRCBBCTLR is also implemented. If TRCBBCTLR is not implemented, then when branch broadcasting is enabled, branch broadcasting is active for all branch instruction addresses.

2.7.6 Conditional instructions tracing

Trace behavior on tracing conditional instructions on page 2-71 describes the tracing of conditional instructions. All trace unit implementations trace conditional branch instructions. However, whether an implementation supports the tracing of conditional non-branch instructions is IMPLEMENTATION DEFINED. If it does, the trace unit can be programmed so that it also traces either:

- No conditional non-branch instructions.
- Conditional load instructions only.
- Conditional store instructions only.
- Conditional load instructions and conditional store instructions.
- All conditional non-branch instructions.

If it is supported, the tracing of conditional non-branch instructions is enabled by setting TRCCONFIGR.COND to a nonzero value.

TRCIDR0.TRCCOND indicates if support for the tracing of conditional non-branch instructions is implemented. If it is, then TRCCONFIGR.COND is a RW field.

TRCIDR0.CONDTYPE indicates whether conditional results are traced with R elements that show pass or fail results, or with R elements that show the values of the APSR condition flags.

2.7.7 Explicit tracing of data load and store instructions

Certain types of instructions, and some events, are always explicitly traced. That is, certain types of instructions and events are always traced as P0 elements. These are listed in *About instruction trace P0 elements on page 2-35*.

Support for explicitly tracing data load and store instructions is optional. In a trace unit implementation that includes this support, the explicit tracing of load and store instructions is enabled by setting TRCCONFIGR.INSTP0 to a nonzero value. The configuration options are:

- Do not trace load instructions or store instructions explicitly. Trace only those items that are listed in *About instruction trace P0 elements on page 2-35* explicitly.
- In addition to those items that are always traced explicitly, trace load instructions explicitly.
- In addition to those items that are always traced explicitly, trace store instructions explicitly.
- In addition to those items that are always traced explicitly, trace load instructions explicitly and trace store instructions explicitly.

TRCIDR0.INSTP0 indicates if support for explicitly tracing load and store instructions is implemented. If it is, then TRCCONFIGR.INSTP0 is a RW field.

Explicit tracing of data load and store instructions is only implemented when data tracing is implemented.

2.7.8 Data address tracing

When the PE executes instructions that perform data transfers, such as loads or stores, an ETMv4 trace unit that supports data tracing can be programmed to output the addresses of those data transfers in the data trace stream.

This option is enabled by setting TRCCONFIGR.DA to 1.

TRCIDR0.TRCDATA indicates whether the trace unit supports data tracing. If it does, both TRCCONFIGR.{DA,DV} are RW fields.

A trace unit can only output data address information for load and store instructions that are explicitly traced.

Therefore, the field that enables the explicit tracing of load and store instructions, TRCCONFIGR.INSTP0, must not be `0b00` when setting TRCCONFIGR.DA to 1.

2.7.9 Data value tracing

When the PE executes instructions that perform data transfers, such as loads or stores, an ETMv4 trace unit that supports data tracing can be programmed to output the values of those data transfers in the data trace stream.

This option is enabled by setting TRCCONFIGR.DV to 1.

TRCIDR0.TRCDATA indicates whether the trace unit supports data tracing. If it does, both TRCCONFIGR.{DA,DV} are RW fields.

A trace unit can only output data value information for load and store instructions that are explicitly traced. Therefore, if setting TRCCONFIGR.DV to 1, the field that enables the explicit tracing of load and store instructions, TRCCONFIGR.INSTP0, must not be set to `0b00`.

Note

If data value tracing is enabled but data address tracing is disabled, the trace unit generates P1 elements that provide links between P2 elements and P0 elements. For more information, see Table 5-10 on page 5-228.

2.7.10 Q elements

When the trace unit supports Q elements, it can be programmed so that an individual P0 element is not necessarily generated for each of the instructions that are described in *About instruction trace P0 elements on page 2-35*. A Q element is generated instead, indicating that at least one instruction was executed, and that zero or more branch or `ISB` instructions might have occurred.

This option is enabled by setting TRCCONFIGR.QE to `0b01` or `0b11`. TRCIDR0.QSUPP indicates whether the trace unit includes support for Q elements. The value of QSUPP determines the values that are supported by TRCCONFIGR.QE.

A Q element:

- Is a P0 element in the instruction trace stream, and must therefore be explicitly committed or canceled.
- Indicates that at least one instruction has been executed.
- Optionally includes a count of the number of instructions that are executed since the most recent P0 element.
- Is always followed by at least one Address element before the next P0 element.

The Address element that follows the Q element indicates where execution continues after all the instructions implied by the Q element have been executed.

When the trace unit has been programmed to use Q elements, the information in the instruction trace stream might not provide sufficient information to determine the execution of every instruction, since not every change in program flow is explicitly indicated in the trace stream.

Q elements can only be used when data trace and conditional non-branch tracing are either not implemented or not enabled.

Q elements are only expected to be used in cases where generating a full ETMv4 instruction trace stream might cause the performance of the PE being traced to degrade significantly.

Q elements are not supported on a trace unit for an Armv7-M, Armv7-R, Armv8-M, or Armv8-R PE.

Chapter 3
About the Trace Unit

This chapter describes the trace unit, and the behavior of the trace unit. It contains the following sections:

3.1 Functions of the trace unit

A trace unit has two main functions:

- To generate trace, by producing trace streams, that can be either:
  - Exported off-chip to an external trace analyzer, known as external debug.
  - Captured on-chip for analysis by on-chip software, known as self-hosted debug.
- To enable filtering of the trace streams, so that:
  - Instruction tracing, and data tracing if it is enabled, can be made active only for particular threads of execution of the program code, or only for specific functions that the PE performs.
  - Data tracing can be selective, and active only for particular data transfers. For example, data trace filtering is useful when there is a requirement to trace all accesses to a particular peripheral.
  - The trace bandwidth and trace storage overheads are reduced.

The following two sections describe these two functions:

- *Trace generation*.
- *Trace filtering on page 3-89*.

3.1.1 Trace generation

As described in *The tracing flow on page 2-32*, a trace unit traces PE execution by generating *trace elements*. The ETMv4 architecture defines the generation of these trace elements from the execution of the PE.

An ETMv4 trace unit can generate two trace element streams:

- An instruction trace element stream.
- A data trace element stream, if data tracing is implemented and enabled.

These two streams of trace elements are then encoded into two streams of trace packets:

- A stream of instruction trace packets. While the trace unit is enabled, this stream is always output.
- A stream of data trace packets. This stream is only output if the trace unit is enabled and data tracing is supported and enabled.

A trace unit can be programmed so that it provides either:

- The instruction trace stream that generates only program flow information.
- Both the instruction and data trace streams, offering full instruction and data tracing.

This depends on the optional features that are implemented, and whether those features are enabled before a trace run.

The following optional features might be implemented, and if implemented, can be enabled and disabled according to requirements:

- Context ID tracing.
- *Virtual context identifier* tracing.
- Cycle counting.
- Global timestamping.
- Branch broadcasting.
- Conditional instructions tracing.
- Explicit tracing of data load and store instructions.

The instruction trace stream contains instruction execution information. The data trace stream contains the addresses and data values of data transfers that the PE carries out.

For more information, see *Separate instruction and data trace streams on page 2-33* and *Optional features on page 2-80*.

For more information about trace generation in general, see Chapter 2 *About the Trace Streams*.

3.1.2 Trace filtering

The ETMv4 architecture supports the implementation of a range of trace unit resources, that can be used to enable and disable tracing based on PE events. The resources that are provided include:

- Up to 16 single address comparators, that can be either programmed to match on a single address or combined in pairs to match on address ranges.
- Up to eight data value comparators, for use with data address comparators.
- Up to four external input selectors.
- Up to eight inputs from the PE comparators.
- Up to eight Context ID comparators.
- Up to eight Virtual context identifier comparators.
- Up to four 16-bit counters.
- A sequencer state machine with up to four states.
- Up to eight single-shot comparators.

The trace unit can be programmed to use these resources as inputs to the ViewInst and ViewData functions. The ViewInst function enables and disables instruction tracing in the instruction trace stream. The ViewData function enables and disables data tracing in the data trace stream. See *The ViewInst function on page 4-119* and *The ViewData function on page 4-132*.

3.2 Trace unit block diagram

An ETMv4 trace unit consists of the following functional blocks:

- For instruction tracing:
  - Instruction trace element generation.
  - Instruction trace protocol generation.
  - Instruction trace buffer.
- For data tracing:
  - Data trace element generation.
  - Data trace protocol generation.
  - Data trace buffer.
- For programming the trace unit:
  - Programming interface with filtering and control logic.

The trace buffers provide temporary storage for the trace streams, for smoothing over any peaks in trace generation.

Figure 3-1 shows the functional blocks of the trace unit.

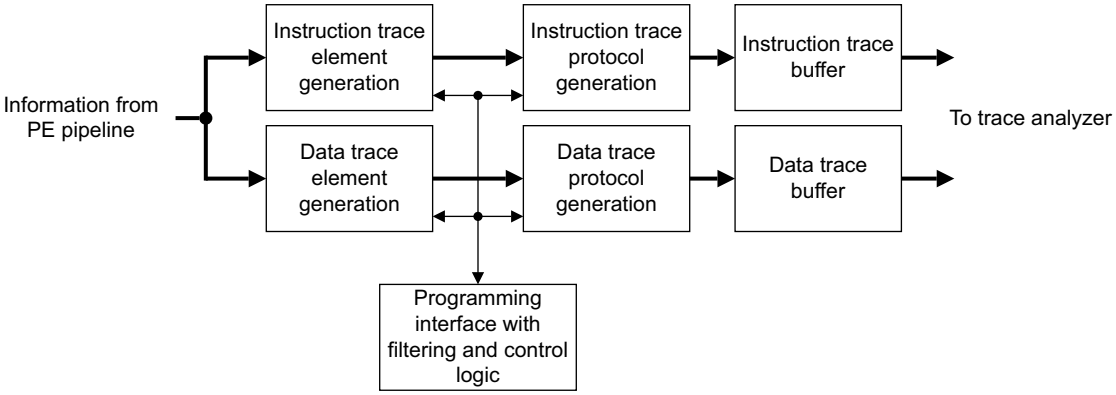


Figure 3-1 Trace unit block diagram

3.3 Trace unit power domains

An ETMv4 trace unit has two logical power domains:

- A core power domain, which contains most of the trace unit logic, including all those trace unit registers that are defined as *trace registers*, and two registers that are defined as *management registers*.

  Note

  Notwithstanding their management register classification, the OS Lock registers, TRCOSLAR and TRCOSLSR, are located in the core power domain.

- A debug power domain, which contains the external debugger programming interface, and also includes all the trace unit management registers except TRCOSLAR and TRCOSLSR.

An ETMv4 trace unit can implement one of the three following power structures:

- Multiple Power Domain Model:
  - In a multiple power domain system, the trace unit core power domain and the trace unit debug power domain are separate and can be independently powered up or powered down.
  - In some systems, the trace unit core power domain is in the same power domain as the PE core power domain. However, some systems might put the trace unit core power domain in a separate power domain from the PE core power domain, so that the trace unit can be powered down when not in use.
  - For more information, see *If a trace unit is implemented in multiple power domains on page 3-101*,
- Single Power Domain Model:
  - In a single power domain system, the trace unit core power domain and the trace unit debug power domain are in the same power domain, and the trace unit is either all powered up or all powered down.
  - In some systems, this single power domain is in the same power domain as the PE core power domain. However, some systems might put the single trace unit power domain in a separate power domain from the PE core power domain, so that the trace unit can be powered down when not in use.
  - For more information, see *If a trace unit is implemented in a single power domain on page 3-102*,
- Unified Power Domain Model:
  - In a Unified Power Domain Model system, the trace unit debug power domain does not exist and is fully incorporated in to the trace unit core power domain, and the trace unit is either all powered up or all powered down.
  - In some systems, this single power domain is in the same power domain as the PE core power domain. However, some systems might put the single trace unit power domain in a separate power domain from the PE core power domain, so that the trace unit can be powered down when not in use.
  - The Unified Power Domain Model is introduced from ETMv4.2.
  - The Unified Power Domain Model differs from the Single Power Domain Model by simplifying some registers, and placing requirements on the system that ensure the trace unit can be powered up by an external debugger.
  - For more information, see *If a trace unit is implemented in a unified power domain on page 3-103*,

The *Register map overview on page 4-164* shows which registers are trace registers and which registers are management registers.

Generation of the trace streams might occur in either the trace unit core power domain, or the trace unit debug power domain.

Some management registers are accessible in both the trace unit core power domain, and the trace unit debug power domain. TRCDEVID, TRCAUTHSTATUS, and TRCDEVARCH are accessible by system instructions even when the trace unit debug power domain is off, and are also accessible from the external debugger interface even when the trace unit core power domain is off. For more details on the accessibility of these registers, see *Access permissions on page 7-340*.

In a typical CoreSight system, the trace unit debug power domain is connected to the system debug power domain, which powers all the debug and trace components. This system allows all the debug and trace components to be powered down when not in use.

Figure 3-2 shows an example of a system where the core power domain of the trace unit is connected to the PE core power domain, and the debug power domain of the trace unit is connected to the system debug power domain.

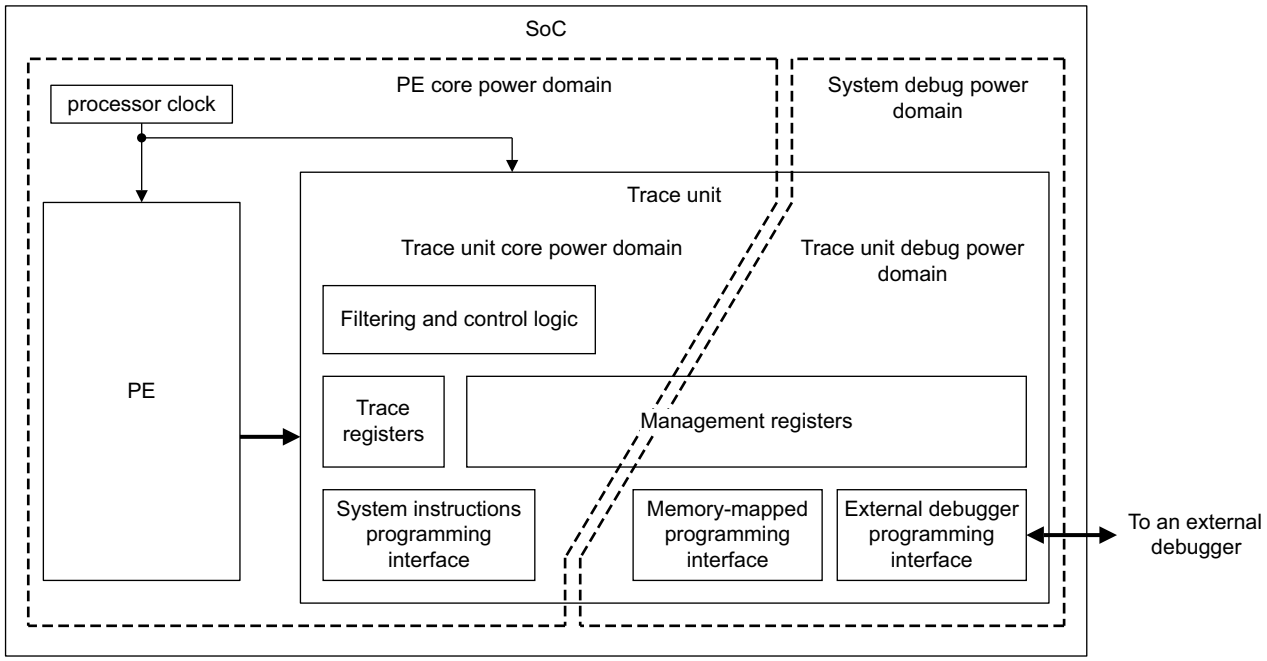


Figure 3-2 An example where the trace unit core power domain is connected to the PE core power domain

To achieve a trace unit implementation that can be fully powered down when not in use, the following conditions are required:

- The core power domain of the trace unit is not part of the PE core power domain. The core power domain of the trace unit is connected to a system power domain that is ordinarily powered down during the normal operating mode of the system.
- The debug power domain of the trace unit is connected to the system debug power domain, and the system debug power domain is ordinarily powered down during the normal operating mode of the system.

This type of system is shown in Figure 3-3 on page 3-93.

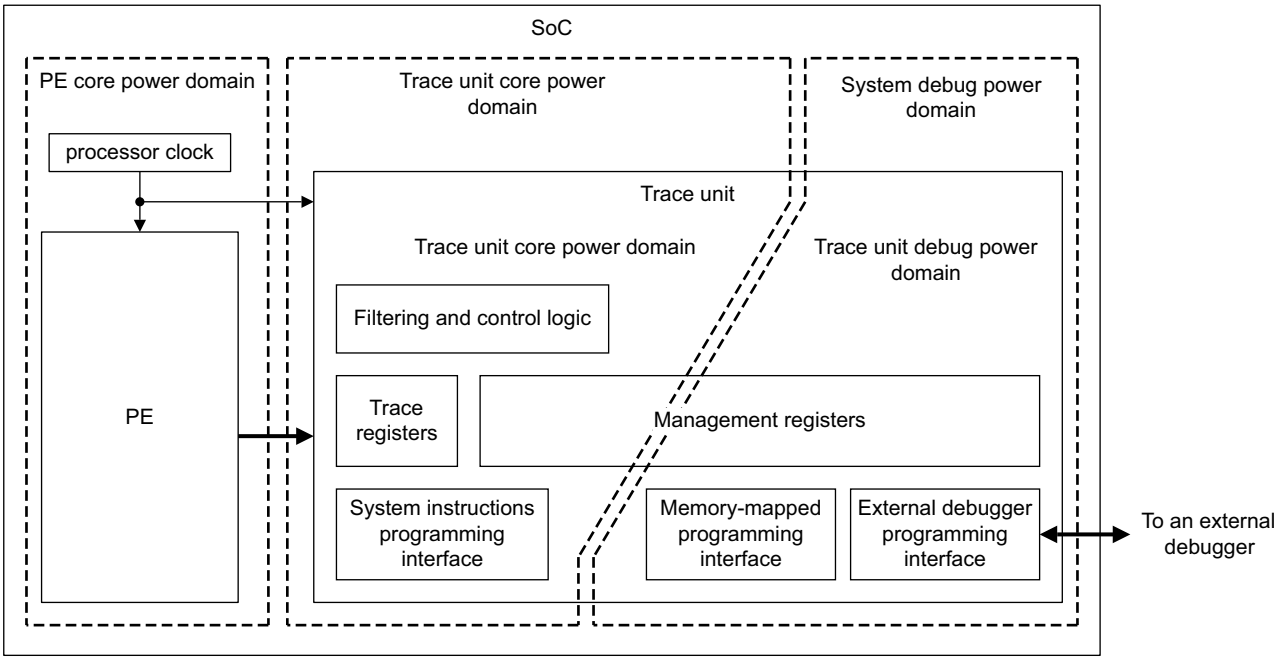


Figure 3-3 An example where the trace unit core power domain is separate from the PE core power domain

Arm recommends that the debug power domain of the trace unit is always powered when an external debugger is connected. This ensures that the management registers are always accessible to an external debugger, and that communication with the trace unit can be maintained while the trace unit core power domain is being dynamically powered down. When the trace unit core power domain is powered down, the TRCPDSR indicates that it is powered down, and therefore that the trace unit trace registers are inaccessible.

3.4 Trace unit powerdown support

The ETMv4 architecture includes optional powerdown support for a trace unit. The main features of this support are:

- The state of the trace unit can be saved before it is powered down, which means that on powering up the trace unit, the trace unit state can be restored. As mentioned in *Trace unit power domains on page 3-91*, the trace unit registers are categorized according to which power domain they are in:
  - The *trace registers* are located in the trace unit core power domain.
  - The *management registers*, apart from TRCOSLAR and TRCOSLSR, are located in the trace unit debug power domain. TRCDEVID, TRCAUTHSTATUS, and TRCDEVARCH are accessible in both the trace unit core power domain, and the trace unit debug power domain. For more details on the accessibility of these registers, see *Access permissions on page 7-340*.

  The trace unit state is held in the trace registers. Therefore, it is the trace registers that can be saved and restored.
- The provision of an OS Lock that prevents accesses to the trace registers from an external debugger. See *TRCOSLAR, OS Lock Access Register on page 7-390*.
- The provision of the TRCPDSR that is in the debug power domain:
  - Displays the power status of the core power domain.
  - Indicates when the state of the trace unit has been lost because of a core power domain powerdown.

The remainder of this section is organized as follows:

- *The procedure when powering down the PE*.
- *Behavior when the OS Lock is locked on page 3-95*.
- *Guidelines for trace unit registers to be saved and restored on page 3-96*.

3.4.1 The procedure when powering down the PE

If the trace unit core power domain is separate from the PE core power domain, as shown in Figure 3-3 on page 3-93, the PE can be powered down without losing the trace unit state.

If the trace unit core power domain is part of the PE core power domain, as shown in Figure 3-2 on page 3-92, then the trace unit state can be saved before powering down the domain.

The trace unit state can also be saved before powering down the trace unit core power domain.

Saving and restoring the trace unit state

To save the trace unit state, on-chip software must use the following procedure:

1. Execute a `DSB` instruction.
2. Execute an `ISB` instruction.
3. If you are using a memory-mapped interface to access the trace unit, unlock the Software Lock by using the TRCLAR, if the Software Lock is implemented.
4. Lock the OS Lock by using the TRCOSLAR, which disables external debugger accesses to the trace registers.
5. Poll TRCSTATR.PMSTABLE until the programmers' model becomes stable.
6. Manually read the trace unit trace registers and save the contents to memory that does not lose power when the trace unit core power domain is powered down. See *Guidelines for trace unit registers to be saved and restored on page 3-96*.
7. Poll TRCSTATR.IDLE to ensure that the trace unit is idle and can therefore be powered down.
8. If system instructions are being used to access the trace unit, use the CPACR or CPACR_EL1 in the PE to disable accesses to the trace unit registers. See *Arm®v8 Architecture Reference Manual*.

9. The trace unit core power domain can now be powered down.

Note

- Do not unlock the OS Lock before powering down.
- The Arm architecture provides a guarantee of observability for certain System registers. None of the ETMv4 System registers provide a guarantee of observability, and therefore require explicit synchronization when accessed using system instructions. In particular, whenever disabling or enabling the trace unit, a poll of TRCSTATR needs explicit synchronization between each read of TRCSTATR.

If this procedure is terminated early, for example if an event prevents the PE from powering down so that this procedure does not complete, and if the OS Lock is unlocked before TRCSTATR.IDLE indicates that the trace unit is idle, then if the trace unit becomes active when the OS Lock is unlocked:

- The trace unit might not restart tracing immediately.
- The trace unit resources might not become active immediately.

To restore the trace unit state when the trace unit is powered up again, on-chip software must use the following procedure:

1. If the memory-mapped interface is being used to access the trace unit, unlock the Software Lock by using the TRCLAR, if the Software Lock is implemented.
2. If system instructions are being used to access the trace unit, use the CPACR or CPACR_EL1 in the PE to enable accesses to the trace unit registers. See *Arm®v8 Architecture Reference Manual*.
3. Check that the OS Lock is locked by reading TRCOSLSR.OSLK to see if it is set to 1.

   Note

   The OS Lock is locked on a trace unit reset. See *Trace unit behavior on a trace unit reset on page 3-98*. Therefore, this step is a check to ensure that the OS Lock is locked, and therefore that accesses to trace registers by an external debugger are disabled.
4. Manually restore the trace unit trace registers from the memory that they were saved to.
5. Unlock the OS Lock by using the TRCOSLAR, which enables accesses from an external debugger.

3.4.2 Behavior when the OS Lock is locked

When the OS Lock is locked, the trace unit is disabled. See *Trace unit behavior when the trace unit is disabled on page 3-100*.

This behavior is similar to the behavior of the trace unit when TRCPRGCTLR.EN is `0b0`, but there are the following differences:

- Accesses to the trace registers from an external debugger cause an error response.
- The OS Lock is locked, as indicated by TRCOSLSR and TRCPDSR.
- The value of TRCPRGCTLR.EN is not affected by locking or unlocking the OS Lock. TRCPRGCTLR.EN remains writable using system instructions or memory-mapped accesses while the OS Lock is locked.

For full details on access permissions to registers while the OS Lock is locked, see *Access permissions on page 7-340*.

3.4.3 Effect of the OS Double Lock

It is IMPLEMENTATION DEFINED whether the OS Double Lock affects the trace unit.

When the OS Double Lock affects the trace unit and the OS Double Lock is locked, then the trace unit behaves as if it is in the “no core power” state.

When the Unified Power Domain Model is implemented, if the PE implements the OS Double Lock, the OS Double Lock does not affect the trace unit.

3.4.4 Guidelines for trace unit registers to be saved and restored

When saving the trace unit state, all registers that lose state when the trace unit core power domain powers down must be saved. Typically, these registers are all the trace unit trace registers. In addition, if the IMPLEMENTATION DEFINED registers TRCIMSPEC0-7 are located in the trace unit core power domain, the state of these registers must also be saved. An implementation might include TRCIMSPEC0-7 in either the trace unit core power domain or the trace unit debug power domain.

The trace unit trace registers that must be saved are:

- The main control registers:
  - TRCPRGCTLR.
  - TRCPROCSELR.
  - TRCCONFIGR.
  - TRCAUXCTLR.
  - TRCEVENTCTL0R.
  - TRCEVENTCTL1R.
  - TRCQCTLR.
  - TRCTRACEIDR.
  - TRCSTALLCTLR.
  - TRCTSCTLR.
  - TRCSYNCPR.
  - TRCCCCTLR.
  - TRCBBCTLR.
  - TRCQCTLR.
- The filtering control registers:
  - TRCVICTLR.
  - TRCVIIECTLR.
  - TRCVISSCTLR.
  - TRCVIPCSSCTLR.
  - TRCVDCTLR.
  - TRCVDSACCTLR.
  - TRCVDARCCTLR.
- The derived resources registers:
  - TRCSEQEVR0-3.
  - TRCSEQRSTEVR.
  - TRCSEQSTR.
  - TRCCNTRLDVR0-3.
  - TRCCNTVR0-3.
  - TRCCNTCTLR0-3.
  - TRCEXTINSELR.
- The resource selection registers:
  - TRCRSCTLR2-31.
- The comparator registers:
  - TRCACVR0-15.
  - TRCACATR0-15.
  - TRCDVCVR0-7.
  - TRCDVCMR0-7.

  - TRCCIDCVR0-7.
  - TRCCIDCCTLR0-1.
  - TRCVMIDCVR0-7.
  - TRCVMIDCCTLR0-1.

- The single-shot comparator registers:
  - TRCSSCCR0-7.
  - TRCSSCSR0-7.
  - TRCSSPCICR0-7.

- The claim tag registers:
  - TRCCLAIMSET.
  - TRCCLAIMCLR.

  The claim tags must also be saved and restored. When saving, read TRCCLAIMCLR and save this value. When restoring, write the saved value to TRCCLAIMSET.

See *Register summary on page 7-336* for details of the registers in this list.

3.5 Trace unit behavior

The following sections describe:

- *Trace unit behavior on a trace unit reset*.
- *Trace unit behavior when the trace unit is enabled on page 3-100*.
- *Trace unit behavior when the trace unit is disabled on page 3-100*.
- *Trace unit behavior on a Warm reset and a Cold reset on page 3-101*.
- *Trace unit behavior on a trace unit powerdown on page 3-101*.
- *Trace unit behavior on a PE powerdown on page 3-104*.
- *Trace unit behavior on a PE low-power state on page 3-105*.
- *Trace unit behavior while the PE is in Debug state on page 3-106*.
- *Trace unit behavior on a trace buffer overflow on page 3-106*.
- *Trace unit behavior on a trace flush on page 3-108*.
- *Trace unit behavior when tracing is prohibited on page 3-108*.

3.5.1 Trace unit behavior on a trace unit reset

A trace unit has two resets:

- A *trace unit reset*, that resets all trace unit trace registers and some trace unit management registers.
- An *external trace reset*, that resets some trace unit management registers.

A trace unit reset is applied when the trace unit core power domain is powered up. An external trace reset is applied when the trace unit debug power domain is powered up. In addition, if the system has a mechanism to initiate a reset of the trace unit on demand, that is, when the trace unit is already powered up, then one or both of these resets might be asserted.

When the Unified Power Domain Model is implemented, an external trace reset occurs whenever a trace unit reset occurs, and the external trace reset does not occur independently of the trace unit reset.

In the Unified Power Domain Model there is only a single reset for the trace unit.

This section is organized into the following subsections:

- *Behavior on a trace unit reset*
- *Behavior on an external trace reset on page 3-99*
- *Values of trace unit registers after reset on page 3-99*.

Behavior on a trace unit reset

A trace unit reset does the following:

- Resets all trace unit registers that are in the trace unit core power domain. These registers are:
  - All trace unit trace registers.
  - Two management registers, which are the OS Lock registers, TRCOSLAR and TRCOSLSR.

  The values of the registers are reset to the values given in Table 3-1 on page 3-99.
- Locks the OS Lock to disable accesses to the trace registers from an external debugger. This can be checked by reading either:
  - The TRCOSLSR to see if TRCOSLSR.OSLK is set to 1.
  - The TRCPDSR to see if TRCPDSR.OSLK is set to 1.

In addition, a trace unit reset might reset TRCITCTRL.IME to 0. TRCITCTRL.IME controls whether the trace unit is in integration mode. It is IMPLEMENTATION DEFINED whether a trace unit reset or an external trace reset resets TRCITCTRL.IME, but one of these resets must reset TRCITCTRL.IME.

Behavior on an external trace reset

An external trace reset does the following:

- Resets all trace registers that are in the trace unit debug power domain. These registers are:
  - All trace unit management registers except the OS Lock registers, TRCOSLAR and TRCOSLSR.

  The values of the registers are reset to those given in Table 3-2 and the associated text.
- Locks the Software Lock, if the Software Lock is implemented. This can be checked by reading TRCLSR.SLK to see if it has the value 1. See also TRCLAR.

In addition, an external trace unit reset might reset TRCITCTRL.IME to 0. TRCITCTRL.IME controls whether the trace unit is in integration mode. It is IMPLEMENTATION DEFINED whether a trace unit reset or an external trace reset resets TRCITCTRL.IME.

Values of trace unit registers after reset

Table 3-1 shows the trace unit registers and fields that are:

- Reset to zero after a trace unit reset.
- Reset to one after a trace unit reset.

All other registers that are reset by a trace unit reset are reset to an UNKNOWN value.

Table 3-1 Reset values for trace unit registers and fields after a trace unit reset

Register or field that is reset	Reset value
TRCPRGCTLR.EN	`0b0`
TRCPROCSELR.PROCSEL	`0b00000`
TRCAUXCTLR	`0b0`
TRCIMSPEC0.EN	zero
TRCOSLSR.OSLK (if implemented)	`0b1`
TRCITCTRL.IME[a] (if implemented)	`0b0`

a. It is IMPLEMENTATION DEFINED whether this field is reset by a trace unit reset or by an external trace reset.

Table 3-2 shows the trace unit registers and fields that are:

- Reset to zero after an external trace reset.
- Reset to one after an external trace reset.

All other registers that are reset by an external trace reset are reset to an UNKNOWN value.

Table 3-2 Reset values for trace unit registers and fields after an external trace reset

Register or field that is reset	Reset value
TRCPDCR.PU	`0b0`
TRCITCTRL.IME (if implemented)[a]	`0b0`
TRCLSR.SLK (if implemented)	`0b1`

a. It is IMPLEMENTATION DEFINED whether this field is reset by a trace unit reset or by an external trace reset.

3.5.2 Trace unit behavior when the trace unit is enabled

The trace unit is enabled when both of the following are true:

- The main enable bit, TRCPRGCTLR.EN, is set to 1.
- The OS Lock is unlocked, that is, TRCPDSR.OSLK and TRCOSLSR.OSLK are both zero.

For an Armv7-M PE, the control bit DEMCR.TRCENA can be used to control whether the trace unit is enabled. This behavior is IMPLEMENTATION DEFINED. For an Armv8-M PE, DEMCR.TRCENA does not affect the trace unit.

When the trace unit is enabled, it means that the trace unit is enabled to generate trace, and that all trace unit resources are enabled.

When enabled, it is required that all PE execution can be traced, except when:

- A trace buffer overflow occurs.
- The authentication interface prevents the tracing of certain pieces of code.

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.

When the trace unit is enabled:

- Writes to most trace unit trace registers might be ignored. It is UNKNOWN whether writes to these registers succeed. If the writes are successful, the behavior of the trace unit is UNPREDICTABLE. 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. The information in *Access permissions on page 7-340* details the registers that might ignore writes while the trace unit is enabled or not 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 shown in the TRCCNTVRn.
  - The sequencer state. This is shown in the TRCSEQSTR.
  - The ViewInst start/stop control. This is shown in the TRCVICTLR.
  - The single-shot comparator control status. This is shown in the TRCSSCSRn.

Enabling 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 logic. Before enabling the trace unit, these resources must be explicitly programmed to give them a state to start from.

3.5.3 Trace unit behavior when the trace unit is disabled

This means that the trace unit is not enabled to generate trace, and that all trace unit resources are disabled.

The trace unit is disabled when either of the following are true:

- The main enable bit, TRCPRGCTLR.EN, is set to 0.
- The OS Lock is locked, that is, TRCPDSR.OSLK and TRCOSLSR.OSLK both have the value 1.

For Armv7-M PEs, the control bit DEMCR.TRCENA can be used to control whether the trace unit is enabled. This behavior is IMPLEMENTATION DEFINED. For Armv8-M PEs, DEMCR.TRCENA does not affect the trace unit.

Note

The OS Lock is automatically locked as a result of a trace unit reset.

On disabling the trace unit:

- The trace unit stops generating trace, and empties the trace buffers by outputting any data in them.
- When the trace buffers are empty, and after the trace unit has become idle, TRCSTATR.IDLE indicates that the trace unit is idle.

- All resources that are visible in the programmers' model retain their values, and become stable at those values. When these resources are stable, TRCSTATR.PMSTABLE indicates that the programmers' model is stable. For more information, see *About the behavior of events on disabling the trace unit on page 4-178*.

If the trace unit has generated any event trace, that event trace must be output before TRCSTATR.IDLE indicates that the trace unit is idle.

When the trace unit is disabled:

- No trace is generated.
- All trace unit resources and events are disabled.
- Event tracing is disabled.
- All external outputs, as defined in *External outputs on page 4-146*, are forced low.
- TRCSTATR.IDLE indicates that the trace unit is idle.
- TRCSTATR.PMSTABLE indicates that the programmers' model is stable. All the resources that are visible in the programmers' model retain their values from when the trace unit was last enabled. These are:
  - The counter values. These are shown in the TRCCNTVRn.
  - The sequencer state. This is shown in the TRCSEQSTR.
  - The ViewInst start/stop control. This is shown in the TRCVICTLR.
  - The single-shot comparator control status. This is shown in the TRCSSCSRn.

3.5.4 Trace unit behavior on a Warm reset and a Cold reset

A PE Warm reset does not cause a Trace unit reset or an external trace reset. This ensures that tracing is possible through a PE Warm reset. A PE Warm reset might occur at the same time as a Trace unit reset or an external trace reset, however, these are asynchronous and unrelated events.

A Trace unit reset does not cause any PE resets, although a PE Cold reset might also involve asserting the Trace unit reset. It is IMPLEMENTATION DEFINED whether a PE Cold reset causes a Trace unit reset.

3.5.5 Trace unit behavior on a trace unit powerdown

As described in *Trace unit power domains on page 3-91*, an implementation of an ETMv4 trace unit has two power domains that can be independently powered down:

- A core power domain.
- A debug power domain.

How these power domains are connected in a system is defined by the system designer. For example, the power domains might be connected as shown in Figure 3-2 on page 3-92 or as shown in Figure 3-3 on page 3-93. Alternatively, a trace unit might be implemented in a single power domain system, so that there is no split in functionality between the two power domains.

If a trace unit is implemented in multiple power domains

If a trace unit is implemented with separate core and debug power domains, the power domains might be connected as shown in Figure 3-2 on page 3-92 or as shown in Figure 3-3 on page 3-93. In this case, if the trace unit core power domain is powered down but the system debug power domain remains powered up:

- The trace unit cannot be accessed by using system instructions. Some management registers can be accessed by using the memory-mapped interface or an external debugger. See *Access permissions on page 7-340*.
- The trace unit trace registers are inaccessible and return an error response.
- TRCPDSR.POWER indicates that the trace unit has no core power.
- TRCPDSR.STICKYPD might indicate that core power has been removed. See *TRCPDSR, PowerDown Status Register on page 7-392* for more information.
- The status of TRCOSLSR.OSLK is UNKNOWN.
- The status of TRCPDSR.OSLK is UNKNOWN.

When the trace unit core power domain is powered down, setting TRCPDCR.PU to 1 requests core power to be restored. The TRCPDCR is a management register, so it is accessible by either:

- An external debugger.
- Memory-mapped access. However, when using memory-mapped access, writes to the TRCPDCR are ignored whenever the Software Lock is implemented and locked.

The status of TRCPDCR.PU is also exported from the trace unit as a signal to a power controller. However, if the PE core power domain is connected to the trace unit power domain, and the PE debug power domain is connected to the trace unit power domain, then this signal can be combined with a signal from the PE, as shown in Figure 3-4.

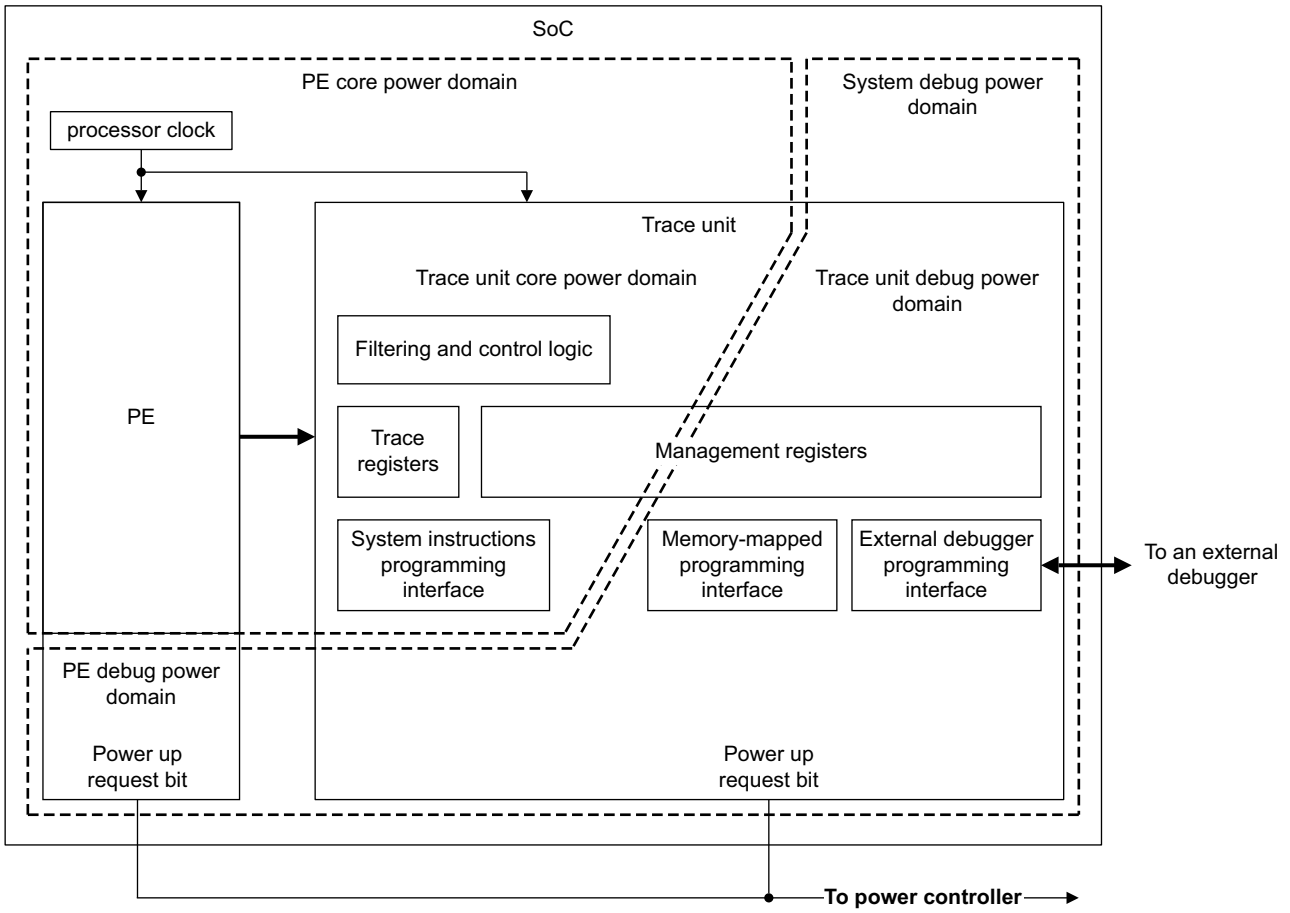


Figure 3-4 A combined powerup request signal

If the power domains are connected as shown in Figure 3-2 on page 3-92 or as shown in Figure 3-3 on page 3-93, and if the system debug power domain is powered down but the trace unit core power domain remains powered up:

- The trace unit cannot be accessed by using the memory-mapped interface or by using an external debugger. However, if the PE is powered up, all trace registers, and some management registers, can be accessed by using system instructions. See *Access permissions on page 7-340*.

Note

Control of the debug power domain is a system issue that is not covered by this specification.

If a trace unit is implemented in a single power domain

If the power domains are connected as shown in Figure 3-5 on page 3-103, the behavior of the trace unit is more straightforward, because it is either completely powered down or completely powered up.

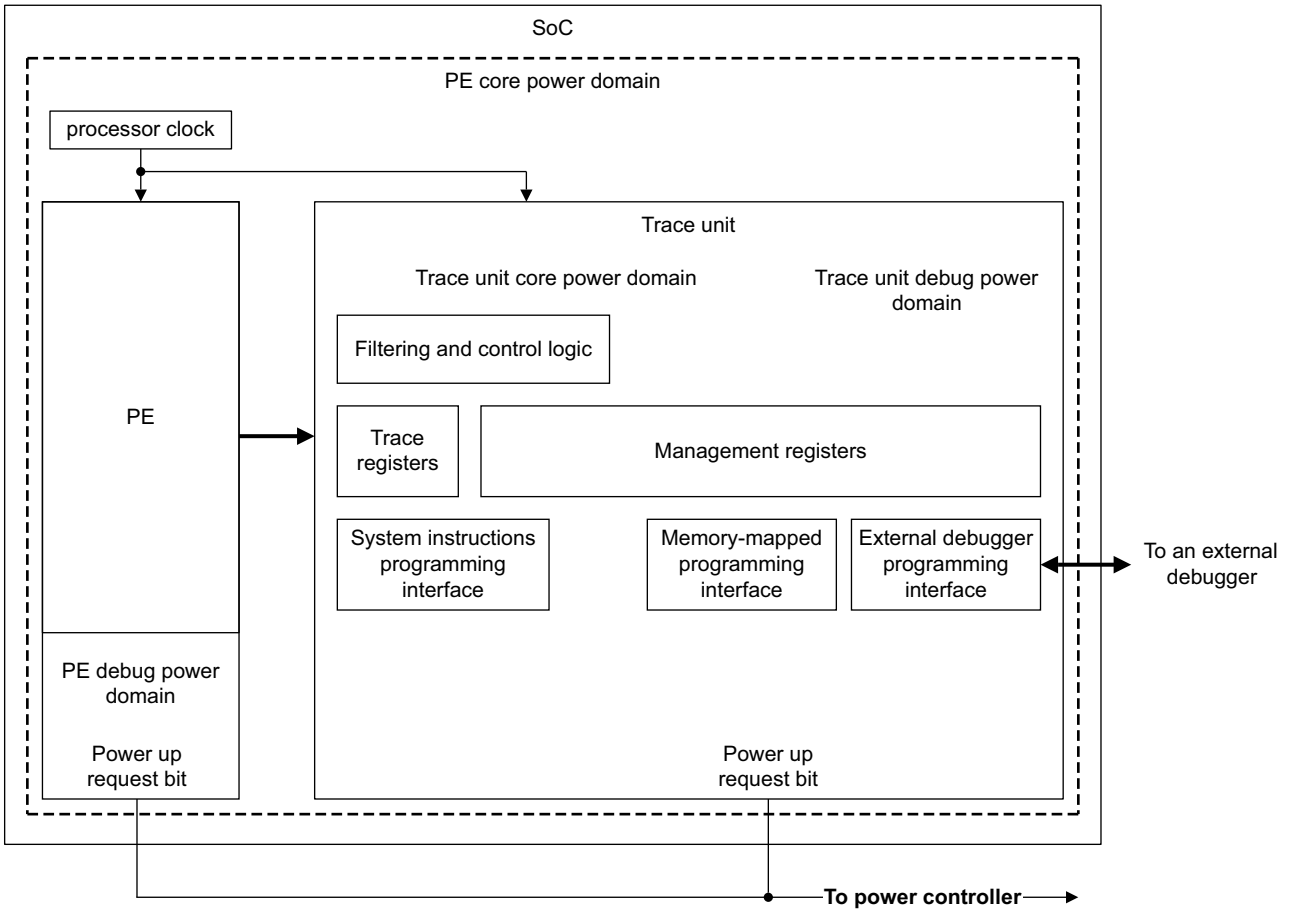


Figure 3-5 A combined powerup request signal in a single power domain

If a trace unit is implemented in a unified power domain

When the Unified Power Domain Model is implemented, if the trace unit core power domain can be powered down, the system must implement an external debug component with a Power-up request mechanism which can request the trace unit core power domain to be powered up.

The Power-up request mechanism must be 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.

Note

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.

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 1.
- The POWERID field in the parent entry has an IMPLEMENTATION DEFINED value.

It is IMPLEMENTATION DEFINED whether the trace unit core power domain is the PE Core power domain or some other power domain. See *Arm®v8-A Architecture Reference Manual* for more information.

For more information on the CoreSight Class 0x9 ROM Table, see *Arm® CoreSight™ Architecture Specification*. Other standards might be added and permitted by Arm Limited from time-to-time.

When the Unified Power Domain Model is implemented, if the Power-up request mechanism is not implemented, the trace unit core power domain and the ROM Table are in the same power-domain.

When the Unified Power Domain Model is implemented, all registers behave as Error when the trace unit core power domain is powered down.

Note

There is no identification mechanism to determine if the Unified Power Domain Model feature is implemented.

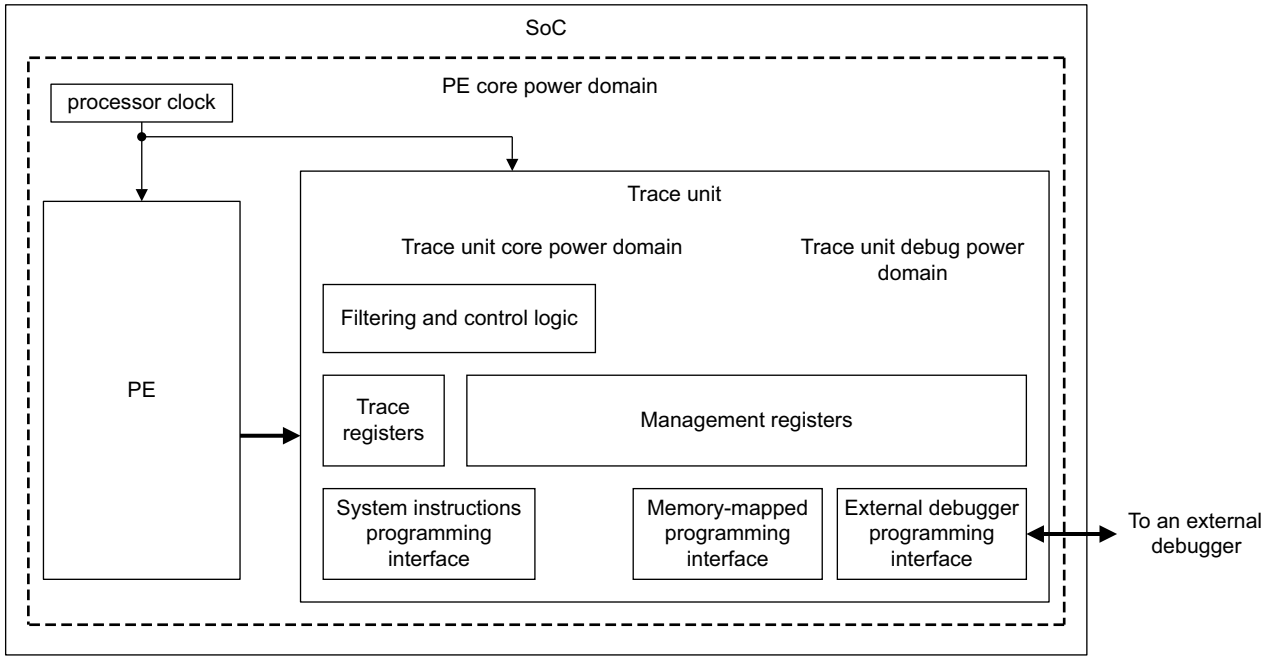


Figure 3-6 A Unified Power Domain Model trace unit

3.5.6 Trace unit behavior on a PE powerdown

How the trace unit behaves when the PE is powered down depends on how the trace unit is implemented in the system.

If the system is a single power domain system, then the trace unit is powered down when the PE and the rest of the system is powered down, and the trace unit state is lost. In this configuration, it is still possible to save trace unit state before powerdown and restore trace unit state after powerup.

If the system is a multiple power domain system, then:

- If the trace unit core power domain is connected to the PE core power domain, as shown in Figure 3-2 on page 3-92, and there is a separate debug power domain that remains powered when the PE core power domain is powered down, then there is the option to save the trace unit state so that it can be restored when the trace unit is powered up again. See *Saving and restoring the trace unit state on page 3-94*.
- If the trace unit core power domain is separate from the PE core power domain, and PE is powered down but the trace unit remains powered up, then the trace unit state is preserved. The option to save the trace unit state for later restoration remains available.

3.5.7 Trace unit behavior on a PE low-power state

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` or a `WFE` instruction. In these cases, it might be advantageous if the trace unit also enters a low-power state. It is IMPLEMENTATION DEFINED whether a trace unit supports a low-power state or not. If the trace unit supports a low-power state, the trace unit might also enter a low-power state when the PE enters Debug state.

If the trace unit supports a low-power state:

- If the trace unit is enabled, and it enters a low-power state, then it appears enabled throughout the time it is in the low-power state.
- All trace that is generated before entering the low-power state must be output before entering the low-power state.
- Events that are in transition through the trace unit must not be lost when entering or leaving low-power state. However, observation of these events might not occur until after the trace unit leaves the low-power state.
- If the `WFI` or `WFE` instructions are classified as branch instructions, and the trace unit enters a 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 `WFI` or `WFE` instruction.
  - Any pending Commit elements.

  For details about configuring `WFI` or `WFE` instructions to be classified as branch instructions, see the field description of TRCIDR2.WFXMODE in *TRCIDR2, ID Register 2 on page 7-374*.

While the trace unit is in a low-power state:

- No trace is generated, including event trace.
- It is IMPLEMENTATION DEFINED whether the cycle counter continues to count or not.
- All trace unit resources remain in the state that they were in before entry to low-power state. This includes the counters, the sequencer, the ViewInst start/stop control, and the single-shot comparator controls.
- All external outputs, as defined in *External outputs on page 4-146*, are forced low.
- Accesses to trace unit trace registers and trace unit management registers are unaffected.
- The trace unit might not recognize external events, such as the assertion of any external inputs.
- Timestamp requests might be ignored.

When the trace unit enters a low-power state, it must automatically perform a trace flush. See *Trace unit behavior on a trace flush on page 3-108* for more details of the flush operation.

It is possible that the trace unit might intermittently leave and reenter 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.

There is no additional requirement for the trace unit to generate a Trace Info or Trace On element when the PE leaves 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 *Trace unit behavior while the PE is in Debug state*.

If the trace unit enters a low-power state, then if any trace synchronization requests occur while the trace unit is in low-power state, the trace unit must handle those requests correctly. See *Synchronization with a trace analyzer on page 2-65* for information on how the trace unit handles trace synchronization requests.

The trace unit can be programmed so that it does not enter a low-power state. In this case, the trace unit resources continue operating and the trace unit can generate trace. This option is enabled by setting TRCEVENTCTL1R.LPOVERRIDE to 1.

A low-power state includes the following scenarios:

- Where the trace unit clock is stopped.
- Where the trace unit core power domain enters a retention state where power is removed but the state of the trace unit is preserved throughout the retention state.
- The PE enters Debug state.

While in a retention state, accesses to the trace unit behave as if in the no core power state. See *Access permissions on page 7-340* for more details on access permissions when in the no core power state.

Note

TRCEVENTCTL1R.LPOVERRIDE does not affect the operation of the PE. In particular, 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 trace.

TRCIDR5.LPOVERRIDE indicates if the implementation supports overriding the low-power state. If it does, then TRCEVENTCTL1R.LPOVERRIDE is a RW field.

3.5.8 Trace unit behavior while the PE is in Debug state

When the PE is in Debug state, ViewInst is inactive. The trace unit does not trace instructions that are executed while the PE is in Debug state.

On entry to Debug state:

- If ViewInst is active, the trace unit generates an *Exception element* to show that the PE has entered Debug state.
- A trace flush is requested. See *Trace unit behavior on a trace flush on page 3-108*.
- ViewInst becomes inactive.

On exit from Debug state:

- If ViewInst becomes active again, the trace unit generates a Trace On element.

An entry to Debug state does not affect the trace unit ViewInst start/stop control. This resource maintains its state while the PE is in Debug state.

The trace unit might not trace an exception if the exception occurs between the PE exiting Debug state and it executing the first instruction. This exception is only traceable if the preceding instruction or exception is traced, but on exit from Debug state there is no preceding instruction or exception.

3.5.9 Trace unit behavior on a trace buffer overflow

As Figure 3-1 on page 3-90 shows, there might be two trace buffers in an ETMv4 trace unit:

- An instruction trace buffer.
- A data trace buffer.

On an instruction trace buffer overflow:

- Instruction tracing becomes inactive until the trace unit can recover from the overflow. Arm recommends that data tracing also becomes inactive. This is to prevent data trace packets from being output when there are no instruction trace packets to associate those data trace packets with.
- The trace unit must not output a partial instruction trace packet. Only complete packets are permitted.
- The trace unit outputs an Overflow instruction trace packet after all the other packets in the instruction trace buffer have been output. This indicates to a trace analyzer that a trace unit instruction trace buffer overflow has occurred. See *Overflow instruction trace element on page 5-191* and *Overflow instruction trace packet on page 6-258*.

On a data trace buffer overflow:

- Data tracing becomes inactive until the trace unit can recover from the overflow. Instruction tracing can continue, because program flow information can still be obtained. Only information about data transfers cannot be obtained if the data trace buffer overflows.
- The trace unit must not output a partial data trace packet. Only complete packets are permitted.
- The trace unit outputs an Overflow data trace packet after all the other packets in the data trace buffer have been output. This indicates to a trace analyzer that a trace unit data trace buffer overflow has occurred. See *Overflow data trace element on page 5-226* and *Overflow data trace packet on page 6-311*.

Typically, the trace unit recovers from a buffer overflow by draining the buffer that has overflowed, so that the packets that it contains are output, and then restarting trace.

If any numbered trace unit events occur while recovering from a trace unit buffer overflow, that is, if any trace unit events occur that would normally result in the generation of Event elements, then:

- For those numbered events that have occurred, at least one Event element must be generated.
- The Event elements must be generated after the trace unit has recovered from a buffer overflow, but before an Overflow element is generated.

Whenever the trace unit recovers from a trace buffer overflow, trace synchronization is automatically requested so that a trace analyzer can resynchronize with the trace streams. Whenever trace synchronization is requested, it is requested in both trace streams. See *Synchronization with a trace analyzer on page 2-65*. This means that:

- If the instruction trace buffer overflows, then after the trace unit recovers:
  - The first packets output in the instruction trace stream must be an A-Sync packet followed by a Trace Info packet.

  The trace unit can output Event, Overflow, Discard, or Ignore packets before it outputs A-Sync packets, but this is not the recommended behavior.
- If the data trace buffer overflows, regardless of whether instruction tracing continues, then after the trace unit recovers:
  - The first packets output in the data trace stream must be an A-Sync packet followed by a Trace Info packet.
  - An A-Sync packet, followed by a Trace Info packet, are output in the instruction trace stream.

  The trace unit can output Event, Overflow, Discard, or Ignore packets before it outputs A-Sync packets, but this is not the recommended behavior.

Following the recovery from an overflow of either trace buffer, and after synchronization packets have been output, Data Sync Mark packets must also be output in both trace streams if data tracing is enabled. This is to enable resynchronization of the data trace stream with the instruction trace stream.

A trace unit might include an optional feature to prevent overflows. TRCSTALLCTLR.NOOVERFLOW controls this feature. Enabling the feature might cause a significant performance impact. This feature prevents overflows if the number or frequency of Event elements is below an IMPLEMENTATION DEFINED threshold. The threshold must be at least one of each Event number, for each time the trace unit is enabled.

3.5.10 Trace unit behavior on a trace flush

Some conditions might cause the trace unit to flush out all the trace it has generated. These are:

- When the trace unit transitions from an enabled to a disabled state, as dictated by the programmers' model.
- If the trace capture infrastructure requests a trace flush, for example if a trace flush is requested on an Arm AMBA™ ATB.
- Before the trace unit enters either:
  - A low-power state.
  - A powerdown state.
- Entry to Debug state.

When a flush is requested, the trace unit must perform the following tasks before responding to the flush request:

1. Encode any remaining elements into trace packets, for example, there might be some Commit elements that are not yet encoded.
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.

When a trace 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. In these cases, in addition to the tasks the trace unit must perform before responding to the flush request, the trace unit must also stop generating trace before responding to the flush request, or before indicating that the trace unit is idle.

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.

When a trace flush is requested, the trace data must be output within a finite period. If the cause of the flush request requires an acknowledgement, such as the flush request mechanism on AMBA ATB, then the acknowledgement must also occur within a finite period.

3.5.11 Trace unit behavior when tracing is prohibited

An executable program might contain regions of code that it is 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 therefore be prohibited while the PE is in certain modes. For example:

- Non-invasive debug, including trace, might be prohibited when the PE is in a Secure-privileged mode.
- Non-invasive debug, including trace, might be prohibited while the PE is in any Secure state.

Trace might also become prohibited if, while tracing program execution, the permitted level of non-invasive debug changes. For example, if trace is permitted and active while the PE is operating in a Secure state, and then the permitted level of non-invasive debug changes from being permitted for a PE Secure state, to not permitted, then trace becomes prohibited.

For an Armv8-M PE, the PE might enter a prohibited region without taking a branch or an exception. In such cases, the trace unit stops generating trace at the target of the previous P0 element, because there is no new P0 element to indicate execution up to the transition into the prohibited region.

An ETMv4 trace unit implements the authentication interface, as defined by the *CoreSight Architecture Specification*, to define whether non-invasive debug is permitted.

In an executable program, where tracing certain regions of code is prohibited, these regions of code are called *prohibited regions*. The following sections use this term to refer to these regions:

- *Behavior when in a prohibited region on page 3-109*.
- *Behavior when non-invasive debug is not permitted on page 3-110*.

- *Behavior when the permitted level of non-invasive debug changes on page 3-110*.
- *Behavior when DBGACK is forced HIGH on page 3-110*.

Note

Accesses to the trace unit registers are not affected by either:

- Being in a prohibited region.
- Being in a state where non-invasive debug is not permitted.

Behavior when in a prohibited region

If trace is active when the PE enters a prohibited region, then trace becomes inactive. When the PE is executing code from a prohibited region, the trace unit behaves as follows:

- No instructions are traced until the PE leaves the prohibited region.
- No exceptions are traced, including PE reset.
- Data transfers caused by instructions that are in the prohibited region are not traced.
- No instruction address, data address, or data value comparators match on any instruction that is in the prohibited region.
- Data transfers caused by instructions that are outside a prohibited region are traced if required, that is, if any filtering applied indicates that those data transfers are traced. This is true even if those data transfers occur at the time when the PE is executing code from a prohibited region.
- If cycle counting is enabled, the cycle counter continues to count. When tracing restarts, cycles spent in the prohibited region are included in the cycle count.
- Context ID comparators are disabled.
- Virtual context identifier comparators are disabled.
- Event tracing is unaffected.
- The ViewInst start/stop control retains its state.
- Other resources, such as the counters, sequencer, and external outputs, behave as normal.

When the PE is executing code from a prohibited region, the trace unit must not output any information about that execution. The trace unit must not output any trace that might indicate something about the execution of the prohibited region, such as the context or any instruction or data addresses, including the address of the first instruction in the prohibited region.

The most common cause of an entry into a prohibited region is an exception. Whenever an exception causes an entry to a prohibited region, the trace unit generates an *Exception element* that indicates the exception type. For more information about *Exception elements*, see *Exception instruction trace element on page 5-196*.

If there are any speculative P0 elements remaining when the PE enters a prohibited region, the trace unit must generate the appropriate Commit or Cancel elements when the resolution of those speculative elements is known.

When the PE leaves a prohibited region, tracing restarts if trace is active, that is, if any filtering applied dictates that trace is active. In this case, the trace unit must generate a Trace On element to indicate to the trace analyzer that there has been a discontinuity in the trace stream.

If the PE leaves a prohibited region other than when a Context synchronization event occurs, the prohibited region is permitted to extend up to the next Context synchronization event. Typically, a PE leaves a prohibited region by an exception return, but a PE might leave a prohibited region when the authentication interface changes, or when moving from Secure to Non-secure state without an exception return.

The trace unit might not trace an exception if the exception occurs between the PE exiting a prohibited region and it executing the first instruction. This exception is only traceable if the preceding instruction or exception is traced, but on exit from a prohibited region there is no preceding instruction or exception.

Behavior when non-invasive debug is not permitted

If all non-invasive debug is not permitted, the following additional conditions apply when the PE is executing code from a prohibited region:

- The trace unit must flush out all the trace that it has generated.
- No new trace is generated, including event trace.
- All external outputs, as defined in *External outputs on page 4-146*, are forced LOW.
- All resources retain their state. This includes the counters, the sequencer, the ViewInst start/stop control, and the single-shot comparators.
- It is IMPLEMENTATION DEFINED whether the cycle counter continues to count.

Where the PE implements Armv8.4-Trace, the trace unit behaves as if non-invasive debug is always permitted.

Behavior when the permitted level of non-invasive debug changes

If the trace unit is enabled and active, and then the permitted level of non-invasive debug changes so that trace becomes inactive, some speculative P0 elements might remain. These elements might not be resolved in the trace stream. In this case, the trace unit generates a discard element to indicate that the speculative elements must be discarded because their status cannot be resolved.

Note

If a change of the authentication interface causes an entry to or an exit from a prohibited region, the exact time when that entry or exit happens is IMPLEMENTATION SPECIFIC. This means that there might be a delay between the time when the change of authentication interface takes place and the time when the entry or exit happens.

Therefore:

- If non-invasive debug changes to not permitted, an entry to a prohibited region might occur shortly after the change of the authentication interface.
- If non-invasive debug changes to permitted, an exit from a prohibited region might occur shortly after the change of the authentication interface. In this case, there is a possibility that trace might not start until after the next Context synchronization event, and therefore, some required instructions might not be traced.

Behavior when DBGACK is forced HIGH

DBGACK is a signal which indicates to the system that the PE is halted in Debug state. Some PEs permit the signal DBGACK to be forced HIGH. This is controlled by the debugger software. If DBGACK is forced HIGH while the PE is not in Debug state, one of the following occurs:

- DBGACK has no effect on the tracing of PE execution, and tracing continues as it would in Non-debug state.
- When DBGACK is HIGH, tracing becomes inactive, similar to the PE being in Debug state. The trace unit is not required to trace an exception that indicates entry into Debug state.

Arm recommends that DBGACK is not forced HIGH while tracing is enabled.

3.5.12 Trace Unit behavior on changes in Exception level or Security state

If the PE changes Exception level, or changes from Secure to Non-secure state other than by an exception return, the trace unit might not observe the change in Exception level or Security state on the exact instruction where it changes in the PE.

Note

Changing from Secure to Non-secure state other than by an exception return is deprecated.

If this change in Exception level or Security state occurs, the trace unit might observe the old or new Exception level or Security state at any point between the target of the P0 instruction before the operation that caused the change, up to the Context synchronization event after the operation that caused the change. This might affect the Exception level or Security state value traced and the operation of the comparators and ViewInst filtering logic which might observe either the old or new Exception level or Security state.

3.5.13 SG instruction

For Armv8-M, when branching from Non-secure state to an SG instruction in Secure state, the SG instruction is executed in Non-secure state. If debug of Secure state is prohibited:

- It is IMPLEMENTATION DEFINED whether a comparator matches on the execution of the SG instruction.
- The SG instruction is not traced, because the subsequent P0 instruction is always in Secure state and is not traced.

See *T32 instruction set on page F-480* for information about the handling of the SG instruction in an IT block.

3.5.14 Trace unit behavior for a multi-threaded processor

For a processor with multiple threads, or PEs, a trace unit might be provided for each thread.

The processor might support enabling and disabling of these threads, either at reset time or dynamically. The trace unit 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.

For threads that are permanently disabled, Arm recommends that the trace unit is not visible. Arm recommends that the trace units for these threads are either not included or are marked as not present in any ROM tables that describe the system.

3.5.15 Sharing the trace unit between multiple PEs

Where PEs do not implement Armv8.4-Trace, a trace unit might be shared between multiple PEs. When sharing between multiple PEs, the trace unit only traces one PE at a time. TRCPROCSELR controls which PE is selected for tracing.

When switching between PEs, the trace unit must be disabled with TRCPRGCTLR.EN set to zero. When the value in TRCPROCSELR is changed, the trace unit core power domain might become powered down, due to the change in the selected PE. After a change in TRCPROCSELR, the debugger must read TRCPDSR.POWER to determine the status of the trace unit core power domain. If the trace unit core power domain is powered down after a switch to a new PE, this power status must be immediately shown in TRCPDSR.POWER. After a switch to a new PE, if TRCPDSR.POWER is zero, power to the trace unit core power domain can be requested by setting TRCPDCR.PU high. The debugger must then wait for TRCPDSR.POWER to become high.

If during the switch to a new PE, the state of the trace registers is lost, TRCPDSR.STICKYPD must be set high, and the debugger must then reprogram the state of the trace registers before enabling the trace unit. The value of TRCPROCSELR must not be lost or reset during a switch to a new PE, to ensure the new PE is continually selected. However, the debugger must be aware that TRCPROCSELR might become inaccessible if the trace unit core power domain turns off during the switch to the new PE, until TRCPDSR.POWER indicates the power has been restored.

Chapter 4
Programming the Trace Unit

This chapter describes different aspects of programming the trace unit. It contains the following sections:

4.1 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 full instruction and data trace. A third application might require tracing of a specific program function or a particular set of data transfers.

The ETMv4 architecture provides for each of these usage models. An ETMv4 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.

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 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.

An ETMv4 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 ETMv4 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.
- Continuous instruction tracing plus continuous data tracing, where both trace streams are output. This mode can only be used if data tracing is implemented.

Continuous instruction tracing can also be used with data-based filtering, if data tracing is implemented.

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.

Data-based filtering

This model can only be used if data tracing is implemented. When data-based filtering is applied, data tracing is active only for certain data transfers. This is useful when, for example, tracing of all accesses to a particular peripheral is required. When using this model, one of the following must also be applied:

- Continuous instruction tracing.
- Instruction-based filtering.

This is because tracing of the parent instruction is required for the tracing of a data transfer.

If data-based filtering is used with instruction-based filtering, that instruction-based filtering must permit tracing of the parent instructions for the required data transfers.

Table 4-1 summarizes each of the basic filtering models.

Table 4-1 Summary of the filtering that can be applied

Filtering model	Possible modes	Is instruction tracing filtered?	Is data tracing filtered if implemented and enabled?
Continuous tracing	Continuous instruction tracing only	N	Not implemented, or implemented but not enabled.
	Continuous instruction tracing plus continuous data tracing[a]	N	N
	Continuous instruction tracing plus data-based filtering[a]	N	Y
Instruction-based filtering	Instruction-based filtering only	Y	Not implemented, or implemented but not enabled.
	Instruction-based filtering plus data-based filtering[a]	Y	Y
Data-based filtering	Data-based filtering plus continuous instruction tracing[a]	See the *Continuous instruction tracing plus data-based filtering* mode in this table.	
	Data-based filtering plus instruction-based filtering[a]	See the *Instruction-based filtering plus data-based filtering* mode in this table.	

a. These can only be used if data tracing is implemented.

The remainder of this section is organized as follows:

- *Trace options*.
- *The continuous tracing model on page 4-116*.
- *The instruction-based filtering model on page 4-118*.
- *The data-based filtering model on page 4-131*.

4.1.1 Trace options

For all the possible filtering modes, the trace unit can be programmed before a trace run to enable various options, including:

- Context ID tracing, if implemented, to indicate to a trace analyzer the Context ID value.
- *Virtual context identifier* tracing, if implemented, to distinguish between different virtual machines.
- Cycle counting, if implemented, 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 branch targets to be traced with an explicit target address.
- Conditional instruction tracing, if implemented, to enable a trace analyzer to determine the conditional status of all conditional non-branch instructions.
- If data tracing is implemented and enabled, tracing either or both:
  - The data addresses of the data transfers.
  - The data values of the data transfers.
- If data tracing is implemented and enabled, tracing either or both:
  - Load instructions explicitly.

— Store instructions explicitly.

Note

Whether a configurable option is implemented can be determined from the trace ID registers in the programmers' model. For more information, see *Optional features on page 2-80*.

4.1.2 The continuous tracing model

The following sections describe:

- *About continuous instruction tracing*.
- *About continuous instruction tracing plus data tracing*.

About continuous instruction tracing

Continuous instruction tracing provides program flow operation, indicating all instructions that are executed by the PE plus all exceptions.

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, data tracing might be either:

- Not implemented.
- Implemented and enabled, and in this case, data tracing might be either:
  - Continuous.
  - Filtered.

See Table 4-1 on page 4-115.

When a trace unit is programmed for continuous instruction tracing, ViewInst is always active during a trace run.

The trace unit can be programmed before a trace run to generate the trace in the form that is required, for example with any of the options that are listed in *Trace options on page 4-115*.

To get full program flow coverage:

- Enable conditional instruction tracing if implemented. This enables a trace analyzer to determine the conditional status of all conditional non-branch instructions. For more information, see:
  - *Trace behavior on tracing conditional instructions on page 2-71*.
  - *Conditional instructions tracing on page 2-84*.

About continuous instruction tracing plus data tracing

There are two possible modes that can be used:

Continuous instruction tracing plus continuous data tracing

This provides full program flow operation, indicating all instructions that are executed by the PE and all data transfers initiated by the PE, plus all exceptions. In this mode, both ViewInst and ViewData are always active during a trace run.

Continuous instruction tracing plus data-based filtering

This provides program flow operation, indicating all instructions that are executed by the PE and all exceptions, plus some of the data transfers initiated by the PE depending on the data-based filtering applied. In this mode:

- ViewInst is always active during a trace run.
- ViewData is active for part of a trace run.

The additional options are:

- In the instruction trace stream:
  - Enable conditional instruction tracing if implemented, to enable a trace analyzer to determine the conditional status of all conditional non-branch instructions. Enabling this option gives full program flow coverage.
  - Trace data load instructions explicitly, data store instructions explicitly, or trace both explicitly. See *Explicit tracing of data load and store instructions on page 2-84*.
- In the data trace stream, trace either:
  - The data addresses of data loads.
  - The data values of data loads.
  - The data addresses of data stores.
  - The data values of data stores.
  - A combination of the above.

Note

In these modes, because data tracing is implemented and enabled, at least one of the following must be programmed:

- Explicit tracing of load instructions.
- Explicit tracing of store instructions.

Which of these is programmed depends on whether the requirement is to trace data transfers associated with load instructions, or data transfers associated with store instructions. If the requirement is to trace both, then both must be enabled.

In addition, the trace unit can be programmed before a trace run to generate trace in the form that is required, for example with any of the options listed in *Trace options on page 4-115*.

Table 4-2 shows some example scenarios for data tracing, with possible usage models for these scenarios. For more information, see *Data address tracing on page 2-84* and *Data value tracing on page 2-84*.

Table 4-2 Example scenarios for data tracing

Possible scenarios[a]					
Loads[b]		Stores[b]			
DA[c]	DV[d]	DA[c]	DV[d]	Usage model	Notes
N	N	N	N	Basic instruction tracing, that gives basic program flow.	To obtain full program flow coverage, enable conditional instruction tracing. See *Conditional instructions tracing on page 2-84*.
N	N	N	Y	Variable tracking.	Tracing of conditional store instructions is required for this model.
N	N	Y	Y	Instruction tracing including store instructions. Data tracing of all store data transfers.	Tracing of conditional store instructions is required for this model.
N	Y	N	N	Register reconstruction.	Tracing of conditional load instructions is required for this model. See *Conditional instructions tracing on page 2-84*.

Table 4-2 Example scenarios for data tracing (continued)

Possible scenarios[a]					
Loads[b]		Stores[b]			
DA[c]	DV[d]	DA[c]	DV[d]	Usage model	Notes
Y	Y	N	N	Register reconstruction with stack visibility.	Tracing of conditional load instructions is required for this model. See *Conditional instructions tracing on page 2-84*.
Y	N	Y	N	Cache analysis.	-
Y	Y	Y	Y	Full instruction and data tracing, that is, full visibility debug.	Tracing of conditional load and store instructions is required for this model. See *Conditional instructions tracing on page 2-84*.

a. The purpose of this table is to show a few examples. Therefore, not all possible scenarios are shown. Any scenarios that are not shown are not prohibited.

b. To enable the trace unit to trace data transfers that are associated with load instructions, loads instructions must be traced explicitly.
To enable the trace unit to trace data transfers that are associated with store instructions, store instructions must be traced explicitly.
See *Explicit tracing of data load and store instructions on page 2-84*.

c. DA = Data Address. This is the address of the data transfer. This can be enabled by setting TRCCONFIGR.DA to 1.

d. DV = Data Value. This is the data value of the data transfer. This can be enabled by setting TRCCONFIGR.DV to 1.

4.1.3 The instruction-based filtering model

If tracing of a particular process or function that the PE carries out is required, or if tracing of a particular thread of execution is required, instruction-based filtering can be applied to the instruction trace, that is, instruction tracing can be made active or inactive based on instruction addresses. The ViewInst function provides this functionality.

By using the ViewInst function, it is possible to:

- Trace a particular piece of code by starting trace on one instruction address and then stopping trace on another instruction address.
- Include instruction address ranges in the trace while excluding other instruction address ranges.
- Start and stop tracing based on an imprecise enabling event.
- Prevent instructions from being traced if they are executed in particular Exception levels.

When instruction-based filtering is applied to obtain a particular piece of code, and if data tracing is implemented and enabled, the trace unit can also trace any data transfers that are associated with that piece of code, depending on how ViewData is programmed.

The remainder of this section is organized as follows:

- *Overview of the ViewInst function on page 4-119*.
- *Behavior when the ViewInst function is imprecise on page 4-120*.
- *Tracing one or more processes or threads of execution by using the ViewInst start/stop control on page 4-120*.
- *Tracing regions of code while omitting other regions of code by using the ViewInst include/exclude control on page 4-124*.
- *Filtering instruction tracing by using the enabling event on page 4-126*.
- *Examples of combining the ViewInst filtering controls, or of using only one or two on page 4-126*.
- *Guidelines for interpreting the ViewInst function result on page 4-128*.

- *Rules for tracing exceptions on page 4-129*.

Overview of the ViewInst function

Figure 4-1 shows a functional overview of the ViewInst function.

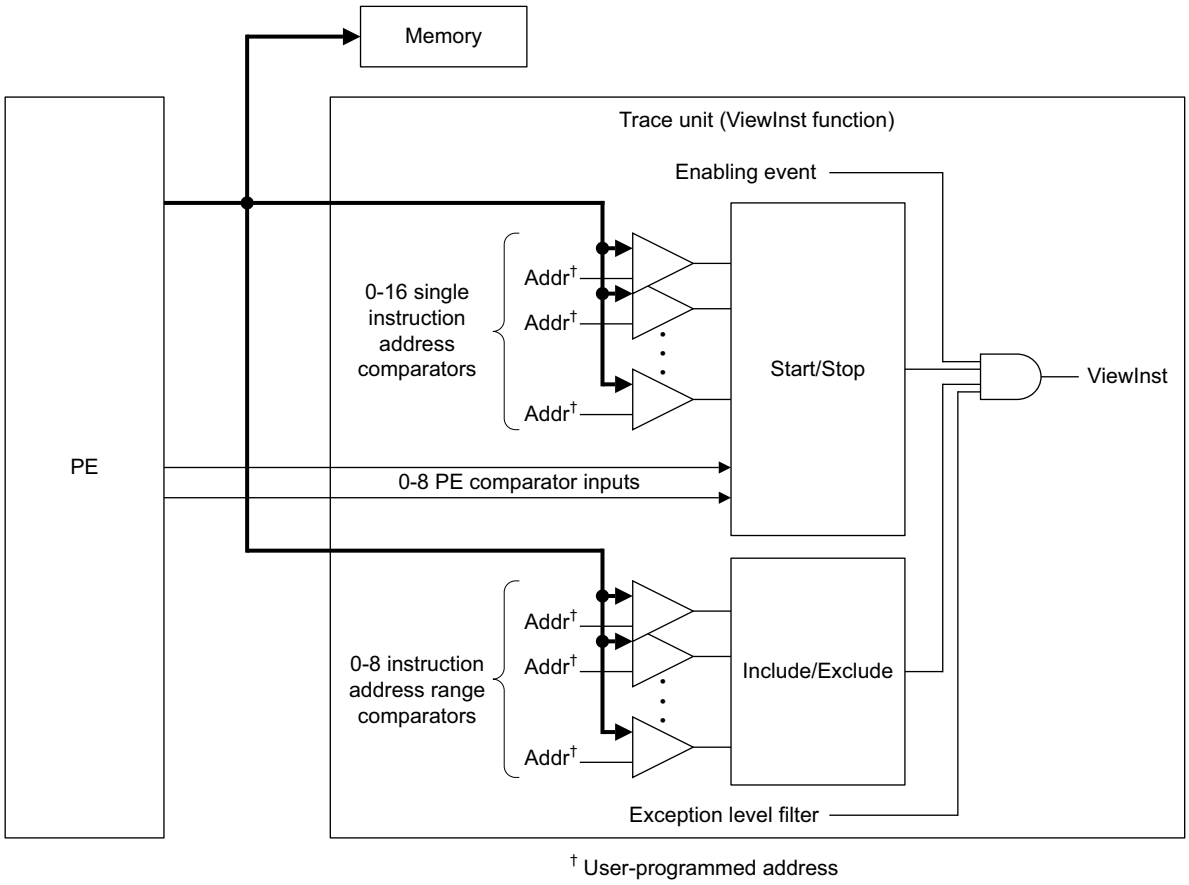


Figure 4-1 The ViewInst function

The ViewInst function has the following capabilities:

- A start/stop control that is based on single instruction address comparators. This enables one or more pairs of addresses to be chosen, each pair consisting of:
  - A start instruction address.
  - A stop instruction address.

  The trace unit traces the piece of code between the start and stop addresses, including the instructions at the start and stop addresses.

  The start/stop control can also operate using PE comparator inputs to the trace unit.

- An include/exclude control that is based on instruction address range comparators. In this case, the addresses that are specified at the inputs of the comparators either:
  - Always have the instructions at those addresses traced.
  - Never have the instructions at those addresses traced.

  Whether the instructions are included or excluded depends on whether the address range comparators are programmed for the *include function* or the *exclude function*.

- An imprecise enabling event input. This input can be programmed to any external resource, or any trace unit resource, by using the TRCVICTLR.

- An Exception level filter. This filter can be programmed to filter trace based on the PE Exception level, preventing instructions that are executed in the specified Exception levels from being traced. The Exception level filter can be programmed separately for Secure and Non-secure states, by using the TRCVICTLR.EXLEVEL_S and TRCVICTLR.EXLEVEL_NS fields.

Programming the ViewInst logic to be sensitive to comparators that are not programmed for instruction address comparisons results in CONSTRAINED UNPREDICTABLE behavior of the ViewInst logic, where ViewInst might or might not be asserted. Such programming includes:

- Selecting an address comparator for the ViewInst include/exclude control or Start/Stop control where the comparator is not programmed for instruction address comparisons.
- Selecting a PE comparator input for the ViewInst Start/Stop control where the PE comparator is not programmed for instruction address comparisons.

Behavior when the ViewInst function is imprecise

Except for some IMPLEMENTATION DEFINED scenarios, ViewInst is imprecise if the enabling event input is set to anything other than continuously TRUE.

If the ViewInst function is imprecise, then the following might occur:

- Tracing might not turn on in time to trace the required instructions.
- Tracing might not turn off in time. This means that some instructions that are not required might be traced.
- The data for an instruction might not be traced.
- Some trace might be missing at the start or end of a region of code.
- Extra trace might be present at the start or end of a region of code.

Tracing one or more processes or threads of execution by using the ViewInst start/stop control

This is useful when the requirement is to trace a particular piece of code and all the functions that the piece of code calls.

Tracing starts on one instruction address and stops on another instruction address.

The start/stop control has several inputs, so that more than one pair of start and stop instruction addresses can be chosen. See Figure 4-1 on page 4-119.

When tracing starts, the instruction at the start address is traced, then tracing is active up to and including the instruction at the stop address, as shown in Figure 4-2.

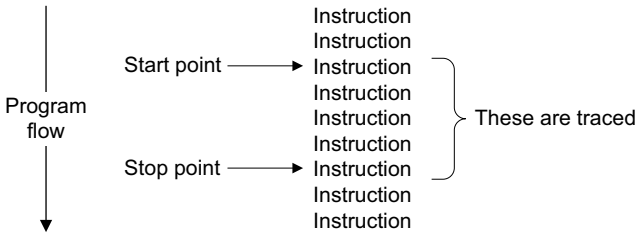


Figure 4-2 Instructions traced include the instructions at the start and stop addresses

Instruction tracing can also be started and stopped by using PE comparator inputs to the trace unit. See *PE comparator inputs on page 4-159*.

The following have no effect on the start/stop control:

- Exceptions
- An entry to Debug state
- A trace unit buffer overflow.

On disabling the trace unit, the logic in the start/stop control becomes static and retains its state until the trace unit is enabled again. See *Trace unit behavior when the trace unit is disabled on page 3-100*. However, if required, the state of the start/stop logic can be changed while the trace unit is disabled.

Note

The start/stop control must be programmed with an initial state when the trace unit is programmed before a trace run.

If an implementation makes speculation visible to the trace unit, the start/stop logic must behave as if no speculation has occurred. That is, if a start point or a stop point occurs speculatively and is then canceled, the start/stop logic must behave as if the start point or stop point did not occur.

If incorrect speculation cancels all execution from the point at which the trace unit becomes enabled, the start/stop logic must behave as if none of the incorrect speculative execution has occurred. This would result in the start/stop logic using the initial value programmed while the trace unit was disabled.

Behavior of the start/stop control during a trace run

During a trace run, the PE trace is represented using trace elements. The ETMv4 architecture defines the generation of trace elements from the execution of the PE. Of all the types of instructions that the PE executes:

- Trace elements are only generated for certain types. These elements are always called P0 elements. For a list of what types of instructions are traced as P0 elements, see *About instruction trace P0 elements on page 2-35*.
- All other instruction types are inferred from the P0 elements.

When a trace analyzer receives a P0 element that represents a particular type of instruction, it infers other instructions from that P0 element. This means that PE execution is indicated as either batches of instructions, or blocks of instructions:

- A block of instructions that starts on an instruction immediately after an instruction that is traced as a P0 element and ends at the next instruction that is traced as a P0 element, as shown in Figure 4-3.

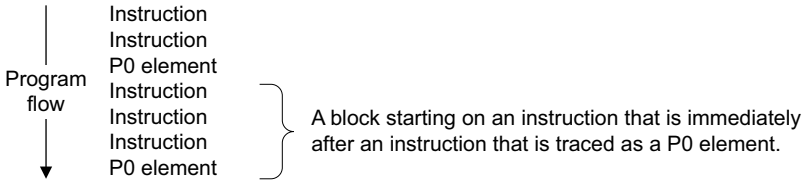


Figure 4-3 A block of instructions

- A batch of instructions that starts with an instruction that is not immediately after a P0 element and ends at the next instruction that is traced as a P0 element, as shown in Figure 4-4.

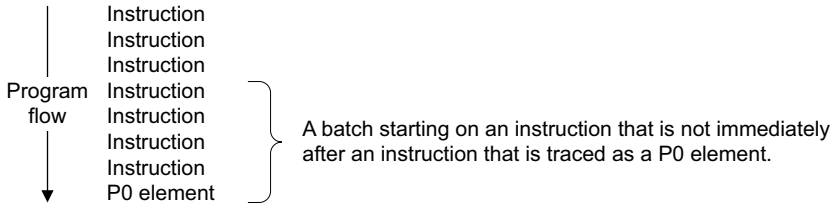


Figure 4-4 A batch of instructions

This means that tracing can start on any instruction, regardless of whether it is immediately after a P0 element instruction.

Note

In addition to the instruction types that are traced as P0 elements, exceptions are traced as P0 elements, and if the PE is an Armv6-M, Armv7-M, or Armv8-M PE, exceptions returns are also traced as P0 elements. See Figure 2-3 on page 2-34.

The behavior of the start/stop control during a trace run is as follows:

- If a batch of instructions contains a start point, then the control is active for the whole of that batch.
- If a batch of instructions contains a stop point, and the start/stop control is already active, then:
  - The control is active for the whole of that batch.
  - The control is inactive for the next batch of instructions, unless the next batch contains a new start point.
- If a batch, block, or sequential block of instructions contains both a start point and a stop point, then the behavior of the control is IMPLEMENTATION DEFINED, and might any of:
  - The control obeys the order of the start and stop points in the batch, block, or sequential block.
  - The control ignores the order of the start and stop points in the batch, block, or sequential block, and instead is active for the whole batch, block, or sequential block. The control is inactive after the last instruction in the batch, block, or sequential block. That is, the order is always considered to be start and then stop regardless of the actual order.

Arm recommends that the start/stop control is not programmed with any start points after any stop points within any potential sequential block of instructions.

Figure 4-5 on page 4-123 shows an example of the behavior of the start/stop control for a piece of code comprising eight blocks of instructions.

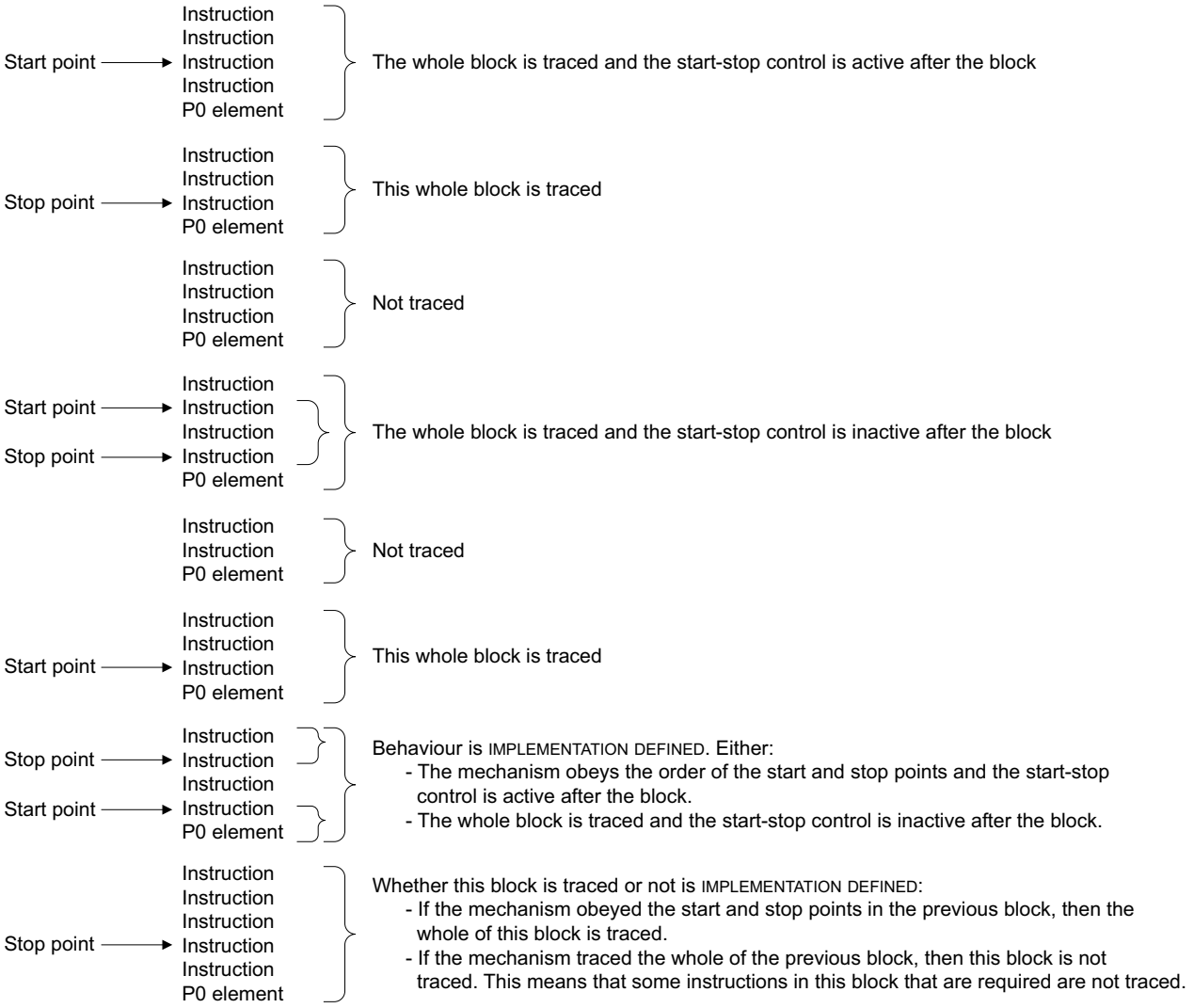


Figure 4-5 Behavior of the ViewInst start/stop control

Two of the blocks that are shown in Figure 4-5 have both a start and a stop point in them. If the implementation ignores the order of the start and stop points, so that the whole block is traced, then:

- The stop point for a piece of code that is to be traced must not be in the same block of instructions as the start point for the next piece of code that is to be traced.

If this rule is not obeyed, part of the second piece of code is not traced. This scenario is shown in the bottom two blocks of instructions in Figure 4-5. If the trace unit is exposed to speculative PE execution, the start/stop control must be tolerant of speculative execution and must only represent the behavior of architecturally-executed instructions. That is, if a start or stop point is executed speculatively and then canceled, the start/stop control state must be restored to the state that it was in before the canceled instructions.

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 start/stop control:

- The start/stop logic is either active or inactive for the instruction at that address.
- The start/stop control is either active or inactive after that instruction.

When using PE comparator inputs to control the start/stop logic, Arm strongly recommends that only PE comparators programmed for instruction address comparisons are used. See *PE comparator inputs on page 4-159*.

For a trace unit for an Armv6-M, Armv7-M, and Armv8-M PE, the following rules apply to the start/stop logic in response to a partially executed instruction that is interrupted and later continued:

- If a start point is set on the instruction, the start/stop logic becomes active on each attempt to execute any part of the instruction.
- If a stop point is set on the instruction, the start/stop logic only becomes inactive when the instruction fully completes.
- If an instruction is canceled then the Start-Stop state must roll back to its state prior to that instruction.
- If any beat of an instruction is canceled then the Start-Stop state must roll back to its state prior to that beat.

Tracing regions of code while omitting other regions of code by using the ViewInst include/exclude control

This mode is useful if:

- Specific ranges of instructions are to be included in the trace.
- Specific ranges of instructions are to be excluded from the trace.
- A combination of the two is required.

The include/exclude control has two functions:

Include function Includes one or more instruction address ranges.

Exclude function Excludes one or more instruction address ranges.

There are between zero and eight instruction address range comparators available for this control. See Figure 4-1 on page 4-119. Some of these comparators can be selected for the include function, and some for the exclude function. The include comparators can be programmed with the instruction address ranges that are to be included. Similarly, the exclude comparators can be programmed with instruction address ranges that are to be excluded.

For example, if all instructions in the address range from `0x0` to `0x2C` are required, but no other instructions are required, a comparator can be selected to be an include comparator and then can be programmed with these two addresses. All instructions that are in this address range, including those at the start and end addresses, are traced.

The include/exclude control differs from the start/stop control in the following ways:

- When the start/stop control 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 include/exclude control 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 start/stop control, as program execution continues through the address range towards the end address, no functions that the piece of code calls are traced.

Instruction address range comparators

An instruction address range comparator is a combination of two single address comparators, where one comparator is programmed with the range start address, and the other is programmed with the range end address, as shown in Figure 4-6 on page 4-125.

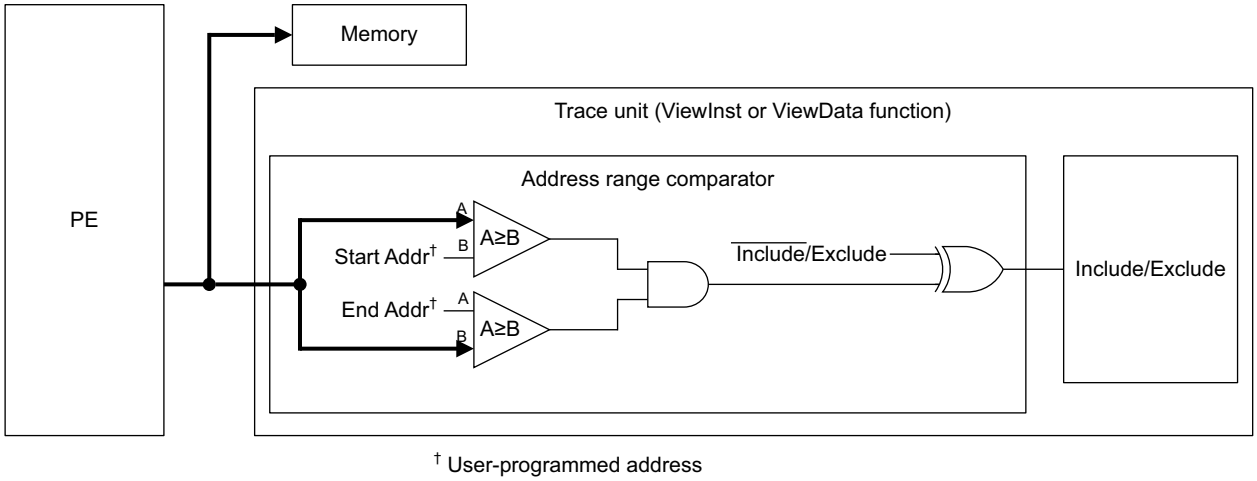


Figure 4-6 Functional overview of an address range comparator

An instruction address range comparator matches if the instruction that the PE accesses is in the following range:

`(access address ≥ start address) AND (access address ≤ end address)`

The range might be an include range or an exclude range, depending on whether the comparator is selected to be an include or an exclude comparator.

Instruction tracing rules when using the ViewInst include/exclude control

An instruction must be traced if:

- Its address matches an address in an include range.

It is expected that an instruction is not traced if:

- Its address matches an address in an exclude range.

However, exclude addresses take priority over include ranges. This enables an exclude range to be located within a larger include range.

In addition:

- If the include/exclude control indicates that an instruction must be traced:
  - The instruction might not be traced if any of the other ViewInst filtering controls, for example the enabling event or start/stop or control, indicate that the instruction is not traced.
- If the include/exclude control indicates that an instruction is not traced:
  - It is expected that the instruction is not traced, regardless of what any of the other ViewInst filtering controls indicate.

Note

- There are occasions when tracing an instruction at an exclude address is permitted. *Guidelines for interpreting the ViewInst function result on page 4-128* describes these occasions.
- The default operation of the trace unit when no include ranges are specified, is to trace the whole of memory.

Table 4-3 shows the usage models of the ViewInst include/exclude control.

Table 4-3 Summary of the ViewInst include/exclude control

Comparators selected for:		
Include	**Exclude**	**Usage model**
N	N	Trace all instructions.
N	Y	Trace all instructions except those in the excluded instruction address ranges.
Y	N	Trace only those instructions that are in the included instruction address ranges.
Y	Y	Trace those instructions that are in the included instruction address ranges, except for those that are in the excluded instruction address ranges. This enables an exclude range to be located within a larger include range.

Table 4-3 shows that:

- When no include address ranges are specified, this indicates to the trace unit that the whole of memory must be included, so that all instructions are traced.
- The exclude function takes priority over the include function.

Since only certain instructions are classified as P0 element instructions, execution is indicated as the execution of a block of instructions from the target of one P0 element instruction up to and including the next P0 element instruction or exception. If a block of instructions is not covered by at least one individual exclude range, it is IMPLEMENTATION SPECIFIC whether the block is excluded or not, even if other exclude ranges cover the rest of the block.

Filtering instruction tracing by using the enabling event

Instruction tracing can also be filtered imprecisely by using the enabling event input to the ViewInst function. See Figure 4-1 on page 4-119. This input can be programmed to either an external input or any resource available in the trace unit, for example the sequencer.

The enabling event input is only sampled on cycles where instructions are processed by the trace unit. It is ignored on other cycles.

The enabling event input can be programmed to use a resource by using TRCVICTLR.EVENT. For more information, see *Selecting trace unit resources on page 4-171*.

Note

The enabling event control is imprecise. This means that if, for example, the enabling event input is programmed to use an address range comparator resource, the following scenario might occur:

- The PE performs some execution that triggers the address range comparator into becoming active.
- Some time passes.
- The enabling event is asserted.

Therefore, there might be a delay between when the address range comparator becomes active, and the time when the enabling event input is asserted. This time delay is IMPLEMENTATION DEFINED and might not be fixed for an implementation.

Examples of combining the ViewInst filtering controls, or of using only one or two

The ViewInst function contains the following filtering controls:

- A start/stop control.
- An include/exclude control.

- An imprecise enabling event input.
- An Exception level filter.

Any combination of these can be used in a trace run. However, as indicated by the AND gate in Figure 4-1 on page 4-119, an instruction is only traced if all the controls indicate that the instruction must be traced. If any one of the controls indicates that the instruction is not to be traced, the instruction is not traced.

This means that if a trace run requires the use of only one of the controls, the others must be disabled so that they always indicate that all instructions must be traced. Each control is disabled as follows:

- To disable the start/stop control, do not program any start or stop points. This indicates to the trace unit that all instructions must be traced. In addition, use TRCVICTLR.SSSTATUS to set the state of the start/stop logic to started.
- To disable the include/exclude control, do not program any include or exclude address ranges. This indicates to the trace unit that all instructions must be traced.
- To disable the enabling event input, program the input to be always active.
- To disable the Exception level filter, program all the implemented bits of the EXLEVEL_S and EXLEVEL_NS fields in TRCVICTLR to 0.

The sections that follow contain some examples:

- *Using only the start/stop control example*.
- *Using only the include/exclude control example*.
- *Using only the enabling event input example*.
- *Using a combination of the start/stop and include/exclude controls example on page 4-128*.

Using only the start/stop control example

This can be used to trace a specific function and all functions that it calls. Do the following:

1. Disable the include/exclude control by not programming it with any address ranges. This means that the include/exclude control indicates that all instructions must be traced.
2. Program the enabling event to be always active.
3. Program the start/stop control to start tracing on the entry point of the function and stop tracing on the exit points of the function.

Using only the include/exclude control example

This can be used to trace a specific function but not any functions that it calls. Do the following:

1. Disable the start/stop control by not programming it with any start or stop points. This means that the start/stop control indicates that all instructions must be traced. In addition, use TRCVICTLR.SSSTATUS to set the state of the start/stop logic to started.
2. Program the enabling event to be always active.
3. Program the include/exclude control to include a function that the code calls:
   a. Select an instruction address range comparator to be an include comparator.
   b. Program it with the instruction address range of the function to be traced.

Using only the enabling event input example

This can be used to filter the trace that is based on an external input or any resource available in the trace unit, for example, on the value of one or more trace unit counters.

1. Disable the start/stop control by not programming it with any start or stop points. This means that the start/stop control indicates that all instructions must be traced. In addition, use TRCVICTLR.SSSTATUS to set the state of the start/stop logic to started.

2. Disable the include/exclude control by not programming it with any address ranges. This means that the include/exclude control indicates that all instructions must be traced.

3. Program the enabling event to the required resource or resources.

Using a combination of the start/stop and include/exclude controls example

This can be used to trace a specific function and all functions it calls, except for some known library functions.

1. Program the enabling event to be always active.

2. Program the start/stop control to start tracing on the entry point of the function and stop tracing on the exit points of the function.

3. Program the include/exclude control:

   a. Select one or more instruction address range comparators to be exclude comparators. How many you select depends on how many library functions you want to exclude.

   b. Program each comparator with the instruction address range of a library function that you do not want to trace.

Guidelines for interpreting the ViewInst function result

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.

When ViewInst transitions from low to high

- If execution occurs while ViewInst is low, it is permitted for a trace unit to trace instructions in certain circumstances. See *Occasions when tracing instructions when ViewInst is low is permitted on page 4-129*.

- If tracing of instructions is permitted while ViewInst is low, but no instructions or exceptions that occur are traced, then this means that there is a discontinuity in the trace. In this case, on ViewInst becoming high, a Trace On element must be generated. For more information, see *Trace On instruction trace element on page 5-190*.

- Any instructions that are executed while ViewInst is high must be traced.

When ViewInst transitions from high to low

- Any instructions that are executed while ViewInst is high must be traced.

- 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. See *About instruction trace P0 elements on page 2-35*. This means that the following scenario is possible:

  1. 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.

  2. ViewInst then goes low.

  3. 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.

- There is no requirement for the target address of a branch or exception to be traced if ViewInst has transitioned to a low state by the time program execution has moved to the target.

Occasions when tracing instructions when ViewInst is low is permitted

These occasions are typically:

- When the instruction that ViewInst indicates is not to be traced is in the same block of instructions 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 block of instructions, or from the earliest instruction that must be traced up to the end of that block of instructions.
- When the instruction that ViewInst indicates is not to be traced is in a block of instructions that precedes or follows a block of instructions containing an instruction that ViewInst indicates must be traced.

  An implementation always traces the block of instructions that contains the instruction that must be traced. However, additional blocks of instructions might be traced. This is more likely to occur when many instructions are executed in close proximity. Figure 4-7 shows this:

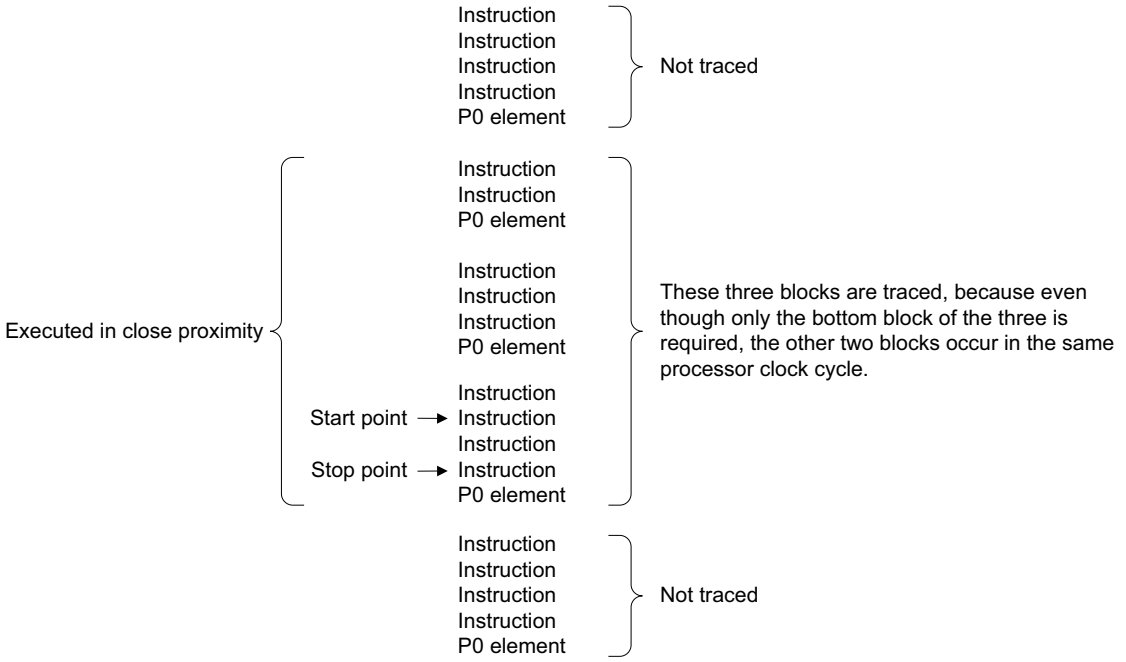


Figure 4-7 An example of when tracing instructions when ViewInst is low is permitted

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. An implementation must avoid any unnecessary or excessive tracing because it can affect the efficiency of the trace and might mean that the quantity of trace that is generated exceeds the available bandwidth of the trace port.

Rules for tracing exceptions

None of the comparators that are used in the ViewInst function are affected when the PE takes an exception.

Note

- When the PE takes an exception, this act in itself is not an execution of instructions, and therefore has no impact on the comparators.
- When an exception is traced, it might imply execution of instructions up to a specified address. These implied instructions might have an impact on the comparators, but the exception itself does not.

Exceptions do not cause the instruction address comparators, that are used for the start/stop control, or the instruction address range comparators, that are used for the include/exclude control, to match. This means that when an exception occurs, the ViewInst function does not indicate that it must be traced.

However, it is useful to trace exceptions, to determine why execution has departed from the normal program flow.

Therefore, if an exception occurs, it must be traced if the instruction or exception immediately before it was traced so that the reason for departing from normal program flow can be found.

An exception might not be traced if it occurs before the first instruction that follows any of the following incidents:

- The trace unit is enabled.
- Exit from Debug state.
- Exit from a prohibited region.

For examples of tracing exceptions, see *Examples of basic program trace when exceptions occur on page A-425*.

If a trace buffer overflow occurs and an exception is the first item after recovery from the overflow, it is IMPLEMENTATION DEFINED whether the exception is traced, even if the last instruction or exception before the overflow was traced.

When tracing speculative execution

If the PE is speculatively executing instructions, then:

- If an exception occurs, it is traced if ViewInst indicates that the preceding instruction or exception must be traced.

If ViewInst indicates that the preceding instruction or exception must be traced, but then that instruction or exception is canceled, the exception is traced if the most recent instruction or exception that has not been canceled must be traced.

If the preceding instruction or exception is traced, and then speculative execution causes tracing to become inactive (for example, the start/stop control might pass its stop point because of the speculative execution), then tracing must become active again to trace the exception.

Forcing tracing of exceptions

The trace unit can be programmed so that it always traces certain exceptions, regardless of whether the instruction or exception immediately before the exception must be traced.

This option is enabled by setting either or both:

- TRCVICTLR.TRCERR to 1. This forces the trace unit to trace all System error exceptions.
- TRCVICTLR.TRCRESET to 1. This forces the trace unit to trace all PE Reset exceptions.

TRCIDR3.TRCERR indicates if support for tracing all System error exceptions is implemented. If it is implemented, then TRCVICTLR.TRCERR is a RW field.

TRCVICTLR.TRCRESET is always a RW field. Forcing tracing of exceptions is not active when executing in a prohibited region. In a prohibited region, the effective value of TRCVICTLR.TRCERR is zero, and the effective value of TRCVICTLR.TRCRESET is zero.

An address is always included with the *Exception element* to indicate the preferred exception return address of the exception. If tracing is only active for the exception, a Trace On element must be output before the *Exception element* to indicate that tracing has become active, and the Trace On element must be followed by an Address element to indicate where tracing becomes active. This Address can be output in one of the following ways:

- An Address packet is output after the Trace On packet and contains the preferred exception return address of the exception. The Exception packet then implies no instruction execution because the Address packet and the Exception packet contain the same address.
- No Address packet is output, and the Exception packet uses the [E1:E0]=b10 encoding which implies an Address element before the Exception. This also implies no instruction execution. See *Exception instruction trace packet on page 6-262* for more details on the Exception packet.

- An Address packet is output after the Trace On packet which contains an address which is not the preferred exception return address. This implies some instruction execution before the exception. The address in the Address packet might be any instruction address from the target of the previous P0 element up to the preferred exception return address. This is permitted behavior as defined in *Occasions when tracing instructions when ViewInst is low is permitted on page 4-129*.

When a System error exception occurs and is traced only because TRCVICTLR.TRCERR==1, if a second exception occurs immediately afterwards then it is IMPLEMENTATION DEFINED whether the second exception is traced. If the System error exception was traced because the execution immediately before the System error was traced, then the second exception is always traced.

When a Reset exception occurs and is traced only because TRCVICTLR.TRCRESET==1, if a second exception occurs immediately afterwards then it is IMPLEMENTATION DEFINED whether the second exception is traced. If the Reset exception was traced because the execution immediately before the Reset execution was traced, then the second exception is always traced.

If a System error exception occurs immediately after exit from a prohibited region, and TRCVICTLR.TRCERR is set to 1, it is IMPLEMENTATION SPECIFIC whether the exception is traced. If the exception is traced, the preferred exception return address must not include information about the prohibited region.

If a PE Reset exception occurs immediately after exit from a prohibited region, and TRCVICTLR.TRCRESET is set to 1, it is IMPLEMENTATION SPECIFIC whether the exception is traced. If the exception is traced, the preferred exception return address must not include information about the prohibited region.

For an Armv7-A, Armv7-R, Armv8-A, Armv8-R, or Armv8-M PE, a System error or PE Reset exception only occurs immediately following a prohibited region if the authentication interface changes dynamically, and this change enables tracing following the prohibited region or enables non-invasive debug. These changes are asynchronous to program execution, and therefore the preferred exception return address that is included with the exception might provide information about where the exception was taken from, or the exception return address might be UNKNOWN.

On Armv6-M, Armv7-M and Armv8-M PEs that support data trace, if a System error or Reset exception occurs and is traced because TRCVICTLR.TRCERR==1 or TRCVICTLR.TRCRESET==1 then the stack push data for these exceptions might not be traced.

4.1.4 The data-based filtering model

If the tracing of certain data transfers is required, for example data transfers associated with a certain range of data addresses or a particular peripheral, or if the data transfers that are carried out for a particular function that the PE executes are required, data-based filtering can be applied to the data trace. That is, data tracing can be made active or inactive based on either data addresses or instruction addresses. The ViewData function provides this functionality.

By using the ViewData function, it is possible to:

- Trace the data transfers for instruction ranges by including the data transfers for those instruction ranges in the trace while excluding other data transfers.
- Include data address ranges in the trace while excluding other data address ranges, or include single data addresses in the trace while excluding other single data addresses.
- Start and stop tracing based on an enabling event.

When using this filtering model, either:

- Continuous instruction tracing must be applied.
- Instruction-based filtering must be applied and the ViewInst filtering function must be programmed to enable tracing of the parent instructions for the data transfers that are required.

This is because tracing of the parent instruction is required for tracing of a data transfer. See *Relationships between P0, P1, and P2 elements on page 2-38*.

The remainder of this section is organized as follows:

- *Overview of the ViewData function*.
- *Tracing data transfers associated with specific instructions by including and excluding instruction address ranges on page 4-133*.
- *Tracing data transfers by including and excluding data addresses or data address ranges on page 4-134*.
- *Tracing data transfers by using the enabling event on page 4-134*.
- *Combining the ViewData filtering controls, or using less than three in a trace run on page 4-134*.
- *Tracing of stack transfers on Armv6-M, Armv7-M, and Armv8-M PEs on page 4-136*.
- *Flexibility in the interpretation of the ViewData function result on page 4-137*.
- *Rules for tracing data addresses on page 4-137*.
- *Rules for tracing data values on page 4-138*.

Overview of the ViewData function

Figure 4-8 shows a functional overview of the ViewData function.

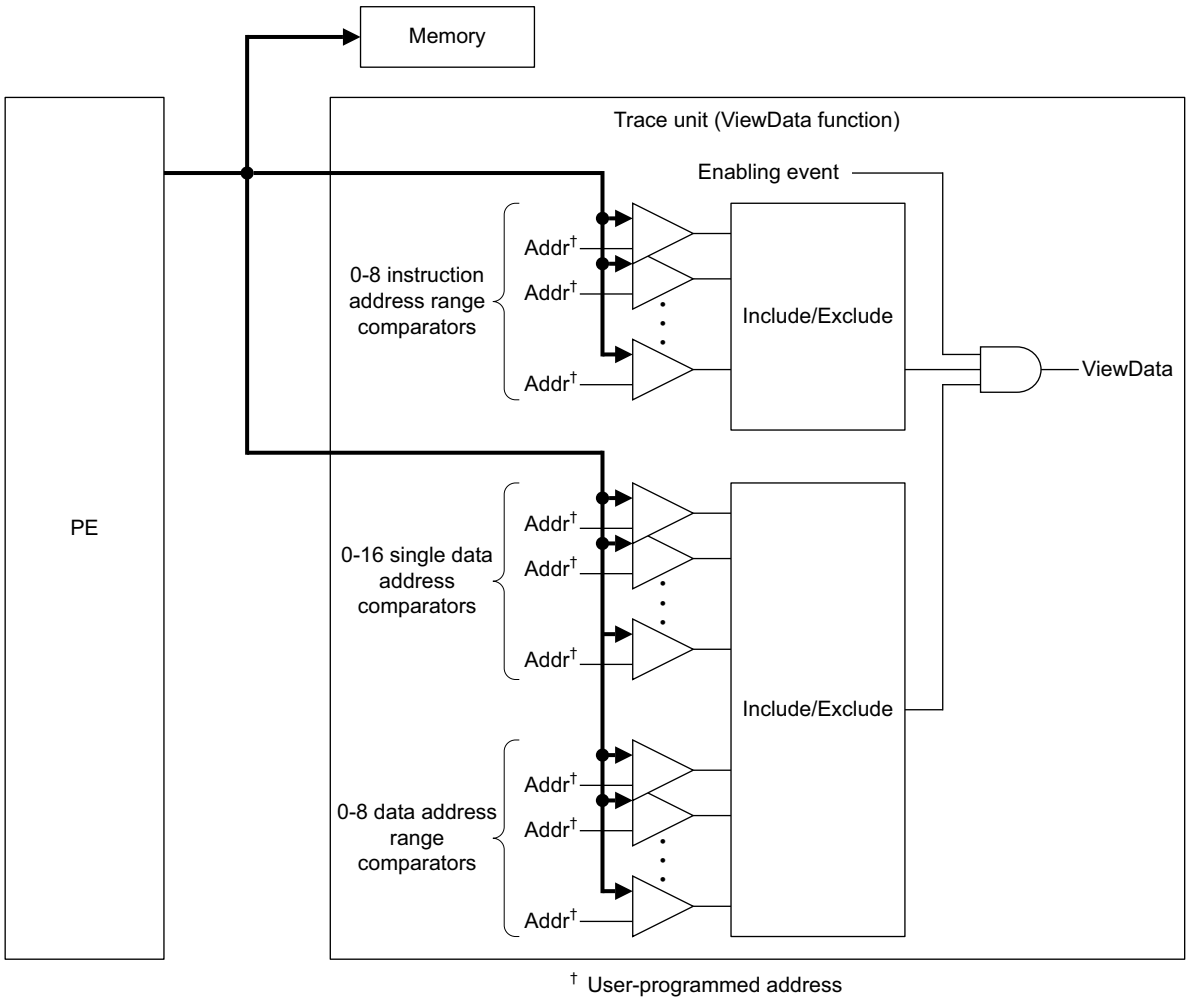


Figure 4-8 The ViewData function

The ViewData function has the following capabilities:

- An include/exclude control that is based on instruction address range comparators. The instruction addresses specified at the inputs of a comparator either:
  - Always have their data transfers traced.
  - Never have their data transfers traced.

  Whether the instructions at the specified addresses have their data transfers traced or not depends on whether the comparators are programmed for the include or the exclude function.

  Single address comparators that are programmed for instruction address comparisons cannot be used with the include/exclude control of the ViewData function.

- An include/exclude control that is based on single data address and data address range comparators.

  The data address that is specified at the inputs of a single address comparator either:
  - Always has data transfers traced.
  - Never has data transfers traced.

  Whether data transfers are included or excluded depends on whether the single address comparator is programmed for the include or the exclude function.

  The data addresses specified at the inputs of an address range comparator either:
  - Always have data transfers at those addresses traced.
  - Never have data transfers at those addresses traced.

  Whether the data transfers are included or excluded depends on whether the address range comparator is programmed for the include or the exclude function.

  Address range comparators and single address comparators that are programmed for data address comparisons with data value comparisons cannot be used with the include/exclude control of the ViewData function.

- An imprecise enabling event input. This input can be programmed to any external resource, or any resource within the trace unit, by using the TRCVDCTLR.

Tracing data transfers associated with specific instructions by including and excluding instruction address ranges

This is useful if only those data transfers associated with a particular instruction range are required.

The include/exclude instruction address ViewData function operates in the same way as the include/exclude instruction address ViewInst function. That is:

- A data transfer must be traced if:
  - It is initiated by an instruction that has an address in an include range.
- It is expected that a data transfer is not traced if:
  - It is initiated by an instruction that has an address in an exclude range.

In addition, in the same way as for the ViewInst include/exclude control, exclude instruction address ranges take priority over include ranges. This means that an exclude instruction address range can be located within a larger include instruction address range.

Also like the ViewInst include/exclude control:

- A data transfer associated with an instruction in an include range might not be traced if any of the other ViewData filtering controls indicate that the data transfer is not to be traced.
- It is expected that a data transfer associated with an instruction in an exclude instruction range address is not traced, regardless of what any of the other ViewData filtering controls indicate.

Note

The default operation of the trace unit when no include instruction address ranges are specified is to trace data transfers for all instructions.

Tracing data transfers by including and excluding data addresses or data address ranges

This is useful if you want to trace only those data transfers that happen at a particular data address or range of data addresses.

In this control, if a data address matches any addresses in the set of include data addresses or ranges that you have specified, then the data at that address must be traced.

Exclude data addresses and ranges take priority over both include data addresses and ranges, and include instruction address ranges.

Like the other ViewInst and ViewData include/exclude controls:

- The data at an include data address might not be traced, for example if any of the other ViewData filtering controls indicate that the data is not to be traced.
- The data at an exclude data address is not traced, regardless of what any of the other ViewData filtering controls indicate.

Note

The default operation of the trace unit when no include data addresses or address ranges are specified is to trace the data for all data accesses.

Tracing data transfers by using the enabling event

Data tracing can also be filtered imprecisely by using the enabling event input to the ViewData function. See Figure 4-8 on page 4-132. This input can be programmed to either an external input, or any resource available in the trace unit, for example the sequencer.

The enabling event input is sampled whenever a data address is accessed, and if the input is high, then the data at that address is traced.

The enabling event input can be programmed to use a resource by using TRCVDCTLR.EVENT. For more information, see *Selecting trace unit resources on page 4-171*.

The enabling event control is imprecise. This means that if, for example, the enabling event input is programmed to use an address range comparator resource, the following scenario might occur:

- The PE performs some execution that triggers the address range comparator into becoming active.
- Some time passes.
- The enabling event input is asserted.

Therefore, there might be a delay between the time when the address range comparator becomes active, and the time when the enabling event input is asserted. This time delay is IMPLEMENTATION DEFINED and might not be fixed for an implementation.

If ViewData is programmed to work with any instruction address range comparators, the ViewData enabling event is not sampled when these instructions are processed. It is only sampled when the data address is processed.

Combining the ViewData filtering controls, or using less than three in a trace run

The ViewData function contains three filtering controls:

- An instruction address range include/exclude control.
- A data address range and single data address include/exclude control.

- An imprecise enabling event input.

Any combination of these can be used in a trace run.

However, as indicated by the AND gate in Figure 4-8 on page 4-132, a data transfer is only traced if all the controls indicate that the data transfer is to be traced. If any one of the three controls indicates that the data transfer is not to be traced, then the data transfer is not traced.

This means that if you want to use only one of the controls, you must disable the other two. Similarly, if you want to use two of the three controls, you must disable the remaining control. You can disable each control as follows:

- To disable the instruction address include/exclude control, do not program any include or exclude instruction address ranges. This indicates to the trace unit that whenever the data address include/exclude control indicates some data transfers that are to be traced, those data transfers must be traced for all instruction addresses.
- To disable the data address include/exclude control, do not program any include or exclude data address comparators. This indicates to the trace unit that whenever the instruction address include/exclude control indicates some data transfers that are to be traced, those data transfers must be traced regardless of what data address they are at.
- To disable the enabling event input, program the input to be always active.

Table 4-4 shows the usage models of the ViewData include/exclude controls, when instruction address range comparators and data address comparators are selected for the include and exclude functions.

Table 4-4 Summary of the ViewData include/exclude control

Instruction address range comparators selected for:		Data address comparators selected for:		Usage model
Include	Exclude	Include	Exclude	
N	N	N	N	For all instructions, trace all data transfers.
N	N	N	Y	For all instructions, trace all data transfers, except for those at excluded data addresses.
N	N	Y	N	For all instructions, trace only those data transfers at included data addresses.
N	N	Y	Y	For all instructions, trace data transfers at included data addresses, but exclude any at excluded data addresses.
N	Y	N	N	For only those instructions that are not in an excluded instruction address range, trace all data transfers.
N	Y	N	Y	For only those instructions that are not in an excluded instruction address range, trace only those data transfers that are not at an excluded data address.
N	Y	Y	N	For only those instructions that are not in an excluded instruction address range, trace only those data transfers that are at an included data address.
N	Y	Y	Y	For only those instructions that are not in an excluded instruction address range, trace data transfers that are at included data addresses, but exclude any that are at excluded data addresses.
Y	N	N	N	For only those instructions in included instruction address ranges, trace all data transfers.
Y	N	N	Y	For only those instructions in included instruction address ranges, trace all data transfers, except for those at excluded data addresses.

Table 4-4 Summary of the ViewData include/exclude control (continued)

Instruction address range comparators selected for:		Data address comparators selected for:		Usage model
Include	Exclude	Include	Exclude	
Y	N	Y	N	For only those instructions in included instruction address ranges, trace only those data transfers at included data addresses.
Y	N	Y	Y	For only those instructions in included instruction address ranges, trace data transfers at included data addresses, but exclude any at excluded data addresses.
Y	Y	N	N	Trace all data transfers for those instructions in included instruction address ranges, but exclude data transfers for any instructions in excluded instruction address ranges.
Y	Y	N	Y	Trace all data transfers for those instructions in included instruction address ranges. However, exclude: • Data transfers for any instructions in excluded instruction address ranges • Data transfers that are at excluded data addresses.
Y	Y	Y	N	Trace only those instructions at included data addresses, for those instructions in included instruction address ranges. However, exclude data transfers for instructions in excluded address ranges.
Y	Y	Y	Y	Trace all data transfers at included data addresses, except for those at excluded data addresses for only those instructions in included instruction range addresses. However, exclude data transfers for instructions in excluded address ranges.

In addition, you can program the trace unit to exclude some or all data transfers for the following instruction types:

- Instructions that transfer data using the PC as the address register with an immediate offset. Such transfers are usually used to load data from literal pools. The data values of these transfers can be determined by inspecting the program image.
- Instructions that transfer data using the SP as the address register with an immediate offset. Such transfers are usually transferring data to or from the stack and these data transfers can often be determined by analyzing other data transfers and from the instruction execution.

The ETMv4 architecture provides these options to help minimize the trace bandwidth.

Tracing of stack transfers on Armv6-M, Armv7-M, and Armv8-M PEs

Armv6-M, Armv7-M, and Armv8-M PEs transfer data to and from the stack when taking exceptions and when returning from exceptions. Tracing of these data transfers can be enabled using TRCVDCTLR.TRCEXDATA. If TRCVDCTLR.TRCEXDATA indicates that tracing of these transfers is enabled, the rest of the ViewData mechanisms are used to determine whether the transfers must be traced.

These transfers are only traced if the parent P0 element is traced. For the stack push transfers, when an exception is taken, the transfers must not be traced if the Exception P0 element is not traced. For the stack pop transfers, on return from an exception, these transfers must not be traced if the Exception Return P0 element is not traced.

These data transfers do not have a parent instruction and therefore any instruction address comparators that are selected for ViewData Include/Exclude must be ignored. Only data address comparators that are selected for ViewData Include/Exclude are used to control the Include/Exclude function.

Flexibility in the interpretation of the ViewData function result

The ETMv4 architecture provides some flexibility in the interpretation of the ViewData function result:

- If ViewData indicates that a data transfer is to be traced, and the parent instruction is traced, then the data transfer must be traced.
- If ViewData indicates that a data transfer is not to be traced, the architecture permits tracing of the data transfer in the following situation:
  - When the data transfer occurs in close proximity to another data transfer that must be traced. For example, if the PE initiates several data transfers in a single clock cycle, then all these transfers might be traced by the trace unit.

Except for the scenario that is mentioned, if the ViewData function indicates that an instruction is not to be traced, then in general it is not traced. An implementation must avoid any unnecessary or excessive tracing because it can affect the efficiency of the trace and might mean that the quantity of trace that is generated exceeds the available bandwidth of the trace port.

Rules for tracing data addresses

Whenever the PE carries out a data transfer, the data address (DA) of that data transfer is not traced if any of the following are true:

- The ViewData instruction address include/exclude control indicates that no data transfers are to be traced for the instruction at the include address.
- The ViewData data address include/exclude control indicates that no data transfers for the data address are to be traced.
- The ViewData enabling event input is inactive for the processor clock cycle that the data transfer is performed on.
- The address of the data transfer is PC-relative, and TRCVDCTLR.PCREL indicates that PC-relative data transfers are not traced.
- The address of the data transfer is SP-relative, and TRCVDCTLR.SPREL indicates that addresses of SP-relative data transfers are not traced, and data value tracing is disabled.
- The address of the data transfer is SP-relative and TRCVDCTLR.SPREL indicates that neither the addresses nor the data values of SP-relative data transfers are traced.

  Note

  It is possible for TRCVDCTLR.SPREL to indicate that the data values of SP-relative data transfers are traced, but not the data addresses.

- The parent P0 element was not traced.
- Both TRCCONFIGR.DA and TRCCONFIGR.DV are 0.

  Note

  If TRCCONFIGR.DA is set to 1, then the trace unit traces the addresses of data transfers.

  If TRCCONFIGR.DV is set to 1, then the trace unit traces the data values of data transfers.

  Whenever they are traced, data addresses are always traced as P1 elements, and data values are always traced as P2 elements. Instructions are traced as P0 elements, though not every instruction type is traced as a P0 element. See *Relationships between P0, P1, and P2 elements on page 2-38* for more information about this.

  It is possible for data value tracing to be enabled when data address tracing is not enabled, that is, for TRCCONFIGR.DV to be set to 1 and TRCCONFIGR.DA set to 0. However, to trace a data value as a P2 element, the trace unit must also generate a P1 element, so that the P2 element can be associated with its

grandparent P0 instruction element. This is because in addition to containing the address of a data transfer, a P1 data address element is also a link between a P2 data value element and a P0 instruction element. See Figure 2-6 on page 2-39.

Therefore, if TRCCONFIGR.DV is set to 1 but TRCCONFIGR.DA is set to 0, the trace unit must generate a P1 element whenever ViewData indicates that a data transfer is traced.

This means that the only programming of TRCCONFIGR.DA and TRCCONFIGR.DV that data addresses are not traced, is when both are set to 0, as shown in Table 4-5.

Table 4-5 When a P1 element is generated

DA	DV	P1 element generated?	The address contained in the P1 element is:
0	0	N	-
0	1	Y	UNKNOWN
1	0	Y	Known
1	1	Y	Known

Rules for tracing data values

Whenever the PE carries out a data transfer, the data value (DV) of that data transfer is not traced if any of the following are true:

- TRCCONFIGR.DV is set to 0.
- The parent data address for that data transfer is not traced, or a P1 element with an UNKNOWN address is not generated to link the data value with the parent instruction.

4.2 Trace unit resources

An ETMv4 trace unit provides the following resources:

- *Counters*.
- *Sequencer on page 4-144*.
- *External inputs on page 4-146*.
- *External outputs on page 4-146*.
- *Memory access resources on page 4-147*.

4.2.1 Counters

Counters that are employed by the ETMv4 architecture are all decrement counters.

The ETMv4 architecture enables a trace unit to connect counter outputs to *trace unit events*, so that a counter at zero state can be used as a resource to activate an event. For example, a counter at zero state might be used to assert an external output or to make ViewInst or ViewData active. See *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177*.

An ETMv4 trace unit can provide up to four 16-bit counters. TRCIDR5.NUMCNTR shows how many counters are implemented. For each counter, the following can be specified:

- The initial counter value. This can be programmed using the TRCCNTVRn.
- The reload value. This can be programmed using the TRCCNTRLDVRn.
- The event that causes the counter to reload with the reload value. This event is called RLDEVENT. In addition, if required, the counter can be programmed so that it automatically reloads whenever it reaches zero.
- The event that enables the counter to decrement. This event is called CNTEVENT. The counter decrements whenever CNTEVENT is active.

Counters within the trace unit are clocked from the processor clock. If the PE is stalled, the counters continue to count. However, if the trace unit has entered a low-power state as a result of the PE stalling, then the counters do not continue to count.

Each counter operates in one of two possible modes:

Normal Mode:

On reaching zero, the counter remains at zero until the reload event, RLDEVENT, occurs. In this mode, the counter-at-zero resource is active for the whole of the time that the counter is at zero.

Self-reload Mode:

When the counter reaches zero, it is reloaded with the reload value the next time the decrement event is active. The counter-at-zero resource is active for one cycle when the counter value is zero, the decrement event is active, and the reload event is not active.

Figure 4-9 to Figure 4-13 on page 4-140 show some examples of counter operation in each mode, for a counter that decrements from `0x3`, and has a reload value of `0x3`.

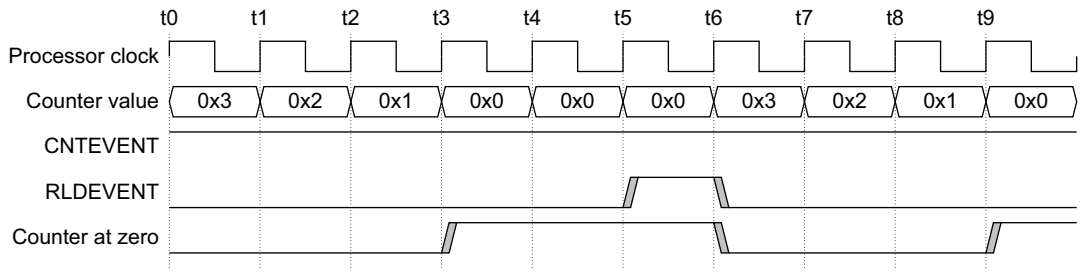


Figure 4-9 Counter operation in Normal mode (example 1)

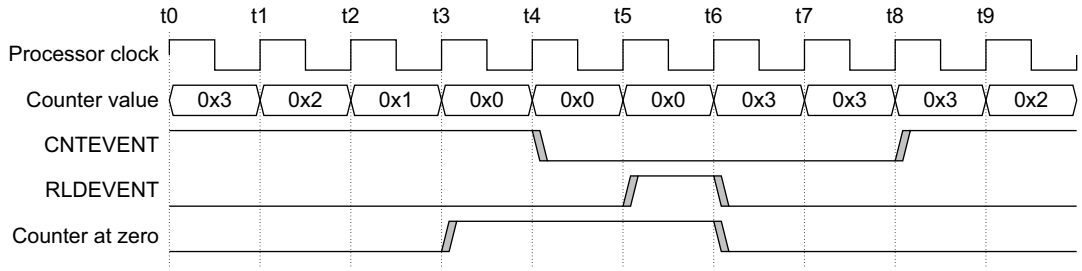


Figure 4-10 Counter operation in Normal mode (example 2)

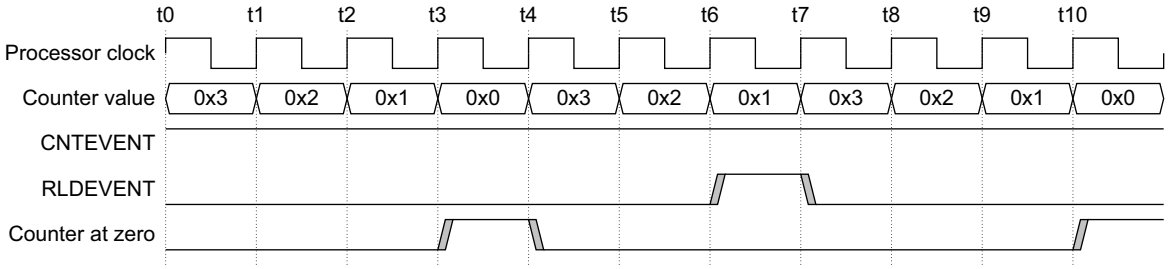


Figure 4-11 Counter operation in Self-reload mode (example 1)

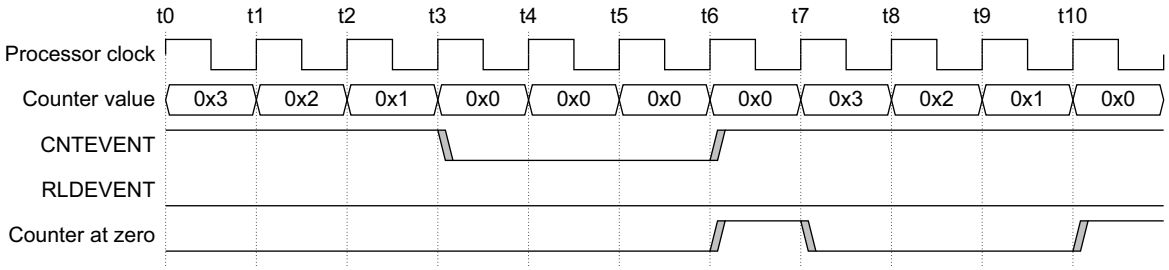


Figure 4-12 Counter operation in Self-reload mode (example 2)

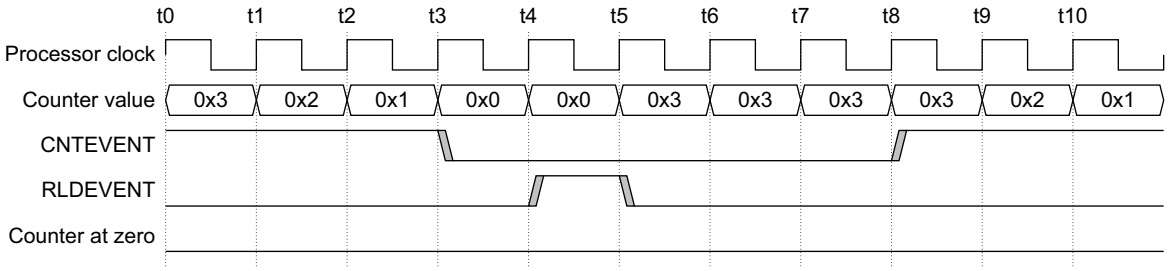


Figure 4-13 Counter operation in Self-reload mode (example 3)

In either mode:

- Whenever the decrement event is inactive, the counter does not decrement.
- The reload event takes priority over the count decrement event.

Which mode a counter operates in is determined by what TRCCNTCTLRn.RLDSELF is programmed to.

Forming a larger counter from two separate counters

Some counters can be chained together to form a larger counter, so that every time one counter reloads, another counter decrements. This is shown in Figure 4-14 on page 4-141. In this example, Counter 0 has a reload value of 0x2.

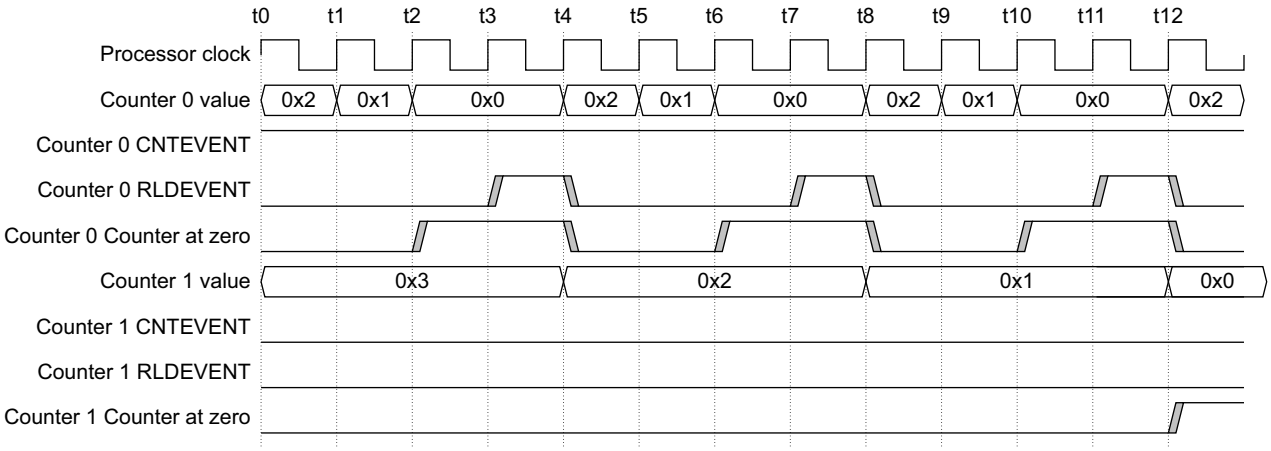


Figure 4-14 The operation of two counters chained together (Normal mode)

Only certain counters can be programmed to do this, as follows:

- Counter 1 can be programmed to decrement when counter 0 reloads.
- Counter 3 can be programmed to decrement when counter 2 reloads.

The decrement event for the higher counter n is active when any of the following occur:

- The lower counter reloads due to either:
  - The reload event that is selected by TRCCNTCTLR<n-1>.RLDEVENT.
  - The self-reload mechanism that is controlled by TRCCNTCTLR<n-1>.RLDSELF.
- The decrement event that is selected by TRCCNTCTLRn.CNTEVENT is active.

When two counters are chained to form a larger counter, the larger counter must appear as a 32-bit counter without any tearing of the values between the two counters. 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 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.

For counters 1 and 3, TRCCNTCTLRn.CNTCHAIN is a RW field that determines whether the counter is chained. For counters 0 and 2, TRCCNTCTLRn.CNTCHAIN is RES0.

Formal description

A dec_action signal is constructed which indicates whether the counter decrements. This is based on TRCCNTCTLRn.CNTEVENT and, for counters which support chaining, on TRCCNTCTLRn.CNTCHAIN and on whether the lower counter is reloading. There are the following globally defined variables.

```
// The current value of each counter
array bits(16) Counters[0..3];
// The counter-at-zero resources
array boolean CounterAtZero[0..3];

//
// EvalAllCounters() is called each clock cycle
//
EvalAllCounters()
    array boolean reload[0..3];
    array bits(16) new_values [0..3];
    (reload[0], new_values[0])= EvalCounter(0, FALSE);
    (reload[1], new_values[1])= EvalCounter(1, reload[0]);
    (reload[2], new_values[2])= EvalCounter(2, FALSE);
    (reload[3], new_values[3])= EvalCounter(3, reload[2]);
```

```
//
// EvalCounter() is called for each counter
//
(boolean, bits(16)) 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 = (Counters[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 && Counters[index] == 0):

    // The counter only decrements if it is non-zero and does not reload and
    // dec_action is active.
    decrement = !reload && (Counters[index] != 0) && dec_action;

    // Determine the next value of the counter
    if reload then
        next_value = TRCCNTRLDVR[index].VALUE;
    elsif decrement then
        next_value = Counters[index] - 1;
    else
        next_value = Counters[index];

    CounterAtZero[index] = resource_active;
    return (reload, next_value);
```

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.

This behavior is summarized in Table 4-6 and Table 4-7 on page 4-143. The term X is used to indicate that the value of the decrement action or the counter value has no effect on the operation.

Table 4-6 Counter operation in Normal mode

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	Not 0	Stable	No	No activity

Table 4-6 Counter operation in Normal mode (continued)

RLDEVENT	dec_action	Counter value	Action	Resource Active	Notes
Inactive	1	Not 0	Decrement	No	Decrement when not zero
Active	X	0	Reload	Yes	Reload, but resource is active because counter is at zero
Active	X	Not 0	Reload	No	Reload

Table 4-7 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 counter is at zero
Inactive	1	0	Reload	Yes	Reload because dec_action is active and the counter is at zero. The resource is active only in this cycle.
Inactive	1	Not 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

Provision of a reduced function counter

The ETMv4 architecture supports the implementation of one reduced function counter that has the following attributes:

- The decrement event, TRCCNTCTLRn.CNTEVENT, is not implemented for the counter. The counter is permanently enabled to decrement on every processor clock cycle.
- The reload event, TRCCNTCTLR.RLDEVENT, is not implemented for the counter. The counter reloads every time it reaches zero. This is equivalent to Self-reload mode always being enabled.
- TRCCNTCTLRn.RLDSELF is not implemented and the counter behaves as if Self-reload mode is always enabled.
- TRCCNTCTLRn.CNTCHAIN is not implemented because this is the only counter.
- TRCCNTRLDVRn is implemented and specifies the reload value for the counter.
- When the trace unit is enabled, as defined in *Trace unit behavior when the trace unit is enabled on page 3-100*, the counter always starts at a value equal to or less than the reload value.
- The counter value cannot be written to or read, and therefore cannot be saved or restored.

If implemented, this counter is always counter 0. TRCIDR5.REDFUNCNTR shows whether counter 0 is implemented as a reduced function counter or not.

Programming the counters

The registers that are used to program the counters are:

- *TRCCNTRLDVRn, Counter Reload Value Registers, n=0-3 on page 7-360*.
- *TRCCNTCTLRn, Counter Control Registers, n=0-3 on page 7-359*.
- *TRCCNTVRn, Counter Value Registers, n=0-3 on page 7-361*.

4.2.2 Sequencer

An ETMv4 trace unit can contain a sequencer state machine that has four states, as shown in Figure 4-15.

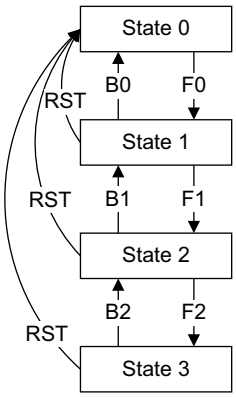


Figure 4-15 Sequencer states

TRCIDR5.NUMSEQSTATE shows whether the state machine is implemented.

You can connect the sequencer to events, so that the sequencer transitions from one state to another when certain events occur. The TRCSEQEVRn registers, enable you to choose what events cause the state machine to transition. Each register enables you to choose the following:

- An event that causes the state machine to progress to the next state.
- An event that causes the state machine to transition backwards to the previous state.

You can choose different events to cause the sequencer to transition between different states. For example, a particular event might cause an F0 transition from state 0 to state 1 on one processor clock cycle, whereas a different event might cause an F1 transition from state 1 to state 2 on the next processor clock cycle. A third independent event might cause a B1 transition backwards from state 2 to state 1 on the third clock cycle.

Forward transitions take priority over backward transitions. This means that if two events occur that means a forward transition conflicts with a backward transition in the same processor clock cycle, then the forward transition takes priority and the backward transition is ignored.

The sequencer can progress through multiple states in a single processor clock cycle. For example, if the sequencer is in state 0 and the 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.

The sequencer can also be reset to state 0 from any other state. The TRCSEQRSTEVR enables you to choose an event to reset the sequencer.

When the event that causes a RST transition occurs, the sequencer always finishes the clock cycle in state 0 and cannot progress to another state until the next clock cycle.

A RST transition always takes priority over any other transitions, so that if the event that causes a RST transition is active in the same clock cycle as events that cause other transitions, then the RST transition takes priority and all other transitions are ignored.

Table 4-8 provides a summary of the sequencer state transitions.

Table 4-8 Summary of the sequencer state transitions

		End state			
		0	**1**	**2**	**3**
Start state	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 | (B1 & B0 & !F2 & !F1 & !F0)	B1 & (!B0 | F0) & !F1 & !F2	(!B1 | F1) & !F2	F2
	3	RST | (B2 & B1 & B0 & !F2 & !F1 & !F0)	B2 & B1 & (!B0 | F0) & !F2 & !F1	B2 & (!B1 | F1) & !F2	!B2 | F2

Note

If multiple events that cause transitions become active in one processor clock cycle, there is no guarantee that the order of these 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.

The ETMv4 architecture provides each sequencer state as a trace unit resource, so that states can be used to trigger other events in the trace unit. This means that the sequencer can be used as shown in Figure 4-16.

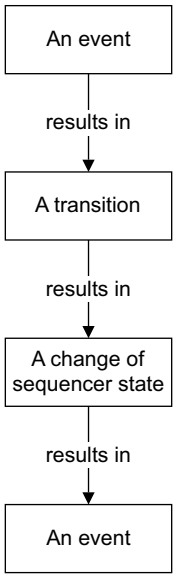


Figure 4-16 Using a sequencer state as a trace unit resource

See *Selecting trace unit resources on page 4-171*.

If the sequencer progresses through multiple stages in a single processor clock cycle, then for each state that it passes through, the event that it triggers must be active for that cycle. For example, if the sequencer is in state 0, and in one processor clock cycle it moves to state 3, then the 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 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 events that state 2 and 1 are connected to must not become active.

Programming the sequencer

The registers that are used to program the sequencer are:

- *TRCSEQEVRn, Sequencer State Transition Control Registers, n=0-2 on page 7-400*.
- *TRCSEQRSTEVR, Sequencer Reset Control Register on page 7-401*.
- *TRCSEQSTR, Sequencer State Register on page 7-401*.

4.2.3 External inputs

The ETMv4 architecture supports the implementation of between 0 and 256 external inputs to a trace unit. The number of external inputs that a trace unit has is IMPLEMENTATION DEFINED. TRCIDR5.NUMEXTIN shows how many external inputs are implemented.

Typically, a trace unit has:

- An IMPLEMENTATION DEFINED number of inputs for connection to the performance monitoring events of the PE.
- Four inputs for ASIC-specific functionality, that is, inputs that come from elsewhere within the SoC.

The ETMv4 architecture enables up to four external inputs to be selected as resources, so that they can be used to trigger events in the trace unit. See *Selecting trace unit resources on page 4-171*. The TRCEXTINSELR is used to choose four of the external inputs for this function.

4.2.4 External outputs

The ETMv4 architecture supports between zero and four external outputs. The number of external outputs that a trace unit has is IMPLEMENTATION DEFINED, within the following constraints:.

- Prior to ETMv4.3, between one and four external outputs are implemented.
- From ETMv4.3, between zero and four external outputs are implemented.

These external outputs are used to indicate events to a trace analyzer.

Events are controlled by trace unit resources. For example, an instruction address comparator can be used to drive one of the events.

The external outputs can be driven by any trace unit resource, for example a comparator, counter or particular sequencer state.

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. 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.

To program an external output, use the TRCEVENTCTL0R to select a trace unit resource.

The TRCIDR0.NUMEVENT field shows how many events are supported for the particular implementation. See *TRCIDR0, ID Register 0 on page 7-370*.

4.2.5 Memory access resources

Memory access resources include:

- *Single address comparators on page 4-149*.
- *Address range comparators on page 4-151*.
- *Data value comparators on page 4-153*.
- *Context ID comparators on page 4-157*.
- *Virtual context identifier comparators on page 4-158*.
- *PE comparator inputs on page 4-159*.
- *Single-shot controls for address comparators on page 4-159*.
- *Using data address comparators and data value comparators to detect store-exclusive transfers on page 4-162*.

Table 4-9 summarizes each of these resources.

Table 4-9 Summary of trace unit memory access resources

Resource type	Purpose	Number implemented
Single address comparator	Matches on either: • A single instruction address. • Optionally, a single data address. Can be used: • As an individual trace unit resource. • For the start/stop control in the ViewInst function. • For the include/exclude control in the ViewData function.	0-16. **Note** Single address comparators are implemented in pairs within a trace unit. One pair of single address comparators can be programmed to comprise one address range comparator. See *Address range comparator* in this table.
Address range comparator	Matches on either: • An instruction address in a range of instruction addresses. • Optionally, a data address in a range of data addresses. Can be used either: • As an individual trace unit resource • For an include/exclude control in either: — The ViewInst function. — The ViewData function, if data tracing is implemented.	0-8. See *Single address comparator* in this table.
Data value comparator	Matches on the data value of a data transfer. Can be used with either: • Single data address comparators. • Data address range comparators.	0-8.
Context ID comparator	Matches when the PE is executing with the Context ID that the Context ID comparator is programmed with. Can be used either: • As an individual trace unit resource. • With either: — A single instruction address comparator. — Optionally, a single data address comparator. — An instruction address range comparator. — Optionally, a data address range comparator.	0 -8.

Table 4-9 Summary of trace unit memory access resources (continued)

Resource type	Purpose	Number implemented
Virtual context identifier	Matches when the PE is executing with the Virtual context identifier that the Virtual context identifier comparator is programmed with. Can be used either: • As an individual trace unit resource. • With either: — A single instruction address comparator. — Optionally, a single data address comparator. — An instruction address range comparator. — Optionally, a data address range comparator.	0-8.
PE comparator input	Can be driven from a comparator within the PE. Can be used either: • As an individual trace unit resource. • For the start/stop control in the ViewInst function.	0 -8
Single-shot control for an address comparator	Shows when an accessed instruction or data transfer is nonspeculative. Makes it possible for a trace unit event to be activated based on only nonspeculative execution. Can be used to provide a trace analyzer with a start trace or stop trace signal. Can be used with a combination of: • Single instruction address comparators. • Single data address comparators, that might or might not have data value comparators that are associated with them. • Instruction address range comparators. • Data address range comparators, that might or might not have data value comparators that are associated with them.	0 -8.

Arm strongly recommends that:

- Any instructions or data items which might be traced must be able to cause a comparator match.
- Any instructions or data items which are able to cause a comparator match must be able to be traced.

This avoids scenarios where an item is present in the trace but does not cause a match, or a spurious match occurs but the item is not present in the trace. For example, a spurious match might occur when enabling or disabling the trace unit.

Note

Filtering of the trace stream using the ViewInst or ViewData functions might prevent items from appearing in the trace stream. However, filtering of the trace has no effect on comparator operation so it is not applicable to this recommendation.

For Armv8 PEs, if tagged addresses are in use, as defined in *Armv8 Architecture Reference Manual*, then:

- The Access address for instruction addresses does not include the tag.
- Depending on the current Exception level, bits[63:56] are either:
  - The sign-extension of bit[55].
  - All zeros.

Single address comparators

An ETMv4 trace unit provides between 0 and 16 single address comparators, that each compare either the instruction address or the data address with a user-programmed value. These single address comparators are implemented within the trace unit in pairs, therefore a trace unit implementation must contain an even number of single address comparators. TRCIDR4.NUMACPAIRS shows how many pairs of single address comparators are implemented.

Single address comparators can be used:

- For the start/stop control in the ViewInst function. See Figure 4-1 on page 4-119.
- In pairs, to form address range comparators. See *Address range comparators on page 4-151*. These address range comparators can then be used in the include/exclude controls in either the ViewInst or the ViewData function. See *Overview of the ViewInst function on page 4-119* and *Overview of the ViewData function on page 4-132*.
- As individual trace unit resources to indicate a trace unit event to a trace analyzer or other trace unit resource. For example, a single address comparator might be programmed to match on instruction address `0x1000`, and might be selected as a resource for activating both:
  - A transition between trace unit sequencer states.
  - The assertion of a trace unit external output to a trace analyzer.

  In this case, when the instruction at address `0x1000` is executed, an output from the trace unit is asserted, and the trace unit sequencer resource changes state.

Each single address comparator can be programmed to match on any one of the following:

- An instruction address, regardless of condition code that is passed or failed.
- Optionally, a data load address only.
- Optionally, a data store address only.
- Optionally, a data load or data store address.

In addition, there are the following controls over each comparator:

- If the comparator is programmed to match on a data address, the comparator can be programmed to mask bits[63:56] of the address so that these bits are ignored.

  Note

  This feature is only implemented if the trace unit implements 64-bit data address tracing.

- 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.

When comparing an instruction address, a single address comparator matches if its programmed address exactly matches the address of the lowest byte of an instruction. 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 comparator is programmed with `0x1000`, then it always matches on instruction address `0x1000`, but it is IMPLEMENTATION DEFINED whether it matches on `0x1001`, `0x1002`, or `0x1003`. To avoid unexpected behavior from a single address comparator, Arm recommends that the comparator is always programmed with an address that is for the lowest byte of an instruction.

Note

Armv8 provides support for disabling the use of IT instructions on more than one subsequent instruction, using the ITD bit in the SCTLR, HSCTLR, and SCTLR_EL1 PE registers. When the ITD bit is set to 1, and if a single address comparator is programmed to match on the address of an instruction that is after an IT instruction, then it is IMPLEMENTATION SPECIFIC whether that comparator matches.

In these scenarios, Arm recommends that the single address comparator is programmed to match on the address of the IT instruction.

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.

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.

When comparing the address of a data transfer, a single address comparator matches if its programmed address exactly matches an address that is accessed by a data transfer. For example, if a comparator is programmed with the address 0x2000, and data is either stored to or loaded from 0x2000, then the comparator matches. Equally, if a comparator is programmed with the address 0x2000 and a 64-bit word is either stored to or loaded from 0x1ffc, then the comparator matches. If a trace analyzer is required to watch for data transfer at a range of addresses, for example from 0x2000 to 0x200F, then an address range comparator can be used. See *Address range comparators on page 4-151*.

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. In this case, if the address that the single address comparator is programmed with is matched more than once:

- The comparator must signal a match at least once.
- The comparator must not signal more matches than the number of accesses that match the programmed address.

A single address comparator might match when:

- An instruction is speculatively executed, if the comparator is programmed to match on that instruction address.
- A data transfer is speculatively carried out, if the comparator is programmed to match on the address that the data is stored to or loaded from. In this case, the comparator might match even if:
  - The data transfer causes a synchronous or an asynchronous abort.
  - The data transfer is part of a failed store-exclusive operation.
  - The instruction that caused the data transfer is canceled because of mis-speculation.

Address comparators do not match when instructions are executed in Debug state, or when instructions executed in Debug state initiate data transfers.

When a Single Address Comparator is programmed to match on LE or LETP instructions, one of the following occurs:

- The comparator fires when the instruction is executed.
- The comparator does not fire when the instruction is executed.

This behavior might change dynamically.

Arm recommends programming Single Address Comparators to match on the instruction before an LE or LETP instruction.

When a Single Address Comparator is programmed to match on the instruction after the BF branch point and an implicit branch occurs, one of the following occurs:

- The comparator fires.
- The comparator does not fire.

This behavior might change dynamically.

Arm recommends programming Single Address Comparators to match on the instruction before the BF branch point.

Using single address comparators in conjunction with other comparator types

Each single address comparator can be used in conjunction with one, or a combination of, the following:

- A data value comparator. See *Data value comparators on page 4-153*.
- A Context ID comparator. See *Context ID comparators on page 4-157*.
- A Virtual context identifier comparator. See *Virtual context identifier comparators on page 4-158*.

Programming single address comparators

A single address comparator is programmed by setting:

1. The Address Comparator Value Register, TRCACVRn to the required address value.
2. The associated Address Comparator Access Type Register, TRCACATRn, to a value that specifies:
   - Whether the comparator matches on an instruction address or a data address.
   - Various other settings.

Address range comparators

Two single address comparators can be arranged to form one address range comparator. An address range comparator is programmed with an address range, so that whenever any address in that range is accessed, the comparator matches. A trace unit can contain between zero and eight address range comparators.

Address range comparators can be used:

- For the include/exclude controls in either the ViewInst or the ViewData functions. See *Overview of the ViewInst function on page 4-119* and *Overview of the ViewData function on page 4-132*.
- As trace unit resources to indicate a trace unit event to a trace analyzer or other trace unit resource. For example, an address range comparator might be programmed to include data addresses in the range `0x0` to `0x2C`, and might be selected to activate a trace unit external output. In this case, whenever any data address in the range `0x0` to `0x2C` is accessed, the external output is asserted.
- As individual trace unit resources to indicate store-exclusive transfers. See *Using data address comparators and data value comparators to detect store-exclusive transfers on page 4-162*.

An address range comparator is programmed by setting:

- The first single address comparator to the start address of the instruction or data address range.
- The second single address comparator to the end address of the instruction or data address range.

Note

The address that the second single address comparator is programmed with must be greater than or equal to the address that the first single address comparator is programmed with, that is, the end address must be greater than or equal to the start address.

If the two address comparators are not programmed in the same way, then the behavior of the comparator is CONSTRAINED UNPREDICTABLE. That is, the comparator might match at any time or might not match. An example, is if one comparator is programmed to match on an instruction address and the other is programmed to match on a data address.

UNPREDICTABLE behavior occurs if the TRCACATRn registers for both single address comparators are not programmed to the same values. For example, UNPREDICTABLE behavior occurs if:

- The CONTEXT fields in the single address comparators are programmed to different values.

- The EXLEVEL_NS field in the first comparator is programmed with 0b0000 and the EXLEVEL_NS field in the second comparator is programmed with 0b0010.

When an address range comparator is programmed with an instruction address range, the comparator matches if the accessed address is in the following range:

```
(access address ≥ start address) AND (access address ≤ end address)
```

When comparing an instruction address, an address range comparator matches if its programmed address range contains the address for the lowest byte of an instruction. If the programmed address range contains addresses for one or more bytes of the instruction, but does not contain the address for the lowest byte of the instruction, then it is IMPLEMENTATION SPECIFIC whether the comparator matches. 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 address range comparator always matches. However, if the programmed address range starts at either 0x1001, 0x1002, or 0x1003, then it is IMPLEMENTATION SPECIFIC whether the address range comparator matches. To avoid unexpected behavior from an address range comparator, Arm recommends that the comparator is always programmed with an address range that starts with an address for the lowest byte of an instruction.

Note

Armv8 supports disabling IT instructions on more than one subsequent instruction, using the ITD bit in the SCTLR, HSCTLR, and SCTLR_EL1 PE registers. If the ITD bit is set to 1, and if an address range comparator is programmed to include the address of an instruction that is after an IT instruction in its address range, then it is IMPLEMENTATION SPECIFIC whether that comparator matches.

In these scenarios, Arm recommends that the address range comparator is programmed to include the address of the IT instruction in its address range.

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.

Arm recommends that an ARC is programmed to include both the MOVPRFX instruction and the instruction immediately after the MOVPRFX instruction.

When an address range comparator is programmed with a data address range, the comparator matches if a data transfer accesses any bytes that are in the range start address up to and including end address.

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.

In either case:

- The comparator must signal a match at least once.
- The comparator must not signal more matches than the number of accesses that match addresses in the programmed range.

Address range comparators might match on speculative execution. That is, if the PE speculatively executes instructions or data transfers whose addresses are in the programmed range of an address range comparator, then that comparator might match. When a comparator is programmed with a data address range, that comparator might match even if:

- The data transfer causes a synchronous or asynchronous abort.
- The data transfer is part of a failed store exclusive operation.
- The instruction that causes the data transfer is canceled because of mis-speculation.

Address comparators do not match when instructions are executed in Debug state, or when instructions executed in Debug state initiate data transfers.

When an Address Range Comparator is programmed to include an `LE` or `LETP` instruction, but does not include the address of the instruction before the `LE` or `LETP` instruction, one of the following occurs:

- The comparator fires when the `LE` or `LETP` is executed.
- The comparator does not fire when the `LE` or `LETP` is executed.

This behavior might change dynamically.

Arm recommends programming Address Range Comparators to include both the `LE` or `LETP` instruction and the instruction immediately before the `LE` or `LETP` instruction.

When an Address Range Comparator is programmed to include the instruction after the BF branch point, but not before the BF branch point, and an implicit branch occurs, one of the following occurs:

- The comparator fires.
- The comparator does not fire.

This behavior might change dynamically.

Arm recommends programming Address Range Comparators to include both the instruction before and the instruction after a BF branch point.

Using address range comparators with other comparator types

Each address range comparator can be used with one, or a combination of, the following:

- A data value comparator. See *Data value comparators*.
- A Context ID comparator. See *Context ID comparators on page 4-157*.
- A Virtual context identifier comparator. See *Virtual context identifier comparators on page 4-158*.

Programming address range comparators

An address range comparator is programmed by setting:

1. The Address Comparator Value Register, TRCACVRn, for the first single address comparator to the start address of the instruction or data address range.
2. The Address Comparator Value Register, TRCACVRn, for the second single address comparator to the end address of the instruction or data address range.
3. The Address Comparator Access Type Register, TRCACATRn, associated with each TRCACVRn so that:
   - If the comparator is being programmed for data address range comparison with data value comparison, TRCACATRn.DATARANGE is set to 1. This means that the comparator constitutes a pair of comparators that comprises an address range comparator.
   - Other settings are made as appropriate.

Data value comparators

An ETMv4 trace unit provides between zero and eight data value comparators.

The purpose of a data value comparator is to compare a data value after a data address match has occurred. Therefore, data value comparators are used with data address comparators, that can be either single address comparators or address range comparators. The number of data value comparators that are implemented is shown in TRCIDR4.NUMDVC.

When implemented, each data value comparator is associated with a particular pair of single address comparators inside the trace unit. However, an implementation might contain fewer data value comparators than pairs of single address comparators, and this means that for some implementations, not all pairs of single address comparators have an associated data value comparator.

Each data value comparator can be used with either:

- A data address range comparator, that comprises one of the pairs of single address comparators. In this case:
  - Only one of the pair of single address comparators contains the controls that enable a data value comparator to be selected. The single address comparator that contains these controls is the lower single address comparator of each pair.
  - The two single address comparators must not be used as single address comparators. They must only be used as an address range comparator. This is because the behavior of the comparators as single address comparators is UNPREDICTABLE when a data value comparator is selected for use with the address range comparator.
- A single data address comparator. In this case:
  - The single data address comparator is always the lower single address comparator of the pair.
  - The behavior of the address range comparator that the pair can comprise is UNPREDICTABLE, therefore the address range comparator must not be used. However, the behavior of the other single address comparator in the pair is unaffected, therefore this comparator can be used as a single address comparator, although only if a data value match is not also required.

In either case, whenever a data value comparator is used, the data value comparison takes place simultaneously or at some time after a data address match occurs. Therefore:

- If the data value comparator is used with a data address range comparator, then whenever any data address in the programmed range is matched, a data value comparison takes place simultaneously or afterwards.
- If the data value comparator is used with a single data address comparator, then whenever that address is matched, a data value comparison takes place simultaneously or afterwards.

It is the TRCACATRn.DATARANGE field for the address comparator that can be programmed to specify whether the data value comparator is for use with a single data address comparator, or with a data address range comparator. This field is present in only one of a pair of single address comparators.

Where a data value comparator is implemented and associated with a single address comparator, that single address comparator has a data value comparison enable field that is implemented in its Address Comparator Access Type Register. See TRCACATRn.DATAMATCH.

When programming this field, the options are:

- No data value comparison is performed. The associated data value comparator is not used.
- A data value comparison is performed. The single address comparator signals a match only if both the data address and data value match.
- A data value comparison is performed. The single address comparator signals a match only if both:
  - The data address matches.
  - The data value does not match.

Data value comparators might signal a match on speculative execution. For example, if the PE speculatively executes a data transfer whose address causes an address comparator to signal a match, then its associated data value comparator might also signal match if the data value of the data transfer is matched.

Each data value comparator has an associated data value mask, that enables some bits of the data value to be ignored whenever a data value comparison takes place.

The following subsections describe:

- *Programming data value comparators*.
- *Rules when a data value comparator is programmed for use with a single address comparator*.
- *Rules when a data value comparator is programmed for use with an address range comparator on page 4-156*.

Programming data value comparators

When programming a data value comparator, the size of the data value comparison can be specified.

The TRCDVCVRn and TRCDVCMRn are used to program the data value comparators.

The following constraints apply:

- The alignment of the address that the data address comparator is programmed with must correspond to the size of the data value that the associated data value comparator is programmed with. This means that:
  - If the data value is a halfword, then the data address that the address comparator is programmed with must be halfword-aligned.
  - If the data value is a word, then the data address that the address comparator is programmed with must be word-aligned.
  - If the data value is a doubleword, then the data address that the address comparator is programmed with must be doubleword-aligned.
- If the size of the data value is less than a doubleword, so that it occupies only part of the TRCDVCVRn, the data value must be repeated to fill all the TRCDVCVRn. For example:
  - If the data value is a byte, then the TRCDVCVRn must consist of eight bytes that each contain that data value.
  - If the data value is a halfword, then the TRCDVCVRn must consist of four halfwords that each contain that data value.
  - If the data value is a word, then both the upper and lower 32 bits of the TRCDVCVRn must contain that data value.
- Whenever the data value is less than a doubleword, if a mask is applied by programming bits in the associated TRCDVCMRn, then that mask, that is, that pattern of bits, must be repeated to fill the TRCDVCMRn. For example:
  - If the data value is a byte, and it is required that bits[2:0] are ignored whenever a comparison takes place, then the least significant byte in the associated TRCDVCMRn is programmed with `0b00000111`. This pattern of bits must be repeated in the seven most significant bytes of the TRCDVCMRn.
  - If the data value is a halfword, so that the pattern of bits for the mask is also a halfword, then that pattern of bits must be repeated four times to fill the TRCDVCMRn.
  - If the data value is a word, and a word-sized mask is applied, then that word-sized mask must be repeated twice to fill the TRCDVCMRn.

Rules when a data value comparator is programmed for use with a single address comparator

When a data value comparator is programmed for use with a single address comparator, the following rules apply:

- The single address comparator does not signal a match if, when the PE initiates a data transfer, the access is not aligned appropriately for the data value size that the data value comparator is programmed with. This means that:
  - If the data value comparator is programmed with a halfword data value, the single address comparator only signals a match on accesses that are halfword-aligned.
  - If the data value comparator is programmed with a word data value, the single address comparator only signals a match on accesses that are word-aligned.
  - If the data value comparator is programmed with a doubleword data value, the single address comparator only signals a match on accesses that are doubleword-aligned.

- The single address comparator does not signal a match if, when the PE initiates a data transfer, the size of that data transfer is smaller than the data value size that the data value comparator is programmed with. Table 4-10 summarizes this.

Table 4-10 Types of access that a single address comparator signals a match on, for different sizes of data value

Data value comparator data value size	The single address comparator only signals a match on:
Byte	A byte access to the address that the single data address comparator is programmed with.
	A halfword access to the address that the single data address comparator is programmed with.
	A halfword access to a lower address, where the access overlaps the address that the single data address comparator is programmed with.
	A word access to the address that the single data address comparator is programmed with.
	A word access to a lower address, where the access overlaps the address that the single data address comparator is programmed with.
	A doubleword access to the address that the single data address comparator is programmed with.
	A doubleword access to a lower address, where the access overlaps the address that the single data address comparator is programmed with.
Halfword	A halfword access to the address that the single data address comparator is programmed with.
	A word access to the address that the single data address comparator is programmed with.
	A word access to a lower address, where the access overlaps the address that the single data address comparator is programmed with and the access address is halfword-aligned.
	A doubleword access to the address that the single data address comparator is programmed with.
	A doubleword access to a lower address, where the access overlaps the address that the single data address comparator is programmed with and the access address is halfword-aligned.
Word	A word access to the address that the single data address comparator is programmed with.
	A doubleword access to the address that the single data address comparator is programmed with.
	A doubleword access to a lower address, where the access overlaps the address that the single data address comparator is programmed with and the access address is word-aligned.
Doubleword	A doubleword access to the address that the single data address comparator is programmed with.

When using a data value comparator with a single address comparator, the behavior of the address range comparator is CONSTRAINED UNPREDICTABLE. That is, the address range comparator might match at any time or might not match. The behavior of the other single address comparator in this pair is not affected.

Rules when a data value comparator is programmed for use with an address range comparator

When a data value comparator is programmed for use with an address range comparator, the following rules apply:

- The address range comparator does not signal a match if, when the PE initiates a data transfer, the access is not aligned appropriately for the data value size that the data value comparator is programmed with. This means that:
  - If the data value comparator is programmed with a halfword data value, the address range comparator only signals a match if the access is halfword-aligned.
  - If the data value comparator is programmed with a word data value, the address range comparator only signals a match if the access is word-aligned.

— If the data value comparator is programmed with a doubleword data value, the address range comparator only signals a match if the access is doubleword-aligned.

- The address range that the address range comparator is programmed with must be an integer multiple of the data value size that the data value comparator is programmed with. For example, if the data value comparator is programmed with a word, then the address range comparator must be programmed with an address range that is a multiple of 32 bits. This means that if, for example, the requirement is to use a trace unit to watch for word accesses within the four words starting at `0x1000`, then the start address must be `0x1000` and the end address must be the last byte of the last word, `0x100F`.
- The address range comparator does not signal a match if, when the PE initiates a data transfer, the size of that data transfer is different to the data value size that the data value comparator is programmed with.

Table 4-11 Types of access that an address range comparator signals a match on, for different sizes of data value

Data value comparator data value size	The address range comparator only signals a match on:
Byte	A byte access to the comparison address
Halfword	A halfword-aligned halfword access to the comparison address
Word	A word-aligned word access to the comparison address
Doubleword	A doubleword-aligned doubleword access to the comparison address

When using an address range comparator with a data value comparator, the two related single address comparators must not be used and the behavior of these as resources is CONSTRAINED UNPREDICTABLE. The single address comparators might match or might not match.

Context ID comparators

An ETMv4 trace unit provides between zero and eight Context ID comparators.

A Context ID comparator can be either:

- Associated with a single address comparator.
- Associated with an address range comparator.
- Used on its own as a trace unit resource.

When a Context ID comparator is associated with either a single address comparator or an address range comparator, that address comparator can only signal a match if both:

- The accessed address matches.
- The PE is executing with the Context ID that the Context ID comparator is programmed with.

When used on its own, a Context ID comparator matches whenever the PE is executing with the Context ID that the Context ID comparator is programmed with.

Note

When using a Context ID comparator as an independent trace unit resource to activate a trace unit event, the time that the event is activated relative to the time that the Context ID comparator becomes active might be imprecise. For more information, see *About the timing of events that are activated by trace unit resources on page 4-178*.

A Context ID comparator is associated with a single address comparator by programming TRCACATRn.CONTEXT for the single address comparator.

If a Context ID comparator is required for use with an address range comparator formed from two single address comparators, then TRCACATRn.CONTEXT for both comparators must be programmed with the same value.

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. In this case, if the Context ID that the Context ID comparator is programmed with is matched more than once:

- The comparator must signal a match at least once.
- The comparator must not signal more matches than the number of instructions that are executed with the Context ID that the comparator is programmed with.

A Context ID comparator might match on speculative execution, that is, a Context ID comparator might match if the PE speculatively changes the Context ID.

The Context ID comparators do not match on any instructions that are executed in Debug state.

The Context ID might change at points that are not Context synchronization events, for example, when a system instruction is used to write to the current Context ID register. In these scenarios, the Context ID comparator might compare against the old or new Context ID value for any instruction after the P0 element before the system instruction, up to the Context synchronization event after the system instruction.

Virtual context identifier comparators

An ETMv4 trace unit provides between zero and eight Virtual context identifier comparators.

Like a Context ID comparator, a Virtual context identifier comparator can be either:

- Associated with a single address comparator.
- Associated with an address range comparator.
- Used on its own, as a trace unit resource.

When a Virtual context identifier comparator is associated with either a single address comparator or an address range comparator, that address comparator can only signal a match if both:

- The accessed address matches.
- The PE is executing with the Virtual context identifier that the Virtual context identifier comparator is programmed with.

When used on its own, a VMID comparator matches whenever the PE is executing with the Virtual context identifier that the Virtual context identifier comparator is programmed with.

Note

When using a Virtual context identifier comparator as an independent trace unit resource to activate a trace unit event, the time that the event is activated relative to the time that the Virtual context identifier comparator becomes active might be imprecise. For more information, see *About the timing of events that are activated by trace unit resources on page 4-178*.

A Virtual context identifier comparator is associated with a single address comparator by programming TRCACATRn.CONTEXT for the single address comparator.

If a Virtual context identifier comparator is required for use with an address range comparator, that is formed from two single address comparators, then TRCACATRn.CONTEXT for both comparators must be programmed with the same value.

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. In this case, if the Virtual context identifier that the Virtual context identifier comparator is programmed with is matched more than once:

- The comparator must signal a match at least once.
- The comparator must not signal more matches than the number of instructions that are executed with the VMID that the comparator is programmed with.

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.

The Virtual context identifier comparators do not match on any instructions that are executed in Debug state.

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 the VTTBR or 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.

If the PE implements Armv8.4-Trace, the Virtual context identifier comparators must not match when not allowed by TRFCR_EL2.CX.

PE comparator inputs

An ETMv4 trace unit provides up to eight inputs that can be driven from comparators within the PE. For example, a PE might contain breakpoint or watchpoint comparators that are programmed to match on certain instruction or data addresses.

Each PE comparator input can be used:

- To control the ViewInst start/stop logic, as shown in Figure 4-1 on page 4-119.
- To control the Single-shot Comparator controls.
- As an independent trace unit resource, to activate events within the trace unit. See *Selecting trace unit resources on page 4-171*.

For a trace unit for an Armv6-M, Armv7-M or Armv8-M PE, the number of PE comparator inputs is the same as the number of DWT comparators, up to a maximum of 8 PE comparator inputs. If the PE has more than 8 DWT comparators, DWT comparators 0 to 7 map to PE comparator inputs 0-7.

Arm strongly recommends that the PE comparator inputs are synchronous with the instructions executed when these are used by the trace unit. This enables the PE comparator inputs to provide precise filtering capabilities when used with the ViewInst start/stop logic. When using PE comparator inputs to control the ViewInst start/stop logic, Arm strongly recommends using only PE comparators that are programmed for instruction address comparison. From ETMv4.5, for a trace unit for an Armv8-M PE, where the trace unit has PE comparator inputs and does not have any trace unit address comparators, these recommendations apply as rules.

When multiple PE comparisons are performed simultaneously, for example when multiple instructions are executed in a single cycle, Arm strongly recommends that the trace unit treats the PE comparator inputs in program order to ensure predictable behavior of the start/stop logic.

Arm strongly recommends following the rules that are outlined in *Behavior of the start/stop control during a trace run on page 4-121* for handling batches or blocks of instructions that are used with the PE comparator inputs.

Arm strongly recommends that the effects of the PE comparator inputs are tolerant of speculative execution. This means that if an instruction that is used with the ViewInst start/stop logic is then canceled, the effect on the start/stop logic is reversed.

Note

When used with the Single-shot comparator controls, the PE comparator inputs must only fire the Single-shot comparators controls if the instruction is architecturally executed.

TRCIDR4.NUMPC shows how many PE comparator inputs are implemented.

Single-shot controls for address comparators

When a trace unit is exposed to speculative execution, if address comparators are used to activate events in the trace unit, then those events might be activated when speculative execution occurs. This is because:

- As described in *Single address comparators on page 4-149*, a single address comparator might signal a match on speculative execution.

- As described in *Address range comparators on page 4-151*, an address range comparator might signal a match on speculative execution.

In addition, as described in *Data value comparators on page 4-153*, a data value comparator might signal a match on speculative execution. Data value comparators can be used as independent trace unit resources, or can be programmed for use with either single address comparators or address range comparators.

Therefore, if an address comparator is used, for example, to activate a counter or assert an external output, then that counter might become enabled, or that external output might become asserted, as a result of speculative execution.

Single-shot controls for address comparators make it possible for events in the trace unit to be activated based only on nonspeculative execution, that is, only on architectural execution.

An ETMv4 trace unit can provide up to eight single-shot controls. Each control can be used with one or more address comparators.

Note

Arm recommends that at least one single-shot control is implemented when a trace unit implementation contains one or more address comparators.

A single-shot control works in the following way:

1. One or more address comparators are selected by using the TRCSSCCRn for the single-shot control. The selected address comparators can be all single address comparators, all address range comparators, or a combination of both. In addition, each selected address comparator might or might not have a data value comparator that is associated with it.
2. Whenever one of the selected address comparators matches, then when the trace unit knows the status of the instruction or data transfer that has been accessed, that is, whether it has been committed for execution or canceled because of mis-speculation:
   - If the instruction or data transfer has been committed for execution, the single-shot control fires for the duration of one processor clock cycle.
   - If the instruction or data transfer has been canceled because of mis-speculation, the single-shot control does not fire.

   Note

   Conventionally, a single-shot control, as its name suggests, only fires once. However, in the ETMv4 trace unit, a single-shot control can be programmed so that it is reset after every time it has fired.

   For more information, see *Programming a single-shot control to self-reset after it fires on page 4-161*.
3. When the single-shot control is used for instruction address comparisons, it always fires regardless of whether the instruction:
   - Fails its condition code check.
   - Is a failed store-exclusive operation.

   When the single-shot control is used for data address comparisons or data address comparisons with data value comparisons, it does not fire if the instruction:
   - Fails its condition code check.
   - Is a failed store-exclusive operation.

Not every single-shot control can support every type of address comparator. The TRCSSCSRn for each control identifies the type of address comparator that the control can support. The possible address comparator types are:

- Instruction address comparators. If these are supported, both possible uses of instruction address comparators are supported, that is:
  - Single address comparators.
  - Address range comparators.

- Data address comparators. If these are supported, both possible uses of data address comparators are supported, that is:
  - Single address comparators.
  - Address range comparators.
- Data address comparators that have associated data value comparators. Both possible uses are supported:
  - Single address comparators with associated data value comparators.
  - Address range with associated data value comparators.

Single-shot controls can be used as a trace unit resource, to activate trace unit events. For example, a single-shot 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 control can therefore, if programmed to assert an external output, be used to indicate to a trace analyzer that a particular instruction or a particular data transfer has been committed for execution. This means that a trace analyzer can start or stop trace capture that is based on the architectural execution of that instruction or data transfer.

Note

When a single-shot control is used to activate a trace unit event, the event might not become activated until some time after the trace unit has traced the instruction or data transfer. This is because although the trace unit traces the instruction or data transfer as it is executed, the PE might not confirm whether the instruction or data transfer was architecturally executed or canceled because of mis-speculation until some time later, and therefore the single-shot control might not fire until some time later.

For a trace unit with one or more PE comparator inputs, the Single-shot comparator controls can be programmed to use the PE comparator inputs. If the Single-shot comparator control supports the use of those PE comparator inputs, the implementation must ensure that the Single-shot comparator controls only fire when the instruction or data transfer which caused the PE comparator input to fire is architecturally executed.

For PE comparator inputs which are performing comparisons against data addresses or data values, it is IMPLEMENTATION DEFINED whether the Single-shot comparator control fires if any of the following occur:

- The instruction fails its condition code check.
- The instruction is a failed store-exclusive operation.

For a trace unit which does not implement data address or data value tracing, if a PE comparator input which performs comparisons against data addresses or data values is used with a Single-shot comparator control then the behavior of the Single-shot comparator control is implementation specific.

A trace unit for an Armv6-M, Armv7-M, or Armv8-M PE that is tracing a partially executed instruction that has been interrupted and continues later, the following rules apply:

- For an instruction address comparison, the Single-shot comparator control only fires when the last portion of the instruction is successfully executed.
- For a data address or data value comparison, the Single-shot comparator control fires if that data transfer is successfully performed and the ICI bits are set to indicate that the transfer is not repeated.

Programming a single-shot control to self-reset after it fires

As mentioned in *Single-shot controls for address comparators on page 4-159*, an ETMv4 trace unit provides up to eight single-shot controls for address comparators.

For each control, TRCSSCCRn.RST can be programmed so that the control either:

- Only fires once, so that after it has fired, it never fires again.
- Resets after every time it fires, so that it can fire again when a selected address comparator next signals an address match for an instruction or data transfer that is architecturally executed.

If a control is programmed to reset after every time it fires:

- For each successful instruction address or data address match, the control must only fire for a maximum of one processor clock cycle.
- If multiple address matches occur in close succession, for example, if more than one of the address comparators that are selected signal an address match simultaneously, then not all these address matches are guaranteed to cause the control to fire. Only the first address match definitely causes the control to fire.

Using data address comparators and data value comparators to detect store-exclusive transfers

All store-exclusive instruction types comprise two parts:

- The data that is stored to memory.
- An indication of whether the data store is successful.

A single address comparator or an address range comparator, when programmed to match on a data address, can be used to compare against success indicators of store-exclusive instructions. This is possible because, as described in *Data trace behavior on tracing store-exclusive instructions on page 2-77*, each success indicator is treated as a data transfer with the following properties:

- A write access, meaning that a data address comparator can be programmed by setting TRCACATRn.TYPE to `0b10` so that it performs comparisons only on data store addresses.
- The access address is the same as the address of the lowest byte of the memory access, meaning that the address is known, and therefore the TRCACVRn for the data address comparator can be populated.
- The access size of a success indicator is considered to be the same as the size of the data transfer performed, up to a 64-bit transaction.

In addition, data value comparators can be programmed to compare against the values of success indicators. That is, a data value comparator can be programmed to match on either:

- A success indicator value of 0, that indicates a successful data store.
- A success indicator value of 1, that indicates an unsuccessful data store.

In this way, data value comparators can indicate whether store-exclusive transfers are successful or not.

This means that data value comparators can be used with data address comparators. For example, a data value comparator might be used with a data address comparator to indicate only successful store-exclusive transfers to a particular memory address. This can be done by:

- Programming the data address comparator so that it matches only if the data value comparator matches. This can be done by setting TRCACATRn.DATAMATCH to `0b01`.
- Programming the data address comparator with the access address of the success indicator. This is the same as the lowest addressed byte of the memory access.
- Programming the data value comparator to match on a successful data store.

Data trace behavior on tracing store-exclusive instructions on page 2-77 includes some examples of data transfers for store-exclusive instructions.

4.3 Accessing the trace unit

An ETMv4 trace unit 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:

- An external debugger, using the *Arm Debug Interface v5* (ADIv5).
- A memory-mapped interface.
- System instructions. This is also known as coprocessor access.

Not all registers are accessible using all access interfaces. In addition, a trace unit might not implement all the interfaces, however:

- An implementation must support external debugger access.
- To provide for accesses by on-chip software, Arm recommends that support for at least one of the following is implemented:
  - Memory-mapped access.
  - Access by system instructions.
- If the PE implements Armv8.4-Trace:
  - System instruction access must be implemented.
  - Memory-mapped access is deprecated.

If accesses occur simultaneously from multiple access mechanisms then the behavior must be as if all accesses occurred atomically in any order.

The remainder of this section is organized as follows:

- *Register map*.
- *About accesses to registers in different trace unit power domains on page 4-164*.
- *Effect of the DBGSWENABLE signal on accesses to trace unit registers on page 4-165*.
- *Use of the trace unit main enable bit on page 4-165*.
- *External debugger and memory-mapped access on page 4-167*.
- *System instructions on page 4-168*.
- *Synchronization of register updates on page 4-169*.

4.3.1 Register map

An ETMv4 trace unit provides a total of 1024 registers. Each register is either:

- A trace unit *management register*. Most trace unit management registers are located in the trace unit debug power domain. However, two management registers, the TRCOSLAR and the TRCOSLSR, are located in the trace unit core power domain. TRCDEVID, TRCAUTHSTATUS, and TRCDEVARCH are accessible in both the trace unit core power domain, and the trace unit debug power domain. For more details on the accessibility of these registers, see *Access permissions on page 7-340*.
- A trace unit *trace register*. All trace registers are located in the trace unit core power domain.

This power domain split is shown in Figure 3-2 on page 3-92.

When the Unified Power Domain Model is implemented, all registers are located in the trace unit core power domain.

In the register map, the registers are organized into 32 blocks according to function. Each block contains 32 registers, that are either all 32-bit registers or all 64-bit registers. Most blocks contain either all management registers or all trace registers.

Table 4-12 shows the register map.

Table 4-12 Register map overview

Block number	Registers[a]	Bit width	Type of register	Function
0	0-31	32	Trace	Main control
1	32-63	32	Trace	Trace filtering controls
2	64-95	32	Trace	Derived resources
3	96-127	32	Trace	IMPLEMENTATION DEFINED registers and the ID Registers
4	128-159	32	Trace	Resource Selection Control Registers
5	160-191	32	Trace	Single-Shot Comparator Control and Status registers
6	192-223	32	Management	OS Lock and PowerDown registers
7	224-255	-	-	Reserved
8	256-287	64	Trace	Address Comparator Value Registers
9	288-319	64	Trace	Address Comparator Access Type Registers
10	320-351	64	Trace	Data Value Comparator Value Registers
11	352-383	64	Trace	Data Value Comparator Mask Registers
12	384-415	64	Trace	Context ID Comparator Value Registers and Virtual context identifier Comparator Value Registers
13	416-447	32	Trace	Context ID Comparator Control Registers and Virtual context identifier Comparator Control Registers
14-28	448-927	-	-	Reserved
29	928-959	-	-	Reserved for IMPLEMENTATION DEFINED integration and topology detection registers
30-31	960-1023	32	Management[b]	CoreSight management registers

a. For external debugger or memory-mapped accesses, the address offset of the register is the register number that is multiplied by four.
b. The TRCCLAIMSET and TRCCLAIMCLR registers, that are in block 31, are considered to be trace registers, not management registers.

The registers in block numbers 7, 14-28, and 29, are each treated as either:

- A reserved management register.
- A reserved trace register.

For block number 7 and block numbers 14-28, whether a register is treated as a reserved management register or a reserved trace register is IMPLEMENTATION SPECIFIC.

For block number 29, whether a register is treated as a reserved management register or a reserved trace register is IMPLEMENTATION DEFINED. However, if any registers in block 29 are not implemented, then it is IMPLEMENTATION SPECIFIC whether they are treated as either a reserved management register or a reserved trace register.

4.3.2 About accesses to registers in different trace unit power domains

As mentioned in *Register map on page 4-163*:

- Most of the trace unit management registers are located in the trace unit debug power domain, with the exception of the TRCOSLAR and the TRCOSLSR.
- All trace unit trace registers are located in the trace unit core power domain.

The trace unit trace registers are inaccessible when the trace unit core power domain is powered down.

The trace unit management registers are always accessible to an external debugger, except for the TRCOSLAR and the TRCOSLSR. The TRCOSLAR and the TRCOSLSR are both included in block six of the register map. TRCDEVID, TRCAUTHSTATUS, and TRCDEVARCH are accessible in both the trace unit core power domain, and the trace unit debug power domain. For more details on the accessibility of these registers, see *Access permissions on page 7-340*.

For more information about the trace unit power domains, see *Trace unit power domains on page 3-91*.

4.3.3 Effect of the DBGSWENABLE signal on accesses to trace unit registers

ADIv5 defines a signal, **DBGSWENABLE**, that can be used to disable all memory-mapped accesses to the trace unit registers. See the *Arm Debug Interface v5 Architecture Specification*.

4.3.4 Use of the trace unit main enable bit

The trace unit trace registers must only be programmed when the trace unit is disabled and TRCSTATR.IDLE indicates that the trace unit is idle. The exceptions to this rule are:

- TRCPRGCTLR can be accessed and programmed regardless of whether the trace unit is enabled or disabled.
- The TRCCLAIMSET and TRCCLAIMCLR registers can be programmed regardless of whether the trace unit is enabled or disabled. These registers are both always implemented and are used by software to co-ordinate application and debugger access to the trace unit.

Whenever the trace unit is enabled or TRCSTATR.IDLE indicates that the trace unit is not idle, writes to all other trace unit trace registers might be ignored.

For more information, see:

- *Trace unit behavior when the trace unit is enabled on page 3-100*.
- *Trace unit behavior when the trace unit is disabled on page 3-100*.
- *Access permissions on page 7-340*.

Figure 4-17 on page 4-166 shows the procedure that must be used when programming the trace unit registers.

Note

The PE does not have to be in Debug state to program the trace unit trace registers.

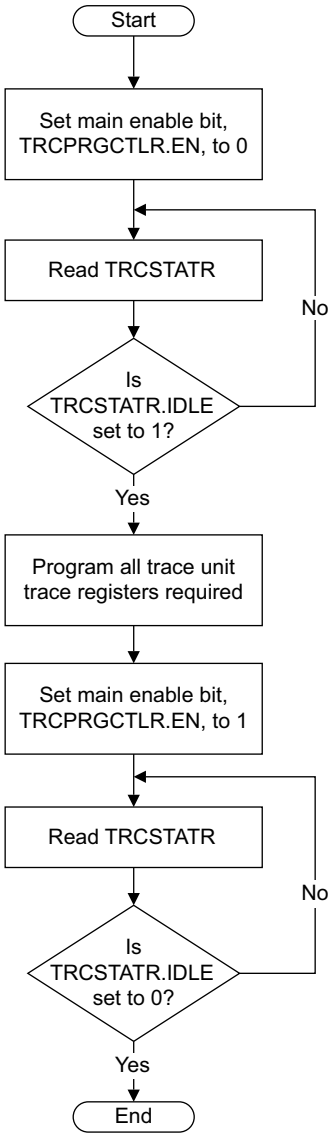


Figure 4-17 The procedure for programming the trace unit trace registers

When programming the trace unit, it is important to be aware that the loops that poll TRCSTATR in Figure 4-17 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.

Note

- Arm recommends that the trace unit can leave the idle state and can reenter the idle state regardless of whether non-invasive debug is enabled or disabled. See *Trace unit behavior when tracing is prohibited on page 3-108* for further details on the behavior of the trace unit when non-invasive debug is disabled.
- The Arm architecture provides a guarantee of observability for certain system registers. None of the ETMv4 System registers provide such a guarantee of observability and therefore require explicit synchronization when accessed using system instructions. In particular, whenever disabling or enabling the trace unit, a poll of TRCSTATR needs explicit synchronization between each read of TRCSTATR.

4.3.5 External debugger and memory-mapped access

The address maps for the external debugger and memory-mapped interfaces are identical, except that some registers that are available to an external debugger are not available to memory-mapped access, and some registers that are available to memory-mapped access are not available to an external debugger.

The memory access sizes that are supported by any peripheral are IMPLEMENTATION DEFINED by the peripheral. For access to the trace unit registers, implementations must support:

- 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® APB3 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.

All registers are only single-copy atomic at word granularity.

The following accesses are not supported and have CONSTRAINED UNPREDICTABLE behavior:

- 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.

This CONSTRAINED UNPREDICTABLE behavior can be 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 UNKNOWN. This is the Arm preferred behavior.

Note

For accesses from the memory-mapped interface, if TRCLSR.SLK is set to 1, meaning the Software Lock is implemented and locked, writes to the registers other than TRCLAR are ignored, including in the cases that are outlined in this section.

For accesses from the external debugger interface, the size of an access is determined by the interface. In an ADIv5 compliant Memory Access Port, MEM-AP, this is specified by the MEM-AP CSW register.

Note

The standard CoreSight APB-AP supports only word accesses.

4.3.6 System instructions

Access by system instructions is also known as coprocessor access.

System instructions for Armv7 and Armv8 AArch32

For Armv7-A, Armv7-R, Armv8-R and Armv8 AArch32, the instructions that are used to read from and write to the trace unit registers are as follows:

```
MRC <p14>, 1, <Rd>, CRn, CRm, opc2
```

```
MCR <p14>, 1, <Rd>, CRn, CRm, opc2
```

The trace unit register number, Reg[9:0], is formed from `CRn`, `CRm`, and `opc2`:

- Reg[9:7] = `CRn`[2:0]
- Reg[6:4] = `opc2`[2:0]
- Reg[3:0] = `CRm`[3:0]

Instructions with `CRn` ≥ `0b1000` are UNDEFINED.

When the PE is in User mode, all accesses are UNDEFINED.

System instructions for Armv8 AArch64

For Armv8 AArch64, the instructions that are used to read from and write to the trace unit registers are as follows:

```
MRS <Rd>, <trace register>
```

```
MSR <trace register>, <Rd>
```

`op0` is always `0b10`, `op1` is always `0b001`.

The `<trace register>` number, Reg[9:0], is formed from `CRn`, `CRm`, and `opc2`:

- Reg[9:7] = `CRn`[2:0].
- Reg[6:4] = `opc2`[2:0].
- Reg[3:0] = `CRm`[3:0].

Instructions with `CRn` ≥ `0b1000` are UNDEFINED.

When the PE is in EL0, all accesses are UNDEFINED.

64-bit support

When accessing the registers using system instructions, some register locations are 32-bits wide and some are 64-bits wide.

When a 64-bit access is used to access a 32-bit register, the upper 32 bits of the access behave as RES0H.

A read from a 64-bit register location using a 32-bit access returns only the lower 32 bits of the register. A write to a 64-bit location using a 32-bit access leaves the upper 32 bits of the register unchanged.

For 64-bit registers:

- From AArch32, two system instructions are provided, one to access each half of the 64-bit register. For a register <n>, the system instruction using the encoding to access register <n> accesses the lower 32 bits of the register, and the system instruction using the encoding to access register <n+1> accesses the upper 32-bits of the register.
- From AArch64, a single system instruction is provided to access all 64-bits of the register. For a register <n>, the system instruction using the encoding to access register <n> accesses the entire register, and the system instruction using the encoding to access register <n+1> is Reserved.

To access both halves of a 64-bit register location when using 32-bit accesses, two 32-bit accesses are required, each to a consecutive 32-bit register address. For example, register number 256 is a 64-bit address comparator register. To read this register, two 32-bit read accesses are required, one to register number 256, and the other to register number 257. Similarly, to write to this register, two writes are required, one to register number 256, and the other to register number 257. This is because within a trace unit, 64-bit registers are made up of two 32-bit registers.

Note

To access a 64-bit register from AArch32 state, system instructions to access both halves of the register are always implemented. If there are no bits implemented in the upper 32 bits of the register, accesses to the upper 32-bits are either RES0 or RAZ/WI, depending on the register description. For more information, see *Trace unit behavior on accesses to reserved trace unit registers and fields on page 7-344*.

4.3.7 Synchronization of register updates

Software running on the PE can program the trace unit registers by using either:

- System instructions, if the interface is implemented.
- The memory-mapped interface, if it is implemented.

Which interfaces are implemented is IMPLEMENTATION DEFINED.

Synchronization when using system instructions to program the trace unit

When using system instructions to program the trace unit, the trace analyzer must be aware that any changes to trace unit registers are guaranteed to be visible to subsequent execution only after a Context synchronization event, which consists of one of the following:

- Taking an exception.
- Returning from an exception.
- Performing an ISB operation.

On an Armv8-A and Armv8-R PE, the following events are also Context synchronization event:

- Exit from Debug state.
- A `DCPS` instruction in Debug state.
- A `DRPS` instruction in Debug state.

However, the following rules apply to trace unit register accesses using system instructions:

- When a system instruction directly reads a register using the same register number as was used by a system instruction to write it, the system instruction is guaranteed to observe the value that is written, without requiring any context synchronization between the write and read instructions.
- When a system instruction directly writes a register using the same register number as 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.

This is important when changing the value of the main enable bit in the TRCPRGCTLR. After writing to the TRCPRGCTLR to change the value of the main enable bit, the trace analyzer must make at least one read of the TRCSTATR before programming any other registers. The trace analyzer must perform a Context synchronization event between writing to the TRCPRGCTLR and reading the TRCSTATR.

Note

- These rules mean that when setting the claim tags using TRCCLAIMSET, a Context synchronization event must be executed before reading back the claim tag state using TRCCLAIMCLR.
- The Arm architecture provides a guarantee of observability for certain system registers. None of the ETMv4 system registers provide such a guarantee of observability and therefore require explicit synchronization when accessed using system instructions. In particular, whenever disabling or enabling the trace unit, a poll of TRCSTATR needs explicit synchronization between each read of TRCSTATR.

Arm recommends that the trace analyzer always executes an ISB instruction after programming the trace unit registers, to ensure that all updates are committed to the trace unit before normal code execution resumes.

Synchronization when using the memory-mapped interface

When using the memory-mapped interface to program the trace unit, the trace analyzer must be aware that a DSB operation causes all writes to memory-mapped trace unit registers that appear before the DSB in program order to be completed.

However, the following rules apply to memory-mapped trace unit register accesses:

- When a load operation directly reads a register using the same address as was used by a store operation to write it, the load is guaranteed to observe the value that is written, without requiring any context synchronization between the store and the load.
- When a store operation directly writes a register using the same address as was used by a previous store operation to write it, the final result is the value of the second store, without requiring any context synchronization between the two stores.

Arm recommends that the trace analyzer always executes a DSB instruction followed by an ISB instruction after programming the trace unit registers, to ensure that all updates are committed to the trace unit before normal code execution resumes.

Some memory-mapped trace unit registers are not idempotent for reads or writes. Therefore, the region of memory that is occupied by the trace unit registers must not be marked as Normal memory, because the Memory Order Model permits accesses to Normal memory locations that are not appropriate for such registers.

For Armv8-A, Armv8-R, and Armv8-M PEs, the region of memory that is occupied by the trace unit registers must be marked as Device-nGRE, or more strongly constrained.

For Armv6-M and Armv7 PEs, the region of memory that is occupied by the trace unit registers must have the Strongly-ordered or Device attribute.

Synchronization between register updates that are made through the external debug interface and updates that are made by software running on the PE is IMPLEMENTATION DEFINED. However, if the external debug interface is implemented through the same port as the memory-mapped interface, then updates made through the external debug interface have the same properties as updates made through the memory-mapped interface.

4.4 Selecting trace unit resources

An ETMv4 trace unit has a range of resources. See *Trace unit resources on page 4-139*. Some of these resources can be selected for activating *trace unit events*. The resources that can be used to activate events are:

- Counters. All counters in an ETMv4 trace unit are decrement counters, so that events can be triggered on a counter reaching zero. See *Counters on page 4-139*.
- A sequencer state machine that can have up to four states. See *Sequencer on page 4-144*.
- External inputs. The ETMv4 architecture supports the implementation of between zero and 256 external inputs to a trace unit. See *External inputs on page 4-146*.
- Memory access resources. These are:
  - Single address comparators, that can each be programmed to match on either an instruction address or a data address. See *Single address comparators on page 4-149*.
  - Address range comparators. Each address range comparator is formed from two single address comparators, so that an address range comparator matches on any access that is in a programmed range of addresses. Each address range comparator can be programmed to match on either instruction addresses, or data addresses. See *Address range comparators on page 4-151*.
  - Data value comparators. Single address comparators can be used with data value comparators, so that a single address comparator only matches if both the address and data value match. Similarly, address range comparators can be used with data value comparators, so that an address range comparator only matches if both the address and data value match. See *Data value comparators on page 4-153*.
  - Context ID comparators. A Context ID comparator matches whenever the PE is executing with the Context ID that the comparator is programmed with. Both single address and data address comparators can be used with Context ID comparators. See *Context ID comparators on page 4-157*.
  - Virtual context identifier comparators. A Virtual context identifier comparator matches whenever the PE is executing with the Virtual context identifier that the comparator is programmed with. Both single address and data address comparators can be used with Virtual context identifier comparators. See *Virtual context identifier comparators on page 4-158*.
  - PE comparator inputs. The ETMv4 architecture supports up to eight inputs that can be driven from comparators within the PE. See *PE comparator inputs on page 4-159*.
  - Single-shot controls for address comparators. Single-shot controls can be used with single address or address range comparators. Whenever the address comparator matches, if the instruction or data transfer is architecturally executed rather than speculatively executed, the single-shot control fires. See *Single-shot controls for address comparators on page 4-159*.

A *trace unit event* might be one of the following:

- The ViewInst filtering function or the ViewData filtering function becoming active. The resources that can activate these events include:
  - Single address comparators.
  - Address range comparators.
  - PE comparator inputs.
  - An imprecise enabling input.

  See Figure 4-1 on page 4-119 and Figure 4-8 on page 4-132.
- The assertion of an external output. The ETMv4 architecture supports the tracing of between one and four arbitrary events, numbered from 0-3. The TRCIDR0.NUMEVENT shows how many events are supported. When a particular numbered event occurs, the trace unit asserts the corresponding external output. See *External outputs on page 4-146*.

  The TRCEVENTCTL0R can be used to choose the trace unit resources that activate each event. Any type of trace unit resource can be chosen.

  If the bit in TRCEVENTCTL1R.INSTEN that corresponds to a particular event is set to 1, then the trace unit generates an Event element in the instruction trace stream when that event occurs. The Event element contains the number of the event that occurred.

If TRCEVENTCTL1R.DATAEN==1, then the trace unit generates an Event element in the data trace stream when event 0 occurs.

- A counter reaching zero. In this case, for example, the resource that activates the event might be a single address comparator. The output of the single address comparator might be wired to the reload input of the counter, so that whenever an address match occurs, the counter is reloaded and, provided that the counter is also enabled, begins to decrement.
- A change of sequencer state. In this case, for example, the resource that activates the event might be a counter, so that whenever the counter reaches zero, the sequencer changes state.

Two of the trace unit resource types, the counters and the sequencer, can also produce trace unit events, because a counter reaching zero is an event and the current sequencer state is an event. This means that these particular resources can be used to construct circular dependencies. That is, counter and sequencer events can also be used as resources, that in turn can activate other trace unit events. For example:

1. A single address comparator might be used to decrement a counter. In this case, the single address comparator is the resource and the change in the value of the counter is the event.
2. The counter reloads when the sequencer enters state 1. In this case, the sequencer changing state is the resource and the counter being reloaded is the event.
3. The sequencer moves to state 1 when the counter is at zero. In this case, the sequencer changing state is the event and the value of the counter being equal to zero is the resource.
4. The sequencer moves to state 0 when another event occurs.

Figure 4-18 shows a high-level view of the connections between the resource selectors, resources, and events.

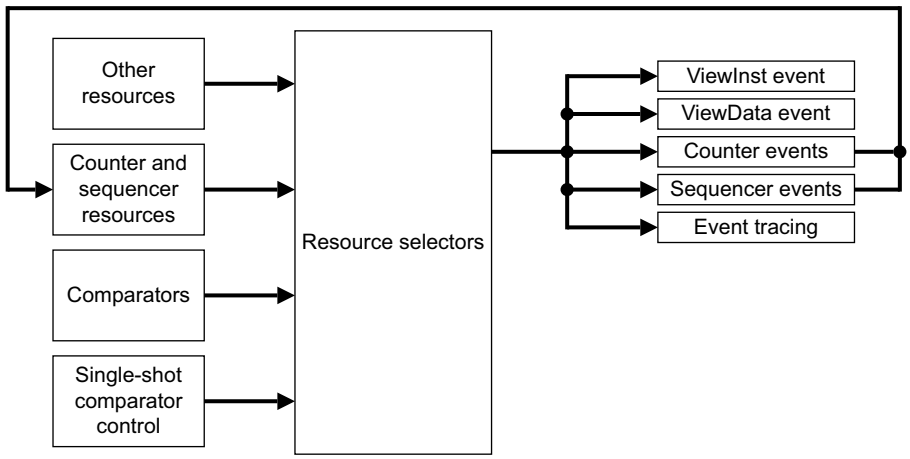


Figure 4-18 Summary of resource selection

The remainder of this section is organized as follows:

- *Grouping of trace unit resources on page 4-173*.
- *Selecting a trace unit resource or a pair of trace unit resources on page 4-173*.
- *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177*.
- *About the timing of events that are activated by trace unit resources on page 4-178*.
- *About the behavior of events on disabling the trace unit on page 4-178*.
- *Example of resource behavior when disabling the trace unit on page 4-179*.

4.4.1 Grouping of trace unit resources

The trace unit resources are organized into 16 logical groups. Each group contains up to 16 single resources, as Table 4-13 shows.

Table 4-13 Resource grouping

Group	Resource number	Resource
0b0000	0-3	When TRCIDR5.NUMEXTINSEL != zero, group 0b0000 with Resource number 0-3 selects External input selector 0-3. When TRCIDR5.NUMEXTINSEL == zero and TRCIDR5.NUMEXTIN != zero, group 0b0000 with Resource number 0-3 directly selects External input 0-3. Otherwise no External inputs are implemented.
	4-15	-[a]
0b0001	0-7	PE comparator inputs 0-7
	8-15	-[a]
0b0010	0-3	Counters at zero 0-3
	4-7	Sequencer states 0-3
	8-15	-[a]
0b0011	0-7	Single-shot comparator control 0-7
	8-15	-[a]
0b0100	0-15	Single address comparators 0-15
0b0101	0-7	Address range comparators 0-7
	8-15	-[a]
0b0110	0-7	Context ID comparators 0-7
	8-15	-[a]
0b0111	0-7	Virtual context identifier comparators 0-7
	8-15	-[a]
0b1000-b1111	0-15	-[a]

a. Reserved.

4.4.2 Selecting a trace unit resource or a pair of trace unit resources

A trace unit resource is selected by using a trace unit *resource selector*.

Each resource selector uses one of the 32 TRCRSCTLRn registers. Trace unit resource selectors are implemented in the trace unit in pairs, so that a maximum of 16 pairs can be implemented.

The minimum number of resource selectors that are implemented depends on the ETM architecture version:

- PEs that comply with ETMv4.3 or later are permitted to have no resource selectors, or one or more pairs of resource selectors implemented.

- PEs that comply with ETMv4.2 or earlier have at least one pair of resource selectors implemented.

For all versions, if resource selectors are implemented, the lowest pair comprises:

- Resource selector 0, which always provides a FALSE result.
- Resource selector 1, which always provides a TRUE result.

TRCIDR4.NUMRSPAIR shows how many pairs of resource selectors are implemented.

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 4-21 on page 4-176.

Each TRCRSCTLRn register from TRCRSCTLR2 to TRCRSCTLR31 has at least the following RW fields:

- A 4-bit GROUP field to select a resource group. See Table 4-13 on page 4-173 for resource groups.
- A 16-bit SELECT field to select the resource numbers within that group. Each bit in the 16-bit field corresponds to a resource number in the selected group. See Table 4-13 on page 4-173. If more than one resource in the group is selected, the result is a logical OR of the outputs of the selected resources.
- A 1-bit INV field to optionally invert the output of the selected resource or the logical OR result of the selected resources. If this bit is set to 1, the output of the selected resource, or the logical OR result of the selected resources, is inverted.

In addition, if the TRCRSCTLRn register is the lower register for a pair of resource selectors, then it has an additional 1-bit RW field, PAIRINV, to optionally invert the logical result of the Boolean function that is applied to the combination of selected resources. See Figure 4-21 on page 4-176 and Table 4-14 on page 4-176. For example:

- TRCRSCTLR2 and TRCRSCTLR3 might constitute a resource selection pair. In this case:
  - TRCRSCTLR2 is the lower register. This has the additional 1-bit RW field, PAIRINV, to optionally invert the result of the Boolean function that is applied to the outputs of the combination of selected resources.
  - TRCRSCTLR3 is the upper register. In this register, PAIRINV is RES0.

This means that, when a resource selection pair is used, the following scenario is possible:

- One TRCRSCTLRn might select only one resource within the group.
- The other TRCRSCTLRn 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 4-19 on page 4-175 shows this.

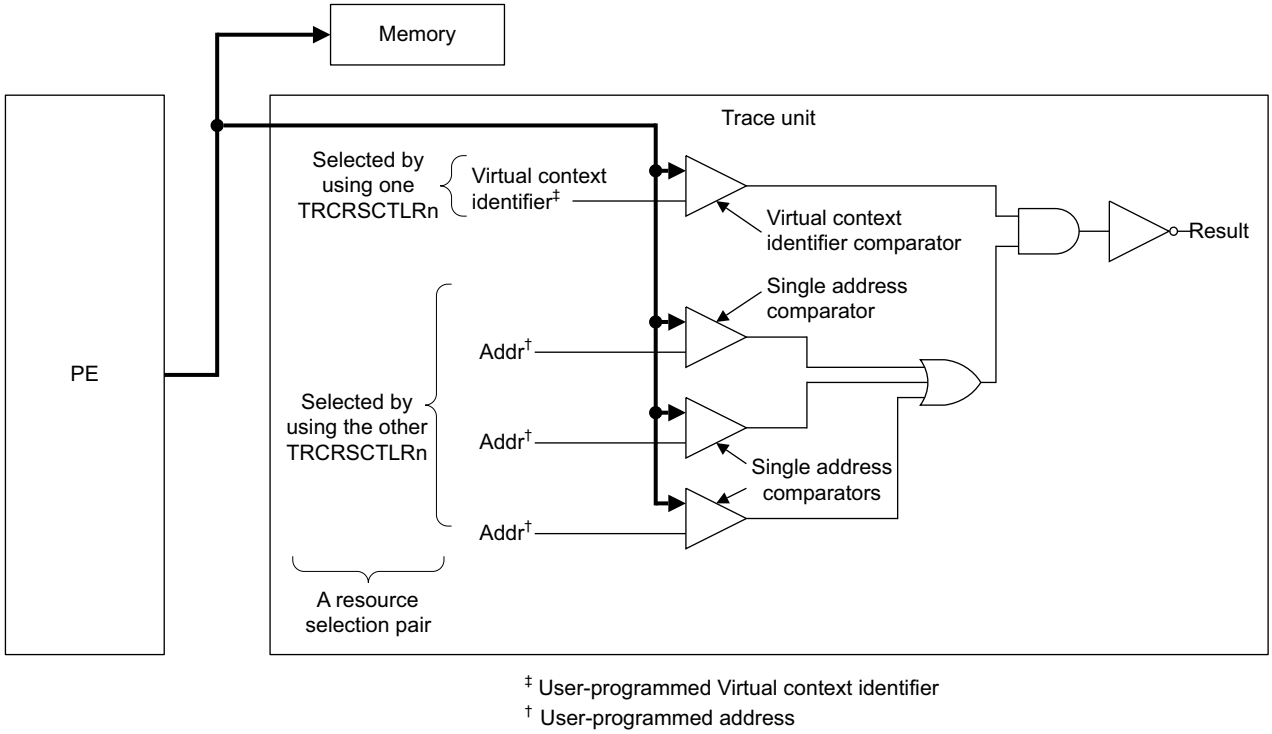


Figure 4-19 An example of resource selection using a resource selection pair

Figure 4-20 shows a single resource selector.

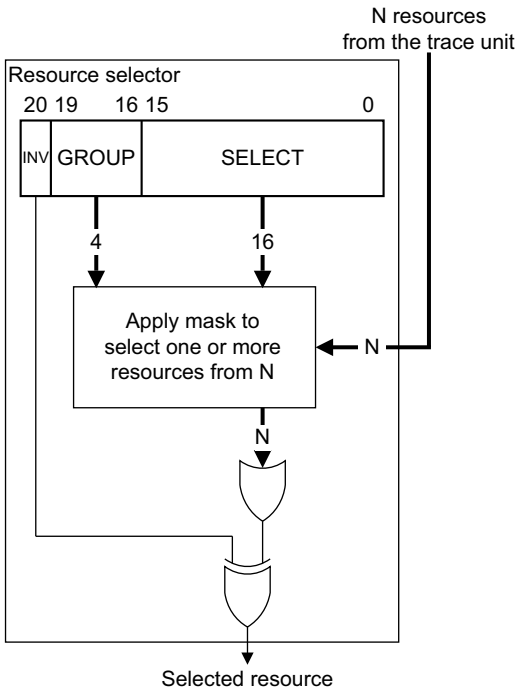


Figure 4-20 A single resource selector

Figure 4-21 on page 4-176 shows a resource selection pair.

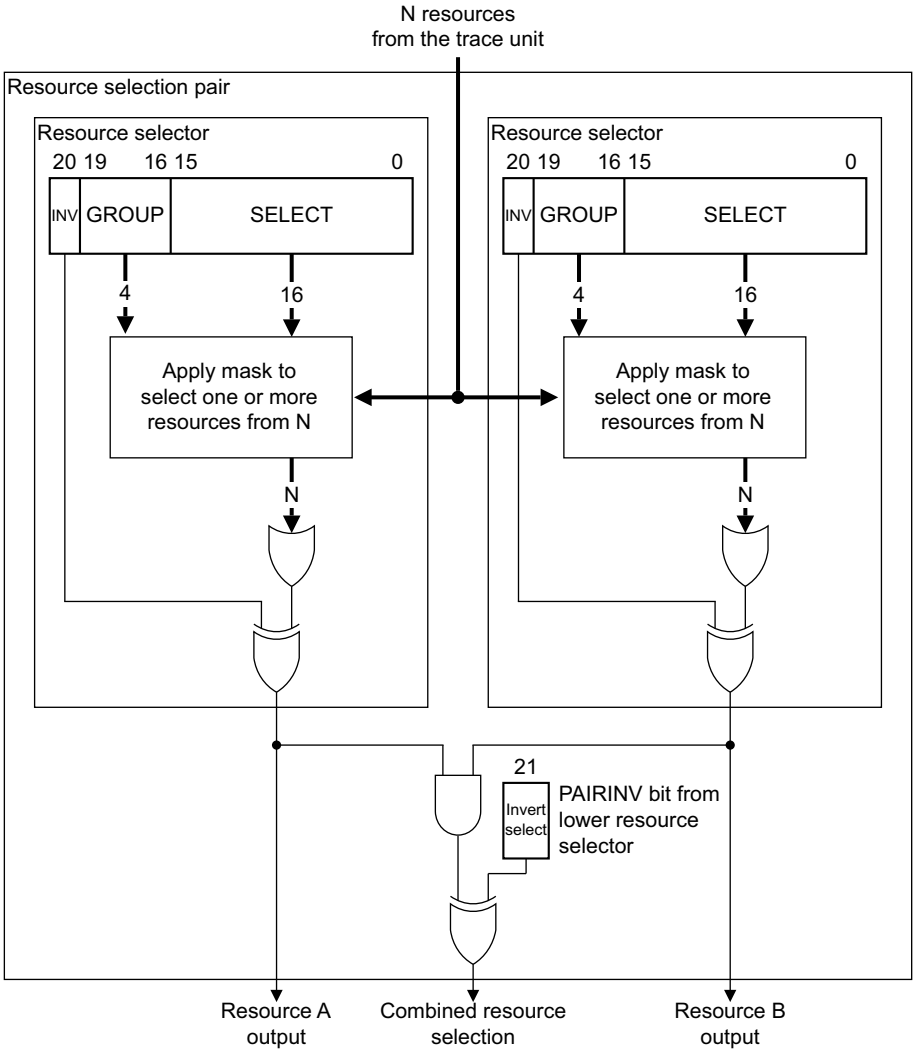


Figure 4-21 A resource selection pair

In Figure 4-21, the Boolean function is selected by using the single-bit inverters, INV, for each resource selector, with the inverter that is applied to the pair, PAIRINV, as shown in Table 4-14.

Table 4-14 Selecting a Boolean function

Function	Resource A INV	Resource B INV	PAIRINV
A AND B	0	0	0
NOT(A) OR NOT(B)	0	0	1
Reserved	0	1	0
NOT(A) OR B	0	1	1
NOT(A) AND B	1	0	0
Reserved	1	0	1
NOT(A) AND NOT(B)	1	1	0
A OR B	1	1	1

4.4.3 Activating a trace unit event with a selected trace unit resource or pair of trace unit resources

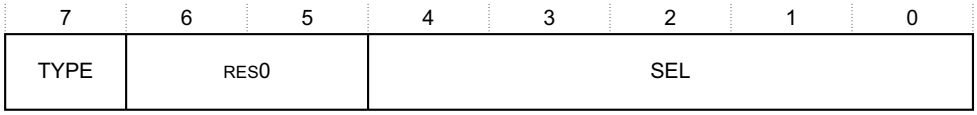
Each trace unit event that is listed in *Selecting trace unit resources on page 4-171* can be activated by a trace unit resource. Each event uses an 8-bit event select field to select a trace unit resource. These 8-bit fields are present in:

- The TRCEVENTCTL0R. This register contains up to four EVENT fields, and the events that use these fields are numbered from 0-3. Whenever a resource selected by one of these fields becomes active, the trace unit:
  - Asserts the corresponding external output.
  - Might generate an Event element, that contains the relevant number, in the instruction trace stream. This occurs if TRCEVENTCTL1R.INSTEN contains a 1 in the bit corresponding to this event.
  - Might generate an Event element in the data trace stream. This occurs if the activated event is event 0, and TRCEVENTCTL1R.DATAEN==1.
- The TRCTSCTLR. This register contains one EVENT field. Whenever the resource selected by this field becomes active, the trace unit inserts a global timestamp into the trace streams.

  Note

  Global timestamps are automatically inserted into the trace streams at other useful points. For more information, see *Global timestamping on page 2-82*.
- The TRCVICTLR. This register contains one EVENT field. Whenever the resource selector selected by this field becomes active, the ViewInst function can become active if other conditions become active. See *Overview of the ViewInst function on page 4-119*.
- The TRCVDCTLR. This register contains one EVENT field. Whenever the resource selector selected by this field becomes active, the ViewData function can become active if other conditions become active. See *Overview of the ViewData function on page 4-132*.
- The TRCSEQEVRn. These registers contain one B and one F event select fields. Whenever the resource selected by these fields becomes active, the sequencer state moves either backwards or forwards.
- The TRCSEQRSTEVR. This register contains one RST event select field. Whenever the resource selected by this field becomes active, the sequencer state moves to state 0.
- The TRCCNTCTLRn. These registers contain one RLDEVENT and one CNTEVENT event select field. Whenever the resource selected by these fields becomes active, the count either reloads or decrements.

The bit assignments for an 8-bit EVENT field are:



TYPE, bit[7] Chooses the type of selected resource:

0 A single resource that has been selected using one TRCRSCTLRn register.

1 A Boolean-combined pair of resources that have been selected using two TRCRSCTLRn registers. That is, a Boolean-combined pair of resources that have been selected using a resource selection pair.

Bits[6:5] RES0.

SEL, bits[4:0] Chooses the selected resource number, based on the value of TYPE:

TYPE==0 Bits[4:0] of SEL are used to select the number of the single resource, from 0-31, used to activate the event.

TYPE==1 Bits[3:0] of SEL are used to select the number of the resource selection pair, from 0-15, that has a Boolean function that is applied to it whose output is used to activate the event. If an unimplemented resource is selected using the SEL field, the behavior of the event is UNPREDICTABLE, and the event might fire or might not fire.

Note

- Some bits of SEL might not be RW bits. For example, an implementation might only contain eight pairs of resource selectors. In this case, SEL is only required to be four bits wide, therefore bit[4] might not be a RW bit.
- The top-most bit of SEL must always be zero when TYPE is 1. For example:
  - If 16 pairs of resource selectors are implemented, bit[4] of SEL is 0 when TYPE is 1.
  - If eight pairs of resource selectors are implemented, bit[4] might not be a RW bit, and bit[3] is 0 when TYPE is 1.
- When TYPE is 1, programming SEL to all zeros results in CONSTRAINED UNPREDICTABLE behavior of the event. The event might be active or inactive at any time.

4.4.4 About the timing of events that are activated by trace unit resources

To help meet performance requirements, the ETMv4 architecture permits a trace unit implementation to pipeline the resource selection logic. However, this might affect the behavior of the trace unit, as follows:

- When using a trace unit resource to activate a trace unit event:
  - The time when the event becomes activated might be significantly later than the time when the resource became active. For example, a Context ID comparator resource might be used to activate a change of sequencer state event. In this case:
    1. The PE changes the Context ID that it is executing with. This triggers the Context ID comparator resource to become active.
    2. Because of pipelining in the trace unit resource selection logic, some time passes.
    3. The sequencer state machine changes state.
- When using counter or sequencer events as trace unit resources to activate other events:
  - The time when the other event becomes activated might be significantly later than the time when the original counter or sequencer event became activated. For example, a counter at zero event might be used as a trace unit resource to reload another counter. The first counter might be counter 2, and the second counter might be counter 3. In this case:
    1. Counter 2 reaches zero. This event is propagated back as a trace unit resource. See Figure 4-18 on page 4-172.
    2. Because of pipelining in the trace unit resource selection logic, some time passes.
    3. Counter 3 is reloaded.

The delay that is caused by pipelining is one of the reasons why controlling the ViewInst function by using its enabling event input, or controlling the ViewData function by using its enabling event input, might be imprecise. See *Filtering instruction tracing by using the enabling event on page 4-126* and *Tracing data transfers by using the enabling event on page 4-134*.

The depth of any pipelining that is implemented, and therefore the time that is taken for a resource change of state to propagate through the pipeline, is IMPLEMENTATION SPECIFIC.

4.4.5 About the behavior of events on disabling the trace unit

As described in *Trace unit behavior when the trace unit is disabled on page 3-100*, an ETMv4 trace unit can be disabled by either:

- Setting the main enable bit, TRCPRGCTLR.EN, to 0.
- Locking the OS Lock, by setting TRCOSLAR.OSLK to 1.

This disables both resources and events.

Whenever a trace unit is disabled, it performs the following procedure:

1. All resources that are selected to activate events, except for the sequencer and any counters, are driven low as inputs to the resource selection 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 selection logic.

3. The states of the counters and the sequencer are propagated through the resource selection logic one more time. That is, the state of the counters and the sequencer are propagated through the resource selection logic for the length of time that it takes for the state of a resource to be propagated through the resource selection logic.

This procedure means that:

- For those events that are activated by a resource that is not a counter or a sequencer, no events are lost, because all those events are updated.

However, if counter and sequencer states are propagated back as trace unit resources, so that a loop is created as shown in Figure 4-18 on page 4-172, then:

- If a counter at zero event is being used as a resource to activate either the sequencer or another counter, then that counter at zero resource might be propagating through the resource selection logic at the time when the procedure ends. In this case, the sequencer state event or other counter at zero event that is activated by that counter at zero resource might be lost.
- If a sequencer state event is being used as a resource to activate a counter, then that sequencer state resource might be propagating through the resource selection logic at the time when the procedure ends. In this case, the counter at zero event that is activated by that sequencer state resource might be lost.

The procedure also 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:

- If the state of the sequencer is propagated back as a trace unit resource, then the view of the sequencer as an event and the view of the sequencer as a resource each show the same sequencer state.
- If the state of a counter is propagated back as a trace unit resource, then the view of the counter as an event and the view of the counter as a resource each show the same counter state. The counter state might be either:
  - The counter is at zero.
  - The counter is not at zero.

When the procedure is complete, TRCSTATR.PMSTABLE can be set to 1. TRCSTATR.PMSTABLE must not be set to 1 before this procedure is complete. When TRCSTATR.PMSTABLE is set to 1, all trace unit resources and events must remain in a quiescent state.

4.4.6 Example of resource behavior when disabling the trace unit

The following is an example of a programming of a trace unit and the expected behavior when disabling the trace unit.

Program the trace unit as follows:

- Program single address comparator 0 to match on an instruction at address `0x1000`.
- Program resource selector 2 to select single address comparator 0.
- Program Counter 0 to decrement on resource selector 2.
- Set RLDSELF to `0b1` in counter 1.
- Program resource selector 3 to select counter 0.
- Program event tracing event 0 to fire on resource selector 3.

If an instruction is executed at address `0x1000` and appears in the instruction trace stream Arm expects that this instruction has an effect on the resources, as recommended in *Memory access resources on page 4-147*.

The following assumptions apply to this example:

- The instruction at address `0x1000` can be traced.

- The instruction causes single address comparator 0 to match.

In this example, counter 0 is at zero, so that the next decrement event causes the counter to reload and the counter-at-zero to fire.

The following sequence occurs:

1. An instruction executes at address `0x1000`.
2. The trace unit is disabled.

In this scenario, Arm expects the following sequence:

1. Single address comparator 0 matches.
2. Resource selector 2 fires for one cycle.
3. The decrement event for counter 0 is active, counter 0 reloads, and the counter-at-zero resource fires for one cycle. The state of the counter is propagated through the resource selection logic one more time through resource selector 3.
4. Resource selector 3 fires, causing event 0 to fire.
5. The trace unit generates an Event element for event 0.
6. The trace unit resources and events are now disabled and no other event elements are generated.
7. TRCSTATR.PMSTABLE can now be set to `0b1`.

4.5 Program examples

The following sections provide examples of:

- *Enabling the trace unit for program flow trace*.
- *Enabling the trace unit for instruction and data trace*.
- *Setting a trigger on an instruction address on page 4-182*.

4.5.1 Enabling the trace unit for program flow trace

To enable the trace unit for basic program flow trace:

1. Program the access control registers so that you can access and control the trace unit. This might include unlocking the OS Lock and unlocking the Software Lock, if using the memory-mapped interface.
2. Set TRCPRGCTLR.EN=0, to disable the trace unit.
3. Program the registers and values that Table 4-15 shows. The registers are in alphabetical order but you can program them in any order.

Table 4-15 Enabling the trace unit for basic program flow trace

Register	Value	Description
TRCCONFIGR	0x000018C1	Enable the return stack, global timestamping, Context ID, and Virtual context identifier tracing.
TRCEVENTCTL0R	0x00000000	Disable all event tracing.
TRCEVENTCTL1R	0x00000000	
TRCSTALLCTLR	0x00000000	Disable stalling, if implemented.
TRCSYNCPR	0x0000000C	Enable trace 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 synchronization.
TRCVICTLR	0x00000201	Enable ViewInst to trace everything, with the start/stop logic started.
TRCVIIECTLR	0x00000000	No address range filtering for ViewInst.
TRCVISSCTLR	0x00000000	No start or stop points for ViewInst.

4. Set TRCPRGCTLR.EN=1, to enable the trace unit.

4.5.2 Enabling the trace unit for instruction and data trace

To enable the trace unit for instruction and data tracing:

1. Program the access control registers so that you can access and control the trace unit. This might include unlocking the OS Lock and unlocking the Software Lock, if using the memory-mapped interface.
2. Set TRCPRGCTLR.EN=0, to disable the trace unit.

3. Program the registers and values that Table 4-16 shows. The registers are in alphabetical order but you can program them in any order.

Table 4-16 Enabling the trace unit for instruction and data trace

Register	Value	Description
TRCCONFIGR	0x00031FC7	Enable all the options except cycle counting and branch broadcast.
TRCEVENTCTL0R	0x00000000	Disable all event tracing.
TRCEVENTCTL1R	0x00000000	
TRCSTALLCTLR	0x00000000	Disable stalling, if implemented.
TRCSYNCPR	0x0000000C	Enable trace synchronization every 4096 bytes of trace.
TRCTRACEIDR	Nonzero	Set a value for the trace ID, with bit[0]=0.
TRCTSCTLR	0x00000000	Disable the timestamp event. The trace unit still generates timestamps due to other reasons such as trace synchronization.
TRCVDARCCTLR	0x00000000	No address filtering for ViewData.
TRCVDCTLR	0x00000001	Enable ViewData.
TRCVDSACCTLR	0x00000000	No address filtering for ViewData.
TRCVICTLR	0x00000201	Enable ViewInst to trace everything, with the start/stop logic started.
TRCVIIECTLR	0x00000000	No address range filtering for ViewInst.
TRCVISSCTLR	0x00000000	No start or stop points for ViewInst.

4. Set TRCPRGCTLR.EN=1, to enable the trace unit.

4.5.3 Setting a trigger on an instruction address

To enable the trace unit to insert an Event element and assert an output when the PE executes a particular instruction:

1. Program the trace unit as *Enabling the trace unit for instruction and data trace on page 4-181* describes.
2. Set TRCPRGCTLR.EN=0, to disable the trace unit.
3. Program, in any order, the registers, and values that Table 4-17 shows.

Table 4-17 Setting a trigger on an instruction address

Register	Value	Description
TRCACVRn, n=0	Address	Set the address of the instruction
TRCACATRn, n=0	0x00000000	Set the comparator for instruction address matching
TRCSSCCRn, n=0	0x00000001	Select single address comparator 0, for single-shot control
TRCSSCSRn, n=0	0x00000000	Clear the STATUS bit to zero
TRCRSCTLRn, n=2	0x00030001	Select single-shot comparator 0 for resource selector 2
TRCEVENTCTL0R	0x00000002	Select resource selector 2 to fire event 0
TRCEVENTCTL1R	0x00000001	Enable event 0 to generate an Event element in the trace

4. Set TRCPRGCTLR.EN=1, to enable the trace unit.

Chapter 5
Descriptions of Trace Elements

This chapter describes the trace elements in the ETMv4 architecture. It contains the following sections:

5.1 Elements summary tables

This section provides a summary of each of the trace elements. It contains two tables:

- Table 5-1 shows the instruction trace elements.
- Table 5-2 on page 5-187 shows the data trace elements.

5.1.1 Instruction trace elements summary table

Table 5-1 Instruction elements summary

Category	Name	Short name	Payload	Purpose
Synchronization	Trace Info	-	Setup, Speculation depth	Provides a point in the instruction trace stream where analysis of the trace stream can begin. Includes information about the tracing setup and the depth of speculation.
	Trace On	-	-	Indicates that there was a discontinuity in the trace stream.
	Discard	-	-	Indicates that all uncommitted P0 elements must be discarded because it is not possible to resolve the speculation.
	Overflow	-	-	Indicates when a trace buffer overflow occurs.

Table 5-1 Instruction elements summary (continued)

Category	Name	Short name	Payload	Purpose
Basic program flow	Atom	E, N	Status, Right-hand key	This P0 element indicates when a P0 element instruction is observed. An *Atom element* can be classified as either: **E** When the P0 element instruction is a taken branch or is a load or store instruction. The payload indicates that the instruction was executed. **N** When the P0 element instruction is a not-taken branch. The payload indicates that the instruction was not executed. Implies execution has continued from the target of the previous P0 element to the next P0 element instruction in the program flow, and if the P0 element instruction is a direct branch and it is an E Atom then this indicates that execution has continued to the target of that branch.
	Q	-	[Number M], Right-hand key	This P0 element indicates that one or more instructions have been executed. It includes an optional number M, that indicates the number of instructions executed. The executed instructions might include branch instructions.
	Exception	-	Exception type, Address, Right-hand key, Pending	Indicates that an exception has occurred. This is a P0 element. The address implies that execution has continued from the target of the previous P0 element up to, but not including, the supplied address. The pending payload indicates whether a serious fault was pending when the exception was taken.
	Exception Return	-	Right-hand key[a]	For Armv7-A, Armv7-R, and Armv8-A or Armv8-R, this is not a P0 element and it indicates that the most recent P0 element was an Exception Return. For Armv6-M, Armv7-M, and Armv8-M, this is a P0 element that indicates an exception return has occurred. P1 elements for the stack pop might be associated with an Exception Return element.
	Address	-	Instruction set, Address	Indicates the start address and instruction set from which the next P0 element implies execution. This is generated when an indirect branch is taken or when an exception occurs or after a Q element is generated.
	Context	-	Context ID, *Virtual context identifier*, Security state, Exception level, 64-bit state	Indicates the execution context of any future P0 elements. This is generated when any of the payload items change.
	Branch Future Flush	-	-	Indicates that the LO_BRANCH_INFO data has been invalidated and an implicit branch will not occur.

Table 5-1 Instruction elements summary (continued)

Category	Name	Short name	Payload	Purpose
	Function Return	-	-	Indicates that a function return instruction has caused a pop of data from the stack. A Function Return element is generated after the *Atom element* for the instruction. This element is a P0 element and the stack pop data transfers for the function return are associated with the Function Return element.
Timing information	Timestamp	-	Time value, Cycle count	Indicates the global timestamp value, to enable the trace streams to be correlated with each other and with other available trace sources.
	Cycle Count	-	Count value	Indicates the number of processor clock cycles between two Commit elements.
Event tracing	Event	-	Event number	Indicates when an arbitrary programmed event occurs.
Speculation resolution	Commit	-	Number M	Indicates how many P0 elements are committed for execution. The oldest M uncommitted elements are committed.
	Cancel	-	Number M	Indicates how many P0 elements are canceled. The most recent M uncommitted elements are canceled.
	Mispredict	-	-	Indicates that the most recent uncanceled *Atom element* has an incorrect E or N payload.
Conditional instruction trace	Conditional Instruction	C	Right-hand key	Indicates that a conditional non-branch instruction has been observed. The right-hand key associates the Conditional Instruction element to a Conditional Result element.
	Conditional Result	R	Left-hand key, Result	Indicates the result of one or more conditional non-branch instructions. The left-hand key associates the Conditional Result element with one or more Conditional Instruction elements.
	Conditional Flush	F	-	Indicates that any unresolved Conditional Instruction elements are discarded.
Synchronization with the data trace	Data Sync Mark	-	[Number M]	Associates P1 data trace elements with the P0 instruction trace elements. It can include an optional Number M to enable coarse correlation of the instruction and data trace streams.

a. This payload is only present on the Armv6-M, Armv7-M, and Armv8-M architectures.

5.1.2 Data trace elements summary table

Table 5-2 Data elements summary

Category	Name	Short name	Payload	Purpose
Synchronization	Trace Info	-	-	Provides a point in the data trace stream where analysis of the trace stream can begin.
	Discard	-	-	Indicates that the trace unit is unable to generate P2 elements for any remaining P1 elements that might require them.
	Overflow	-	-	Indicates when a trace buffer overflows. The buffer that has overflowed might be the instruction trace buffer, or the data trace buffer.
	Suppression	-	-	Indicates that some trace has been lost, and that the generation of some P1 and P2 elements is being suppressed.
	Resynchronization	-	-	Indicates that the trace analyzer may not have all the necessary information to analyze the trace.
Address and data value tracing	P1 Data Address	P1	Address, Transfer index, Endianness, Left-hand key, Right-hand key	Indicates the data address of a data transfer.
	P2 Data Value	P2	Data value, Left-hand key	Indicates the data value of a data transfer.
Timing information	Timestamp	-	Time value	Indicates the global timestamp value in the trace, to enable the trace streams to be correlated with each other and with other available trace sources.
Event tracing	Event	-	-	Indicates that an event has occurred.
Synchronization with the instruction trace	Data Sync Mark	-	[Number M]	Enables synchronization of the data trace stream with the instruction trace stream.

5.2 Descriptions of instruction trace elements

The following sections describe the:

5.2.1 Trace Info instruction trace element

A Trace Info instruction trace element provides a point in the instruction trace stream where analysis of the trace stream can begin.

Trace Info elements include setup information about:

- The static trace programming that does not change during a trace run, including:
  - Whether load instructions are traced explicitly.
  - Whether store instructions are traced explicitly.
  - Whether cycle counting is enabled, and if enabled, the cycle count threshold.
  - Whether tracing of conditional non-branch instructions is enabled.
- Dynamic information that might change during a trace run, such as:
  - The speculation depth. This indicates how many uncommitted P0 elements were traced before the Trace Info element.

The trace unit generates a Trace Info element whenever a trace synchronization request occurs.

A trace synchronization request automatically occurs:

- At the beginning of each new trace run, that is, the first time tracing starts after the trace unit has been enabled. In this case, the Trace Info element is generated when the trace unit is enabled and before any other trace elements are generated.
- After an overflow of either of the trace unit buffers.

In addition, the trace unit can be programmed to generate trace synchronization requests on a periodic basis, so that the trace streams can be analyzed if either stream has been stored in a circular trace buffer. The field that enables this functionality is TRCSYNCPR.PERIOD.

Note

There is no requirement to generate a new Trace Info element after every time that ViewInst becomes inactive. This is because, despite the discontinuity in the trace that is caused by the filtering, the programming of the trace remains the same. The Trace On element is provided to indicate gaps in the trace stream, including any gaps that are caused by filtering. See *Trace On instruction trace element on page 5-190*.

As mentioned, whenever a trace analyzer receives a Trace Info packet, it receives information about the setup of the trace. However, it 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. Therefore, whenever the trace unit generates a Trace Info element, it must also generate a Context element and an Address element soon after the Trace Info element.

The rules for generating the Address and Context elements are as follows:

- If a trace buffer overflow is the cause of a Trace Info element, or if the Trace Info element is the first Trace Info element that is generated in a trace run, then the subsequent Context and Address elements must be generated before the first Atom, Q, or *Exception element* is generated, to provide the trace analyzer with context and address information so that analysis of the Atom, Q, or *Exception element* can begin. This is shown in Figure 2-15 on page 2-67.
- If a trace synchronization request is the cause of a Trace Info element, then:
  - If ViewInst is active when the Trace Info element is generated, the subsequent Context and Address elements must provide the context and address information for the target of the most recent P0 element. See Figure 2-14 on page 2-67.

    Note

    If the trace unit generates the Context and Address elements immediately after the Trace Info element, then the most recent P0 element might have occurred before the Trace Info element. See Figure 2-13 on page 2-66.
  - If ViewInst is inactive when the Trace Info is generated, then when ViewInst becomes active, and after a Trace On element is generated, the Context and Address elements must be generated before the first Atom, Q, or *Exception element* is generated, to provide the trace analyzer with context and address information so that analysis for the Atom, Q, or *Exception element* can begin.

If a Cancel element occurs, then:

- If the Cancel element cancels any P0 elements before a Trace Info element, then a trace analyzer must discard all 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 or Address elements.

  In this scenario, information from the canceled Trace Info element can still be used:
  - The trace unit must generate new Context and Address elements to ensure that the trace analyzer can analyze the trace.
  - These new elements must indicate the target of the most recent P0 element that has not been canceled.

  The Address and Context might indicate the target of a P0 element from before the Trace Info, 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 and Address elements prior to the next new P0 element, then this might prevent the indication of execution of some instructions before the Trace Info element.

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

Using Trace Info elements to start trace analysis

After a trace analyzer has located an A-Sync packet and synchronized with the trace stream, it must search for the following elements to begin to analyze the trace stream:

1. A Trace Info element.
2. A Context element and an Address element.

As described, a trace unit might not generate Context and Address elements immediately after it generates a Trace Info element.

As mentioned, if a Cancel element cancels a Trace Info element then a trace analyzer can still use the information from the discarded Trace Info element, but if the Context and Address elements are also discarded, then it must wait for the trace unit to generate new Context and Address elements.

Encountering Trace Info elements after trace analysis has started

As mentioned, a trace unit might generate Trace Info elements periodically, as a result of trace 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 a Trace Info element, it can ignore subsequent Trace Info elements in the same trace that is run because the static trace programming cannot change and the speculation depth is updated by other element types during the trace run. The other element types that update the speculation depth are listed in Table 6-4 on page 6-239.

5.2.2 Trace On instruction trace element

A Trace On element indicates a discontinuity in the trace stream. The trace unit inserts this element after a gap in the instruction trace stream, for example:

- When the trace unit is enabled and before any P0 elements.
- If some instructions are filtered out of the trace.
- When instruction trace is lost because a trace buffer overflow occurs.

After a trace unit generates a Trace On element, then before the next Atom, Q element, or *Exception element* occurs, it must generate an Address element to indicate where tracing starts.

A Context element must be generated after a Trace On element, and before the next Atom, Exception or Q element, to indicate the context where tracing starts, unless the context has not changed since the previous Context element was output. If this is the first Trace On element, the Context element must be output before the first Atom, Exception, or Q element.

When a Cancel element occurs, a trace analyzer must discard any Trace On elements it encounters as it discards the number of P0 elements indicated by the Cancel element. For example, if a Cancel element indicates that the three most recent P0 elements are canceled, then the trace analyzer must discard all elements from the Cancel element back to, and including, the third most recent P0 element, and any Trace On elements that are encountered in that section of the element stream must also be discarded.

When a reset exception is traced, the preferred exception return address is UNKNOWN, see *Exception instruction trace element on page 5-196*. Any instructions since the most recent committed P0 element are not traced. If ViewInst was active for these instructions, this is not considered a gap in the trace 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 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. Trace analyzers must be aware that these additional Trace On elements might be traced. See *Basic program trace with filtering when an exception occurs, example one on page A-435* for an example where an extra Trace On element is traced.

5.2.3 Discard instruction trace element

A Discard instruction trace element is generated if uncommitted P0 elements remain when the trace unit enters a state where it is no longer able to generate trace. If the trace unit is no longer able to generate trace, the predicted outcomes of instructions that are traced by P0 elements, such as conditional branch instructions, cannot be resolved, and therefore a Discard instruction trace element indicates that all uncommitted P0 elements must be discarded.

A trace unit might no longer be able to generate trace when:

- The trace unit has been disabled. In this case:
  - The trace unit cannot trace the statuses of any uncommitted P0 elements that have already been output.
  - The Discard element is the last element output. All other trace elements must be output before the Discard element.
- The PE has been reset. In this case, the PE cannot complete any execution that might be in progress.
- A trace buffer overflow has occurred.

Note

- A trace unit might not generate a Discard element if no P0 elements are speculative.
- Because a Discard element means that all uncommitted P0 elements must be discarded, a Discard element also sets the speculation depth to zero.

The right-hand key of the next P0 element after a Discard element might be out of sequence from the right-hand keys that are associated with P0 elements prior to the Discard element.

When a trace analyzer encounters a Discard element:

- It must be aware that if the last committed P0 element is a conditional branch instruction, the E, or N status of that instruction might not be correct. This is because the trace unit is unable to generate any Mispredict elements that the conditional branch instruction might require.
- It must be aware that Conditional Result elements might not be generated for Conditional Instruction elements that have already been committed. That is, it must be aware that the trace unit might be unable to trace the results of the condition checks for conditional non-branch instructions that have already been committed.
- It must be aware that data transfers for committed load and store instructions might not be traced.

When a Cancel element occurs then a trace analyzer must not discard Discard elements. However, a Discard element sets the speculation depth to zero so it is not possible for a Cancel element to encounter a Discard element.

5.2.4 Overflow instruction trace element

The Overflow element indicates an overflow of the trace buffer. Therefore, some of the trace data might be lost.

After the trace unit recovers from the overflow, if tracing is:

Active The trace unit generates an Overflow element and then generates a Trace Info element. The Trace Info element indicates a speculation depth of zero.

Inactive The trace unit must immediately generate an Overflow element.

Disabled The trace unit must immediately generate an Overflow element before the trace unit is completely disabled.

If the speculation depth is nonzero then the trace unit generates a Discard element. This element indicates that the speculation status of all uncommitted P0 elements is not traced.

When a Cancel element occurs then a trace analyzer must not discard Overflow elements. However, an Overflow element sets the speculation depth to zero so it is not possible for a Cancel element to encounter an Overflow element.

5.2.5 Atom instruction trace element

An *Atom element* belongs to the P0 element group.

An *Atom element* implies that one or more instructions have been traced, up to and including the next P0 element instruction. A trace analyzer must analyze each instruction in the program image from the current address until it observes a P0 element instruction. This indicates that the PE has executed each instruction between the current address and the P0 element instruction.

A trace unit generates an *Atom element* when it observes a P0 element instruction. The *Atom element* can be one of the following types:

E Atom This *Atom element* is generated when the following instructions are observed:

- All the following instructions that are predicted as taken:
  - Branch instructions.
  - `ISB` instructions.
  - `WFI` and `WFE` instructions, provided TRCIDR2.WFXMODE is `0b1`.
- All load or store instructions, when tracing of those instructions is enabled.

N Atom This *Atom element* is generated when the following instructions are observed:

- All the following instructions that are predicted as not taken:
  - Branch instructions.
  - `ISB` instructions.
  - `WFI` and `WFE` instructions, provided TRCIDR2.WFXMODE is `0b1`.

For conditional branch instructions, an E *Atom element* implies the instruction passed its condition code check and an N *Atom element* implies the instruction failed its condition code check, although a trace unit might use a Mispredict element to modify the trace output.

Note

All other conditional instructions are not traced with an *Atom element* but instead they are traced with the Conditional Instruction element and the Conditional Result element contains the result of the condition code check.

Unconditional branch instructions might be traced using an E Atom or an N Atom. If an unconditional branch is traced using an N Atom, then the trace unit must correct this, either by generating a Mispredict element or by generating a Cancel element.

Regardless of the condition code check, an ISB instruction might be traced as an E Atom or an N Atom:

- All `ISB` instructions that do not pass the condition code and do not perform an Instruction Synchronization Barrier operation must be traced as an N Atom.
- All other `ISB` instructions must be traced as an E Atom.

Note

For an `ISB` instruction, a trace analyzer must not infer the value of the APSR condition flags from an *Atom element*.

The trace unit can be configured to classify `WFI` and `WFE` instructions as branch instructions. When `WFI` and `WFE` are classified as branch instructions, execution of these instructions generates a P0 *Atom element*. See also *Trace unit behavior on a PE low-power state on page 3-105* and *TRCIDR2, ID Register 2 on page 7-374*.

Each *Atom element* is generated in the program order in which they occur and the trace protocol encode and decode process must maintain this order.

Branch or ISB instructions that fail or are predicted to fail their condition code check can cause the trace unit either to generate an Undefined Instruction exception or to execute the instruction as a NOP, if the instruction is also undefined. If the instruction is executed as a NOP, then an N atom must be generated for these instructions. If the instruction generates an Undefined Instruction exception, then no *Atom element* is generated for the instruction because an exception is generated instead. The preferred exception return for the generated exception is the undefined instruction, which indicates that the instruction did not execute.

Each *Atom element* has a right-hand key that associates data transfers with the Atom. The right-hand key for each P0 element is incrementally one more than the previous P0 element, except when a Cancel element occurs and then the right-hand key for the next P0 element decrements by the number of canceled P0 elements.

All *Atom elements* are generated speculatively. A trace analyzer can infer execution from an *Atom element* but only after the *Atom element* has been committed by a Commit element.

For direct branch and ISB instructions, a trace analyzer must infer the target address and instruction set of *Atom elements* from the instruction opcode in the program image. If the direct branch or ISB is from a branch broadcast region, the trace analyzer does not need to infer the target address and instruction set because this is explicitly traced using an Address element.

When a Cancel element occurs then a trace analyzer must discard *Atom elements*.

Branch future instructions

If TRCIDR0.BF indicates that branch future support is implemented, when a branch future instruction is executed and sets LO_BRANCH_INFO.VALID to 0b1, and the branch future instruction is traced, the trace unit generates an E *Atom element*.

If TRCIDR0.BF indicates that branch future support is implemented, when a branch future instruction is executed and LO_BRANCH_INFO.VALID is either unchanged or not implemented, and the branch future instruction is traced, the trace unit generates an N *Atom element*.

Note

LO_BRANCH_INFO.VALID might have set to 1 by a previous branch future instruction.

The *Atom element* associated with a branch future instruction might be mispredicted in the same manner as for other branches.

These *Atom elements* need to be committed or cancelled in the same way as all other *Atom elements* and *Exception elements*.

All branch future instructions are classified as direct branch instructions, and therefore there is no requirement to trace the address of the subsequent instruction, but this might occur as part of periodic synchronization, or when in a Branch Broadcast region.

Implicit branches

When an implicit branch occurs due to an LO_BRANCH_INFO match due to a previous branch future instruction, and the implicit branch is taken, and the implicit branch is traced, the trace unit generates an E *Atom element*.

When an implicit branch occurs due to an LO_BRANCH_INFO match due to a previous branch future instruction, and the implicit branch is not taken due to a BFCSEL instruction which failed the bcond condition code check, and the implicit branch is traced, the trace unit generates an N *Atom element*.

When an E *Atom element* is generated due to an implicit branch caused by a BFX or BFLX instruction, the implicit branch is considered an indirect branch.

When an E *Atom element* is generated due to an implicit branch caused by a branch future instruction other than a BFX or BFLX instruction, the implicit branch is considered a direct branch.

Loop instructions

When a WLS or WLSTP instruction is executed and is traced and the instruction takes the branch, an E *Atom element* is generated.

When a WLS or WLSTP instruction is executed and is traced and the instruction does not take the branch, an N *Atom element* is generated.

When an LE or LETP branch is taken, and the branch is traced, an E *Atom element* is generated.

When an LE or LETP branch is implicitly taken due to an LO_BRANCH_INFO match, and the branch is traced, an E *Atom element* is generated.

When an LE or LETP branch is not taken and exits the loop, and the branch is traced, an N *Atom element* is generated.

Analyzing Atom elements with branch future support implemented

To support analysis of trace of execution from PEs which implement branch future instructions, the trace analyzer is required to store the following state.

```
type D_BRANCH_INFO is (
    boolean valid;
    boolean link;
    boolean t16ind;
    address_reg_t jump_address;
    address_reg_t end_address;
)
```

D_BRANCH_INFO is updated when branch future instructions occur, and when an implicit branch occurs the state stored in D_BRANCH_INFO is used to determine the target of the branch.

If TRCIDR0.BF indicates that branch future support is implemented, then branch future instructions generate an *Atom element* when traced. see Appendix F *Instruction Categories* for more information.

When an implicit branch occurs due to a branch future instruction, and is traced, an *Atom element* is generated to indicate the implicit branch occurred.

When a branch future instruction is traced with an E *Atom element*, this indicates to the trace analyzer that a future instruction at LO_BRANCH_INFO.END_ADDR is replaced with an implicit branch operation. The trace analyzer keeps track of the LO_BRANCH_INFO state and checks this on each instruction to ensure it uses the correct branch information for the implicit branch.

When an E *Atom element* is traced for a branch future instruction, the trace analyzer stores information about the branch future instruction in D_BRANCH_INFO:

```
D_BRANCH_INFO.valid = TRUE;
D_BRANCH_INFO.jump_address = LO_BRANCH_INFO.JUMP_ADDR;
D_BRANCH_INFO.end_address = LO_BRANCH_INFO.END_ADDR;
D_BRANCH_INFO.link = LO_BRANCH_INFO.LF;
D_BRANCH_INFO.t16ind = LO_BRANCH_INFO.T16IND;
```

When an N *Atom element* is traced for a branch future instruction, D_BRANCH_INFO is unchanged.

When analyzing an *Atom element* and inferring instruction execution, for each instruction the current state in D_BRANCH_INFO is checked to determine if an implicit branch has occurred.

```
if (D_BRANCH_INFO.valid) then
    // Check if the next instruction matches end_address
    if (D_BRANCH_INFO.end_address == current_address) then
        // Branch to the target for an E Atom
        if (E_Atom) then
            next_address = D_BRANCH_INFO.jump_address;
            // Check for return stack push
            if (TRCPRGCTLR.RS == 1 && D_BRANCH_INFO.link) then
                increment = D_BRANCH_INFO.t16ind ? 0x2 : 0x4;
                return_stack.push(current_address + increment);
        // Move to the next sequential instruction for an N Atom
        if (N_Atom) then
            increment = D_BRANCH_INFO.t16ind ? 0x2 : 0x4;
            next_address = current_address + increment;
        D_BRANCH_INFO.valid = FALSE;
```

When D_BRANCH_INFO.valid is TRUE, the state stored in D_BRANCH_INFO supersedes the information in the program image at D_BRANCH_INFO.end_address.

Branch future instructions might be inside an IT block. The type of the *Atom element* generated for these instructions does not indicate if the branch future instruction passed or failed the IT condition code check, but only indicates if the LO_BRANCH_INFO was updated.

If branch future support is implemented in the trace unit, branch future instructions are always considered branch instructions. This is regardless of whether the branch future functionality is implemented or enabled in the PE.

If branch future support is not implemented in the trace unit, branch future instructions are never considered branch instructions. This is regardless of whether the branch future functionality is implemented or enabled in the PE. Furthermore, because the branch future instructions are not considered branch instructions, conditional branch future instructions are considered conditional instructions and therefore generate Conditional Instruction elements when Conditional instruction tracing is enabled.

Atom elements associated with branch future instructions or implicit branches are committed or cancelled in the same way as other *Atom elements*.

The type of *Atom element* associated with a branch future instruction might be mispredicted in the same manner as for other branches.

5.2.6 Q element

A Q element belongs to the P0 element group in the instruction trace stream, and must be explicitly committed or canceled. A trace unit can generate a Q element to imply that at least one instruction has been executed, possibly including branch or `ISB` instructions.

Q elements are optional, and must be explicitly enabled if the trace unit is to use them. Q elements must only be enabled when data tracing and conditional non-branch tracing are either both not implemented or both not enabled.

When Q elements are enabled, the instruction trace stream might not contain enough information to determine the complete program flow, because some changes in flow might not be explicitly indicated. Arm recommends that Q elements are only used in cases where generating the full ETMv4 instruction trace stream might cause the performance of the PE being traced to degrade significantly.

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.

A Q element is always followed by an Address element that indicates where execution is to continue after all the instructions that are implied by the Q element have been executed. If the last instruction implied by the Q element is a branch instruction then the Address element indicates the target of that branch. Otherwise, it indicates the instruction address immediately following the last instruction that is implied by the Q element. If the trace unit is disabled immediately after a Q element, there is no requirement to output the Address element after the Q element.

Each Q element is generated in the program order in which they occur, and the trace protocol encode and decode process must maintain this order.

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 branch instruction, it is UNKNOWN whether the branch was taken. The status of these branches is not explicitly given in the trace stream but it might be possible to infer the status of a given branch that is based on other trace that is generated. After the trace analyzer has analyzed M instructions, the following Address element indicates where PE execution continues.

A Q element must not imply Exceptions. Exceptions must be traced using *Exception elements*. If a Q element implies an Exception Return instruction which is taken then that instruction must be the last instruction that is implied by the Q element. The trace unit must then generate an Exception Return element.

If a Q element implies an executed `ISB` instruction, then this must be the last instruction implied by the Q element if execution continues from a new context after the `ISB`.

After a Q element, if execution continues from a new context, a Context element is required after the Q element. The Context element might be generated before or after the 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.

If the return stack is enabled, then if any instructions that are implied by a Q element are indirect branches, they do not cause a return stack match. If any instructions that are implied by a Q element are Branch with Link instructions, then they do not cause any entries to be pushed on the return stack.

If TRCQCTLR is implemented, the trace unit supports the ability to control when Q elements are permitted in the trace using address range comparators.

When leaving a region where Q elements are permitted, either by a branch or by sequential execution out of the region, if a Q element is being used to imply the execution of the last instruction in the region, this is the last instruction that is implied by the Q element. The Q element is output and the subsequent Address element indicates the address of the first instruction that is executed out of the region.

When entering a region where Q elements are permitted, either by a branch or an exception, the branch or exception are traced using non-Q elements. Any subsequent instructions in the permitted region might be traced using Q elements.

When entering a region where Q elements are permitted by sequential execution into the region, 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. The Q element must not indicate execution of any P0 element instructions outside the permitted region.

5.2.7 Exception instruction trace element

An *Exception element* belongs to the P0 element group.

A trace unit generates an *Exception element* when a PE takes an exception, if tracing is enabled.

An *Exception element* contains the following information:

- An indication that an exception has occurred.
- The type of exception, such as PE reset, IRQ, or HardFault.
- An exception return address.

For an Exception, 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.

A trace unit generates *Exception elements* as they occur in the program order and the trace protocol encode and decode process must maintain this order.

Each *Exception element* has a right-hand key. The right-hand key for each P0 element is incrementally one more than the previous P0 element, except when a Cancel element occurs. If a Cancel element occurs, then the right-hand key for the next P0 element decrements by the number of canceled P0 elements.

For *Exception elements*, the right-hand key:

- Might point to data transfers, for Armv6-M, Armv7-M, and Armv8-M PEs. These data transfers are when the PE pushes its state on the stack.
- Does not point to any data transfer for Armv7-A, Armv7-R, Armv8-A, and Armv8-R PEs.

When a Cancel element occurs, a trace analyzer must discard *Exception elements*.

If an exception is taken under certain conditions, then the trace unit must generate an Address element before the *Exception element*, unless the Address element would be removed due to a return stack match. The conditions under which this scenario might be necessary are:

- The exception is taken at the target of a taken indirect branch.
- The exception is taken at the target of a taken direct branch or ISB from a branch broadcast region.
- The exception is taken at the target of another exception.

If the context changed before the exception, then a Context element must also be generated before the *Exception element*. This provides information about the address and context from where the exception was taken.

If an exception is the result of an attempt to execute an instruction at an invalid address, the trace unit generates an *Exception element* to trace the full invalid address, within the following constraints:

- All address bits up to the virtual address width implemented by the PE must be traced.
- All other address bits must represent an invalid address for the current translation stage.

Arm recommends the full invalid address is traced.

For example, on an Armv8-A PE, 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.

Prior to ETMv4.5, only a single instruction could be partially executed, known as an interrupt continuable instruction. From ETMv4.5, when the PE implements MVE, multiple instructions might be partially executed.

When an exception is taken, the trace indicates all instructions which have been fully or partially executed.

Architectural exceptions

Table 5-3 shows the exception types for the Armv7-A, Armv7-R, Armv8-A, and Armv8-R architectures. For Armv8-A and Armv8-R, the table also shows exceptions for AArch32 and AArch64 state.

Table 5-3 Exception types for the Armv7-A, Armv7-R, Armv8-A, and Armv8-R architectures

Exception type	Armv7-A, Armv7-R	Armv8 taken to AArch32 state	Armv8 taken to AArch64 state
Reset	PE reset	PE reset	PE reset
IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt
Trap	UNDEFINED instruction Instruction or event trapped by a control bit	UNDEFINED instruction Instruction or event trapped by a control bit Illegal execution state	UNDEFINED instruction Instruction or event trapped by a control bit Illegal execution state
Call	SVC, HVC, SMC	SVC, HVC, SMC	SVC, HVC, SMC
Inst fault	Prefetch Abort, including from BKPT Instruction, Breakpoint, and Vector Catch debug events[a]	Instruction Abort, Software Breakpoint Instruction, Breakpoint, and Vector Catch exceptions[b]	Instruction Abort, Branch Target
Data fault	Synchronous Data Abort, including from Watchpoint debug events[a]	Synchronous Data Abort, Watchpoint[b]	Synchronous Data Abort
Inst debug	-	-	Software Breakpoint Instruction, Breakpoint, Vector Catch, Software Step[a]
Data debug	-	-	Watchpoint[b]

Table 5-3 Exception types for the Armv7-A, Armv7-R, Armv8-A, and Armv8-R architectures (continued)

Exception type	Armv7-A, Armv7-R	Armv8 taken to AArch32 state	Armv8 taken to AArch64 state
Alignment	-	-	Misaligned PC, Stack Pointer Misalignment
System error	Asynchronous Data Abort	SError interrupt	SError interrupt
Debug Halt	Software and Hardware debug events that cause entry to Debug state[a]	Halting debug event[b]	Halting debug event[b]

a. In Armv7, `BKPT` Instruction, Breakpoint, Vector Catch, and Watchpoint are *Software debug events* that might cause the PE to enter Debug state or take a debug exception. Software debug events which cause the PE to take a debug exception are traced as one of the Inst fault or Data fault exceptions.

b. In Armv8-A and Armv8-R, *Debug events* is used to describe entries to Debug state, and *Debug exceptions* is used to describe when debug generates exceptions.

Table 5-4 shows the preferred exception return addresses for each exception type.

Exceptions that are routed to another Exception level are traced using their normal exception type. Whether a particular exception is routed is controlled by combinations of the following PE register fields:

- HCR.TGE.
- HCR_EL2.TGE.
- HDCR.TDE.
- MDCR_EL2.TDE.
- SCR.EA, SCR.IRQ, and SCR.FIQ.
- SCR_EL3.EA, SRC_EL3.IRQ, and SCR.EL3.FIQ.
- HCR.AMO, HCR.IMO, and HCR.FMO.
- HCR_EL2.AMO, HCR_EL2.IMO, and HCR_EL2.FMO.

All instructions or exceptions that are trapped using any other control bits are traced using the Trap exception type.

Table 5-4 Preferred return address for exceptions

Exception type	Preferred exception return address
Reset	UNKNOWN[a]
IRQ interrupt	Instruction after the last executed instruction.
FIQ interrupt	Instruction after the last executed instruction.
Trap	For a trapped instruction or UNDEFINED instruction, the address of the instruction. For a trapped exception, 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[b]	Instruction after the last executed instruction.
Debug halt	Instruction after the last executed instruction.[c]

a. The preferred exception return address and context for a PE Reset exception are UNKNOWN. No instruction execution is indicated between the previous P0 element and the exception.

   When a PE Reset exception 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 PE Reset exception, the initial address and context after the previous Trace On element might be UNKNOWN.

b. 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.

c. Refer to the relevant Arm architecture manual for more details on the preferred return addresses for entry to Debug state. For Armv8-A and Armv8-R, the preferred return address is the value that is loaded into the DLR on entry to Debug state.

Table 5-5 shows the exception types and the preferred exception return address for the Armv6-M, Armv7-M, and Armv8-M architectures.

Table 5-5 Exception types for the Armv6-M, Armv7-M and Armv8-M architectures

Exception type	Preferred exception return address[a]
Reset	UNKNOWN[b]
NMI	Instruction after the last executed instruction[c].
HardFault	Instruction after the last executed instruction.
MemManage	Instruction that caused the fault.
BusFault	For synchronous faults, the instruction that caused the fault. For asynchronous faults, the instruction after the last executed instruction.
UsageFault	Instruction that caused the fault.
SecureFault[d]	See *Tracing SecureFault exceptions on page 5-205*.
Debug Monitor	Instruction after the last executed instruction.
SVCall	Instruction after the SVC instruction.
IRQ0-IRQ495	Instruction after the last executed instruction.[c]
PendSV	Instruction after the last executed instruction.
SysTick	Instruction after the last executed instruction.
Debug halt	Instruction after the last executed instruction.
Lazy FP push[e]	Instruction after the last executed instruction.
Lockup	For Armv7-M, the Lockup address. See *Armv7-M Architecture Reference Manual*. For Armv8-M, see *Further information for tracing exceptions on Armv8-M on page 5-205*.

a. For exceptions that are tail-chained, the preferred exception return address is always `0xFFFFFFFE`.

b. The preferred exception return address and context for a PE Reset exception are UNKNOWN. No instruction execution is indicated between the previous P0 element and the exception.

   When a PE Reset exception 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 PE Reset exception, the initial address and context after the previous Trace On element might be UNKNOWN.

c. When an interrupt continuable instruction is interrupted and the ICI bits are not set to zero, indicating that the instruction can be continued, the preferred exception return address must be (address of the interrupted instruction) + n, where n:

   Is +2 for 16-bit instructions.

   Is +2 or +4 for 32-bit instructions.

d. Similar to other exceptions, the SecureFault exception is only traced if the instruction or exception immediately before the SecureFault exception is traced. If non-invasive debug is disabled in Secure state and the SecureFault must be traced, then the handler address of the SecureFault exception must not be traced.

e. Lazy FP push is only required when data tracing is enabled. If non-invasive debug is disabled in Secure state and Lazy state preservation occurs in Secure state, the Lazy FP push is not traced, even if the Lazy FP push is pushing Non-secure data.

When an imprecise System Error exception occurs and is taken to AArch64, the preferred exception return address is the address stored in the relevant ELR when the exception is taken. When an imprecise System Error exception occurs and is taken to AArch32, the preferred exception return address is the address stored in the LR when the exception is taken, minus 4, which is consistent with other interrupts taken to AArch32.

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.

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.

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.

Non-architectural exceptions

Some exceptions are not architectural, such as:

- *Error Correction Code* (ECC) error correction.
- Generic replay of program execution.
- *Wait For Interrupt* (WFI).

An implementation might also define other exceptions that are not architectural exceptions.

These exceptions are not always architectural exceptions and therefore program code might not return from these exceptions and an Exception Return element might not be traced.

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

The use of any exceptions other than the architectural exceptions is optional and IMPLEMENTATION DEFINED. These exceptions are not required to be traced but these exceptions are intended to be used to simplify tracing of certain microarchitectural situations.

The WFI exception might indicate how far execution has progressed before the PE enters the Wait for Interrupt state, although this is not mandatory. An alternative method to indicate the progress of instruction execution before and after a WFI instruction is to trace the next P0 element instruction or exception after the PE restarts execution after the WFI.

Additional information for tracing exceptions on Armv6-M, Armv7-M, and Armv8-M

If late arrival preemption occurs, the original exception is not traced and only the late arriving exception traced.

When an exception is tail-chained, the preferred exception return address is traced as `0xFFFFFFFE`. This indicates that no execution has occurred. There is no requirement to trace the exception return target address when tail-chaining occurs. The Exception level traced with the preferred exception return address is UNKNOWN when a tail-chained exception is traced.

If the PE enters lockup state:

- If tracing was active, a Lockup *exception element* is generated. For Armv7-M, the preferred exception return address for the exception is the lockup address, see *Armv7-M Architecture Reference Manual*, but for Armv8-M, see *Further information for tracing exceptions on Armv8-M on page 5-205*.
- No further P0 elements are generated while the PE is in lockup state.
- If tracing was active on entering lockup state, any exception that causes the PE to leave lockup state is traced as normal and the preferred exception return address is the lockup address, see *Armv7-M Architecture Reference Manual*.

If a derived exception is taken because of an error that occurred while taking an exception, this is treated like a late arriving exception and the derived exception is traced instead of the original exception. If multiple derived exceptions occur, only the final derived exception is traced. The preferred exception return address for the derived exception is the normal preferred exception return address for the original exception.

If a lockup occurs due to an exception entry, the exception, which is either the original exception or a derived exception, is traced. The lockup exception is then traced with a preferred exception return address of `0xFFFFFFFE`. The Exception level that is traced with the preferred exception return address is UNKNOWN.

If a derived exception occurs due to an error while returning from an exception, this is traced in a similar way to tracing tail-chaining, and the preferred exception return for the derived exception is `0xFFFFFFFE`.

If entry to lockup state occurs due to an error while returning from an exception, this is traced in a similar way to tracing tail-chaining and a lockup exception is traced with the preferred exception return of `0xFFFFFFFE`.

If a normal exception occurs and is taken as a late arriving exception instead of a derived exception, the normal exception is traced, with a preferred exception return address of the original exception. In all cases of late arriving and derived exceptions, only the final exception that is taken is traced.

When any exception is traced, if a serious fault is also pending, this is indicated in the exception type. This can determine the approximate location where a fault occurred if higher priority exceptions are taken while a fault is still pending. The exception type includes an indication if any of the following exceptions are pending:

- BusFault.
- HardFault.
- MemManage fault.
- SecureFault.

If an instruction caused the entry to lockup state, and if program execution continues without taking an exception, this instruction might not be traced on exit from lockup. Additionally, any address comparators might not match on the lockup instruction.

If a derived exception occurs while returning from an exception, for example a BusFault on the exception return stack operations, the entry to the derived exception is treated in the same way as a tail-chained exception. In this scenario, the exception return is considered to have executed, and it is not canceled.

When data tracing is enabled, the data transfer operations that are performed as part of the exception entry are traced and associated with the *Exception element*.

Note

For a PE reset exception, data transfers might be traced and associated with the processor reset exception. These transfers can safely be ignored.

Tracing partially executed instructions on an exception

When an exception is taken, an instruction is considered partially executed if any of the following are true:

- The instruction is interrupt continuable, and RETPSR.ICI indicates a non-zero value for that instruction.
- The instruction is a beat-wise vector instruction, and RETPSR.ECI indicates at least one beat has been executed for that instruction and not all beats have been executed.
- The instruction is between 2 partially executed beat-wise vector instructions.
- The instruction is immediately after a single partially executed beat-wise vector instruction.

When execution resumes on an instruction which was previously partially executed, but a subsequent exception occurs without advancing RETPSR.ICI or RETPSR.ECI, or completing the instruction, this subsequent attempt to execute the instruction is not considered a partially executed instruction.

Note

Other than interrupt continuable and beat-wise vector instructions, the Armv8- M architecture permits only implicit branches to be partially executed. Implicit branches are only permitted to be partially executed when between 2 partially executed beat-wise vector instructions, or immediately after a single beat-wise vector instruction.

When an *Exception element* is generated and RETPSR.ECI is not zero, the preferred exception return address in the *Exception element* indicates which instructions have been partially or fully executed. The preferred exception return address is defined as the address of the youngest partially or fully executed instruction, plus `0x2`.

When an exception occurs, the youngest partially or fully executed instruction is defined as:

Table 5-6 Tracing partially executed instructions

RETPSR.ECI	Youngest partially or fully executed instruction	Preferred exception return address
`0b00000000`	Instruction before the instruction at return address	Return address
`0b00000001`	Instruction A	`A+0x2`
`0b00000010`	Instruction A	`A+0x2`
`0b00000011`	Reserved	
`0b00000100`	Instruction A	`A+0x2`
`0b00000101`	Instruction B, where B is a beat-wise vector instruction	`B+0x2`
`0b00000101`	Instruction B, where B is an implicit branch	Target of instruction B
`0b0000011x`	Reserved	
`0b00001xxx`	Reserved	

See the definition of the RETPSR.ECI field in the Arm ®v8-M Architecture Reference Manual for details of instruction A and instruction B.

When an exception occurs and RETPSR.ICI is non-zero, the preferred exception return address is defined in Table 5-7.

Table 5-7 Tracing partially executed instructions

RETPSR.ICI	Preferred exception return address	Notes
`0b00000000`	Return address	No interrupt continuable instructions.
Otherwise	Return address + `0x2` or Return address + `0x4`	Instruction at return address is a 32-bit instruction
	Return address + `0x2`	Instruction at return address is a 16-bit or 32-bit instruction

It is possible that one or more instructions are between two beat-wise vector instructions, and an exception is taken such that both of the vector instructions are partially executed. In this scenario, RETPSR.ECI has the value `0b00000101` indicating two vector instructions are partially executed, and the return address of the exception indicates the first vector instruction.

Furthermore, one or more of these intervening instructions might be a branch, including implicit branches. The trace for this scenario generates an *Atom element* to indicate the branch executed, and this *Atom element* is committed and indicates that the first vector instruction and the branch have been executed. Subsequently, an *Exception element* is generated which indicates that the second vector instruction has been executed, with a preferred exception return address of the second vector instruction plus `0x2`.

When execution returns from the exception, these instructions will typically be executed again to completion starting from the first vector instruction, and this means the branch will be traced a second time, with another *Atom element* generated.

When an exception is taken, if an instruction was traced and the instruction is neither fully executed or partially executed, then the trace unit must indicate the instruction has been canceled, using *Cancel elements* where appropriate.

When an exception is taken, if an instruction was traced and the instruction is fully executed, then the trace unit must indicate the instruction has been committed, using *Commit elements* where appropriate.

When an exception is taken, if an instruction is partially executed:

- Prior to ETMv4.5, the instruction might be observed by the trace unit or might not be observed by the trace unit.
- From ETMv4.5, the instruction must be observed by the trace unit.

When a partially executed instruction is not observed, the trace unit considers the instruction to have not architecturally executed, which means that:

- The *Exception element* does not indicate the instruction was executed, and therefore has a preferred exception return address of the instruction.
- If the instruction is a P0 instruction, an *Atom element* is not generated.
- If the instruction is conditional, a Conditional Instruction element is not generated.
- The Single-shot Comparator Controls do not consider the instruction to have executed.
- The ViewInst start/stop logic does not consider the instruction to be a start point or a stop point that architecturally executes.

When a partially executed instruction is observed:

- The *Exception element* indicates the instruction was executed. See Table 5-6 on page 5-202 and Table 5-7 for more details on the preferred exception return address.
- If the instruction is a P0 instruction, an *Atom element* is generated and this is committed.

- If the instruction is conditional and conditional instruction tracing is implemented and enabled for this instruction, a Conditional Instruction element is generated.
- The Single-shot Comparator Controls consider the instruction to have executed, for more information see *Programming a single-shot control to self-reset after it fires on page 4-161*.
- The ViewInst start/stop logic has special behaviors, for more informtion, see *Behavior of the start/stop control during a trace run on page 4-121*.

When an exception is taken and no instructions have been partially executed, the *Exception element* will have a preferred exception return address corresponding to the instruction after the last completed instruction.

When an exception is taken and one or more instructions have been partially executed, and the most recent partially executed instruction is not a branch instruction, the *Exception element* will have a preferred exception return address of the most recent partially executed instruction, plus `0x2`. This indicates the partially executed instruction has been executed, although does not indicate whether it has been partially executed or fully executed.

When an exception is taken and one or more instructions have been partially executed, and the most recent partially executed instruction is a branch instruction, the *Exception element* will have a preferred exception return address of either the target of the branch instruction, or the most recently partially executed instruction plus `0x2`. This means the branch will have been traced as executed, but nothing at the target has been executed, although the trace does not indicate whether the branch has been partially executed or fully executed.

These rules permit an implicit branch to be partially executed, either as the instruction immediately after a beat-wise vector instruction that is also partially executed, or between two beat-wise vector instructions that are also partially executed. Therefore an implicit branch might be traced with an *Atom element* and indicated as committed using a *Commit element*, yet when execution returns from the exception the branch might be traced again with an *Atom element*. A trace analyzer should be aware that such branches might be traced more than once, even if only completely executed a single time.

When data trace is implemented and enabled:

- Partially executed instructions must be observed by the trace unit, since the instruction must be traced to allow the data elements to be traced and associated with the instruction.

  Note

  A partially executed instruction might be traced multiple times, each time it is partially executed or resumed.

- The data transfer operations that are performed as part of the exception entry are traced and associated with the *Exception element*.

  Note

  For a PE reset exception, data transfers might be traced and associated with the processor reset exception. These transfers can safely be ignored.

See Appendix A *Examples of Trace* for some examples of exceptions during vector instructions.

Exception element analysis

The tracing of partially executed instructions generally mean that a partially executed instruction appears in the trace more than once. The first occurrence is before the exception, and a second occurrence is when the PE returns from the exception to continue execution.

When two beat-wise vector instructions are overlapping with a Loop End instruction or implicit branch between these instructions, and an exception is taken with both of the vector instructions partially executed, the Loop End instruction or implicit branch will be indicated as executed before the *Exception element*. When execution returns from the exception, the Loop End instruction or implicit branch will typically be executed again, and thus a second *Atom element* will be generated. A trace analyzer needs to be aware of this scenario, because it might appear that more iterations of the loop occurred than expected, due to the multiple *Atom elements* for the Loop End instruction or implicit branch.

Further information for tracing exceptions on Armv8-M

Tracing Lockup exceptions

The definition of the Lockup address is different between Armv7-M and Armv8-M, and therefore the definition of the preferred exception return address is different for Armv8-M PEs.

For Armv8-M, the preferred exception return address for a Lockup exception is as follows:

- When a Lockup occurs for any of the following reasons, when an instruction caused the Lockup, the preferred exception return address is the address of the instruction:
  - BusFault or MemManage fault on an instruction fetch.
  - BusFault on a precise data access.
  - MemManage fault on a data access.
  - Breakpoint triggered.
  - UsageFault other than INVPC.
  - SecureFault, where one of the following bits was set to 1:
    SFSR.INVTRAN
    SFSR.INVEP
    SFSR.AUVIOL
- When a Lockup occurs due to execution of an SVC instruction, the preferred exception return address is the address of the SVC instruction plus 2.
- For all other causes of Lockup, including any caused by taking an exception or returning from an exception, the preferred exception return address is the Lockup address as defined in the Armv8-M architecture. This address is `0xEFFFFFFE`.

Tracing SecureFault exceptions

Where an instruction directly causes one of the following flags to be set to 1 and a SecureFault exception is taken, the preferred exception return address is the address of the instruction:

- SFSR.INVTRAN.
- SFSR.INVEP.
- SFSR.AUVIOL.

When an exception return causes one of the following flags to be set to 1 and a SecureFault exception is taken, the preferred exception return address is `0xEFFFFFFE`:

- SFSR.INVER.
- SFSR.INVIS.

This SecureFault exception is traced in a similar way to tracing tail-chaining.

In the case of a branch from Non-secure to Secure state without the required SG instruction at the branch target, the preferred exception return address is of the branch target, indicating the SecureFault occurred at the branch target.

When taking an exception causes the following flags to be set to 1, the exception is traced as a derived exception, and the preferred exception return address is the address of the original exception:

- SFSR.AUVIOL.

When lazy state preservation causes a SecureFault, the preferred exception return address is the preferred exception return address of the original exception. This is traced as a derived exception.

Tracing Beat-wise vector exceptions

Armv8.1-M allows the PE to execute some vector instructions in a beat-wise manner, and furthermore permits overlapping of the execution of multiple vector instructions.

Beat-wise execution is only visible in the trace when an exception occurs. When an exception occurs during beat-wise vector execution, some beats of an instruction might be executed before the exception is taken, and further beats might be executed when returning from the exception. For the PE to determine which beats to execute on returning from an exception, information is stored in the RETPSR.ECI field.

Overlapping execution of multiple beat-wise vector instructions is only visible in the trace when an exception occurs. A trace analyzer needs to be aware that when an exception occurs there might be multiple instructions that have not completed all of their execution, and those instructions might be resumed when returning from the exception.

Figure 5-1 shows an example sequence of instructions, with both non-overlapping execution and overlapping execution.

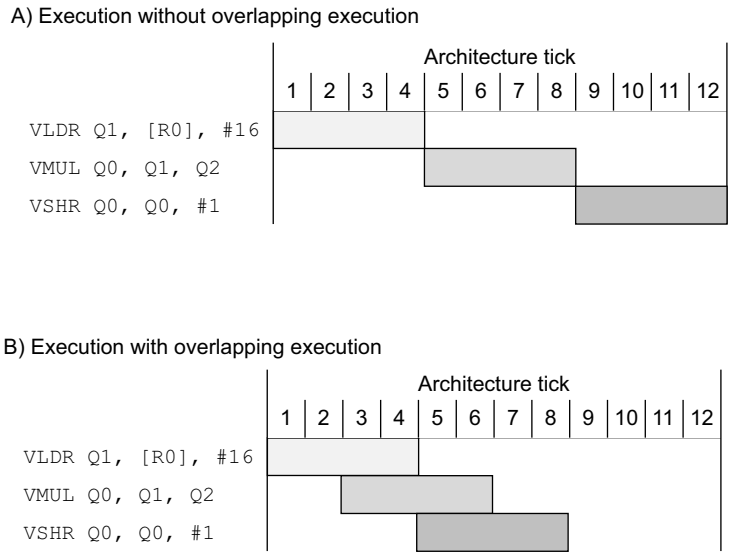


Figure 5-1 Overlapping and Non-overlapping execution

The following figures show example sequences of instructions, with an exception taken at different architecture ticks during the execution.

In Figure 5-2 on page 5-207, a single beat of one instruction has been executed.

- When the exception is taken, the VLDR instruction is indicated as executed.
- The preferred exception return address is of the VLDR instruction plus `0x2`.
- If a ViewInst start point occurred on the VLDR instruction, then the Start/Stop state will be in the started state.
- If a ViewInst stop point occurred on the VLDR instruction, then the Start/Stop state does not enter the stopped state.

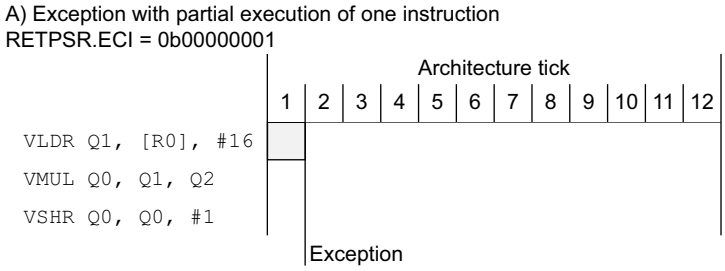


Figure 5-2 Exceptions during vector instructions example A

In Figure 5-3, two beats of one instruction have been executed.

- When the exception is taken, the VLDR instruction is indicated as executed.
- The preferred exception return address is of the VLDR instruction plus `0x2`.
- If a ViewInst start point occurred on the VLDR instruction, then the Start/Stop state will be in the started state.
- If a ViewInst stop point occurred on the VLDR instruction, then the Start/Stop state does not enter the stopped state.

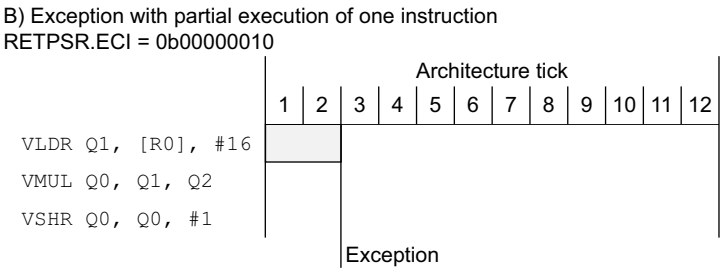


Figure 5-3 Exceptions during vector instructions example B

In Figure 5-4, three beats of one instruction have been executed.

- When the exception is taken, the VLDR instruction is indicated as executed.
- The preferred exception return address is of the VLDR instruction plus `0x2`.
- If a ViewInst start point occurred on the VLDR instruction, then the Start/Stop state will be in the started state.
- If a ViewInst stop point occurred on the VLDR instruction, then the Start/Stop state does not enter the stopped state.

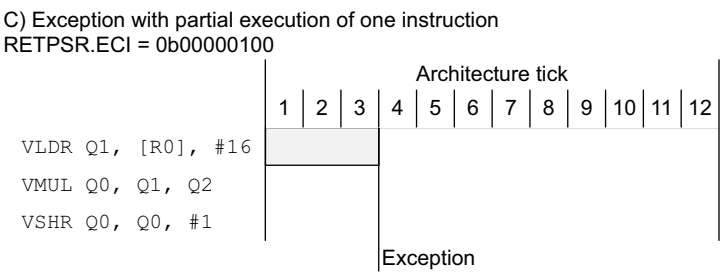


Figure 5-4 Exceptions during vector instructions example C

In Figure 5-5, three beats of one instruction have been executed, and one beat of a second instruction has been executed.

- When the exception is taken, the VLDR instruction and the VMUL instruction are indicated as executed.
- The preferred exception return address is of the VMUL instruction plus 0x2.
- If a ViewInst start point occurred on either the VLDR or VMUL instructions, then the Start/Stop state will be in the started state.
- If a ViewInst stop point occurred on either the VLDR or VMUL instructions, then the Start/Stop state does not enter the stopped state because neither instruction has completed.

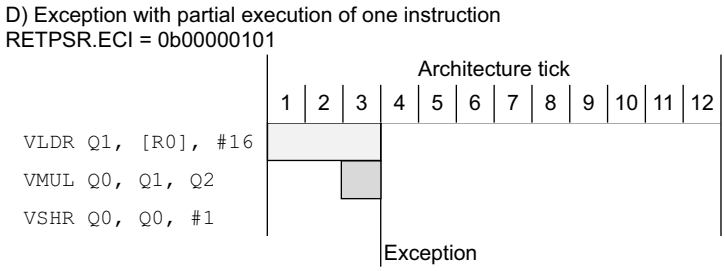


Figure 5-5 Exceptions during vector instructions example D

For more information see:

- *Overview of the ViewInst function on page 4-119*.
- *Exception instruction trace element on page 5-196*.
- *Tracing branch future instructions on page A-442*.

5.2.8 Exception Return instruction trace element

The Exception Return element indicates when an exception return occurs and the most recent Atom or Q element identifies the instruction that caused a return from an exception. The behavior depends on the architecture, for:

Armv7-A, Armv7-R, and Armv8-A and Armv8-R

For these architectures, the Exception Return element is not a P0 element and it does not include a right-hand key. A Commit or Cancel element does not explicitly commit or cancel an Exception Return. However, if a trace unit performs a Commit on the Atom or Q element associated with the Exception Return then it commits both the Atom or Q element and the Exception Return. Similarly, if a trace unit performs a Cancel on the Atom associated with the Exception Return then it cancels both the Atom or Q element and the Exception Return.

An Exception Return element might be generated for a Return from Exception instruction that fails its condition code check.

Armv6-M, Armv7-M, and Armv8-M

For these architectures, the Exception Return element is a P0 element and it includes a right-hand key. The key associates the data transfers for the exception return event with the position in the instruction trace. As the Exception Return element is a P0 element, a Commit or Cancel element does explicitly commit or cancel an Exception Return.

An Exception Return instruction in Armv6-M, Armv7-M, and Armv8-M cannot be identified purely from the instruction opcode. Instead an instruction can be identified as an Exception Return instruction if it loads the EXC_RETURN value into the PC in Handler mode and is one of the following instructions:

- A POP/LDM instruction that includes loading the PC.
- An LDR instruction that has the PC as a destination.

- A BX instruction that is used with any register.

Furthermore, the ETMv4 architecture is relaxed to permit an implementation to treat these instructions that load the EXC_RETURN value into the PC in Thread mode as an Exception Return instruction, or to not treat them as an Exception Return instruction. Arm recommends these instructions are not treated as an Exception Return instruction.

When a Cancel element occurs then a trace analyzer must discard Exception Return elements.

5.2.9 Address instruction trace element

An Address element indicates both of the following for the next instruction to be executed:

- The instruction set.
- The instruction address.

If a trace analyzer cannot infer the address or instruction set from the trace then the trace unit must generate an Address element. Occasions when the trace analyzer might not be able to infer the address or instruction set from previous trace include:

- When an indirect branch is taken.
- When a direct branch or ISB is taken in branch broadcast mode, see TRCBBCTLR for more information.
- When an exception is taken.
- When mis-speculation occurs and the address cannot be inferred.
- After a Q element is generated.

The Address element that results must be generated before the next P0 element, unless either of the following is true:

- The Address element can be omitted because of a return stack match. See *Use of the return stack on page 5-222*.
- Tracing is inactive at the target of the branch or exception.

In addition, a trace unit might generate an Address element after:

- A Trace Info element, to indicate the address where analysis of program execution can start.
- A Trace On element, to indicate the address where tracing became active.

Typically, an Address element is required after an *Exception element* to indicate the target of the exception, since a trace analyzer is not usually able to infer the exception target address.

In some scenarios, an exception might be generated in the trace where the exception target is the next sequential instruction from the last instruction before the exception. This behavior depends on many factors and might only occur for IMPLEMENTATION DEFINED exceptions. If an exception is taken to the next sequential instruction, the trace unit is not required to output an Address element indicating the target address of the exception because this can be inferred from the previous execution.

A trace analyzer must have access to both an Address element and a Context element before it can determine the instruction set in use. This is because the Address element provides the instruction set and the Context element provides information on whether the PE is in AArch32 or AArch64 state. For more information, see *Decoding the instruction set from an Address packet on page 6-287*.

If a change of instruction set occurs that requires a change to the 64-bit state then the trace unit must generate a new Context element.

If there is a requirement to output an Address element, this must be output before the next Atom, Q, or *Exception element*.

For Armv8 PEs, if tagged addresses are in use, the virtual address in the instruction trace stream does not include the tag and, depending on the current Exception level, bits[63:56] are either:

- The sign-extension of bit[55].
- All zeros.

During an attempt to execute an instruction at an invalid address, if the trace unit generates an Address element, the Address element must indicate an invalid address, within the following constraints:

- All address bits up to the virtual address width implemented by the PE must be traced.
- All other address bits must represent an invalid address for the current translation stage.

Arm recommends the full invalid address is traced.

For example, on an Armv8-A PE, 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. An Armv8-A PE includes Translation Control Registers, TCR_EL<x>, which contain a TBI field for controlling whether to ignore the top byte of an address. If the current TBI field is changed from 0 to 1, 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 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.

Where the branch target is an invalid address, the invalid address must be traced as the preferred exception return address, according to the following constraints:

- All address bits up to the virtual address width implemented by the PE must be traced.
- All other address bits must represent an invalid address for the current translation stage.

Arm recommends the full invalid address is traced.

If a pointer authentication check fails, an exception might be taken from an invalid address, and an invalid address must be traced according to the following constraints:

- All address bits up to the virtual address width implemented by the PE must be traced.
- All other address bits must represent an invalid address for the current translation stage.

Arm recommends the full invalid address is traced.

Note

Bits[1:0] of addresses that are used in A64 instructions are always traced as zero.

Additional 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 instruction trace stream. These scenarios include, but are not limited to:

- When an instruction address is incorrectly speculated, and a subsequent Address element corrects the value of the previous incorrect Address element.
- When an instruction address can be inferred by the trace analyzer, for example for the target of a direct branch or ISB instruction, but an Address element is output anyway with the same address.

Arm recommends that the generation of additional unnecessary Address elements is minimized to ensure trace bandwidth is minimized.

When a Cancel element occurs then a trace analyzer must discard Address elements.

In a similar way to all other indirect branches, when tracing continues at the target of an indirect implicit branch then an *Address element* is generated to indicate the target of the branch. This *Address element* might not be output if a return stack match removes the need for the *Address element*.

When an implicit branch occurs inside a branch broadcast region and is traced with an E *Atom element*, then an Address element must be generated when tracing continues at the target of the implicit branch. An implicit branch occurs inside a branch broadcast region if D_BRANCH_INFO.end_address is inside a branch broadcast region.

5.2.10 Context instruction trace element

The Context element indicates the execution context in which the instructions execute. The Context element provides the following information:

- The Context ID. See *Context ID tracing on page 2-81*.
- The Virtual context identifier. See *Virtual context identifier tracing on page 2-81*.
- 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.

A trace unit must generate a Context element:

- When a Context synchronization event occurs which changes the context:
  - When an `ISB` instruction performs an Instruction Synchronization Barrier operation.
  - On exception entry.
  - On exception return.
- Whenever tracing is enabled, after the initial Trace Info element.
- Periodically, after a Trace Info element is generated.
- When tracing becomes active, if the context has changed.
- If mis-speculation means an incorrect Context element was output.

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.

In Armv7 PEs that implement the Security Extensions, or in Armv8 PEs that implement EL3 using AArch32, the CONTEXTIDR_EL1 is banked between the Security states. On these PEs the Context ID value is the value of the CONTEXTIDR_EL1 for the current Security state. On all other PEs the Context ID value is the value of the CONTEXTIDR_EL1.

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 Context ID Register.
- Writes to the VTTBR.
- Writes to the CONTEXTIDR_EL2.
- Changes from Secure to Non-secure state without using an exception return.
- Changes in Exception level other than via an exception or exception return.

In these scenarios, the trace unit might generate the Context element containing the new context value at any time between the P0 element prior to the system instruction and 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.

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. This is because no instructions or exceptions are indicated to have been executed from the new

context. A Context element indicating the new context must be generated after the exception because the exception is a Context synchronization event. If the exception changes the context, then the Context element must indicate the new context. This might happen if, for example, the Security state changes.

On a PE reset, and prior to the PE registers being updated, the Context ID and Virtual context identifier are traced as zero.

A trace unit is not required to generate a Context element if tracing becomes inactive before any instructions are executed in the new context.

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 instruction trace stream. Such a scenario might include when the context 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.

Table 5-8 shows the Exception levels that the Context element can use, depending on the Arm architecture, the Security state, and the type of exception.

Table 5-8 Exception levels for a Context element

Exception level	Armv8-A, Armv8-R[a]	Armv7-A[b]	Armv7-R[b]	Armv6-M, Armv7-M, Armv8-M[bc]
Security state == NS				
EL0	EL0	User	-	-
EL1	EL1	System, Supervisor, Abort, FIQ, IRQ, Undef	-	-
EL2	EL2	Hyp	-	-
EL3	-	-	-	-
Security state == S				
EL0	EL0	User	User	Thread
EL1	EL1[d]	-	-	-
EL2	EL2[e]	-	-	-
EL3	EL3	Mon, System, Supervisor, Abort, FIQ, IRQ, Undef	System, Supervisor, Abort, FIQ, IRQ, Undef	Handler

a. See the *Armv8 Architecture Reference Manual* for information about the exception model in 32-bit modes.
b. Armv7 PEs that do not implement the Security Extensions are considered to operate in Secure state.
c. For Armv8-M PEs, the trace unit does not distinguish between the Secure and Non-secure states, and treats both states as Secure state.
d. The Armv8-A architecture does not permit EL1 if the EL3 mode is configured to be 32-bit state. In 32-bit state, all Secure privileged modes must use EL3.
e. If ETMv4.4 architecture is implemented, the indication of EL2 while in Secure state is permitted.

When a Cancel element occurs, a trace analyzer must discard Context elements.

When in a region where the Virtual context identifier is not to be traced as defined by TRFCR_EL2.CX and Virtual context identifier tracing has been enabled, the trace unit must generate Context elements with a Virtual context identifier value of zero.

5.2.11 Branch Future Flush element

In some circumstances, LO_BRANCH_INFO in the PE might be invalidated and a subsequent implicit branch will not occur. The effect of tracing a branch future instruction is that the trace analyzer will expect an implicit branch at the BF branch point, however if LO_BRANCH_INFO has been invalidated then the implicit branch will not occur. A *Branch Future Flush element* indicates that the LO_BRANCH_INFO data has been invalidated and an implicit branch will not occur.

If D_BRANCH_INFO in the trace analyzer contains valid state for an implicit branch, this state is invalidated when a *Branch Future Flush element* occurs, and the original program image instruction at D_BRANCH_INFO.end_address is used.

For more information see *Analyzing Atom elements with branch future support implemented on page 5-194*.

Branch Future Flush element generation

When any of the following occur, and a branch future instruction had been traced and set LO_BRANCH_INFO.VALID to TRUE, a *Branch Future Flush element* is generated:

- (LO_BRANCH_INFO.VALID && LO_BRANCH_INFO.BF) transitions from TRUE to FALSE, including:
  - A taken branch.
  - A PE Reset.
  - Armv8.1-M exceptions.
  - Exit from Debug State.
  - Exception return.
  - Function return.
- A *Trace On element* is generated.
- Other implementation specific events.
- A *Branch Future Flush element* is not generated when any of the following occur:
  - IMPLEMENTATION DEFINED exceptions which do not clear LO_BRANCH_INFO.VALID.
  - A lazy FP push.
  - Lockup.

Multiple Branch Future Flush elements might be merged if no *Atom elements* or *Exception elements* are in between the *Branch Future Flush elements*.

Branch Future Flush elements appear in architectural order in respect to *Atom elements* and *Exception elements*.

If an IMPLEMENTATION DEFINED exception does clear the LO_BRANCH_INFO.VALID in the PE then a *Branch Future Flush element* must be explicitly traced.

Some *Branch Future Flush elements* are implied by analyzing other trace packets, and when this occurs a Branch Future Flush packet does not need to be explicitly traced.

Branch Future Flush element analysis

Branch Future Flush elements must be discarded by *Cancel elements*.

When a *Branch Future Flush element* is received, D_BRANCH_INFO.valid is set to FALSE.

5.2.12 Resynchronization element

The *Resynchronization element* indicates to the trace analyzer that it may not have all the necessary information to analyze the trace. This can occur when instructions, such as branch future instructions, have not been traced but their effects are observed. In such situations it is not always possible to predict which instructions are executed and where the PE branches to. The *Resynchronization element* instructs the trace analyzer to restart part of the trace synchronization process.

Resynchronization element generation

The *Resynchronization element* is output when an implicit branch occurs and is traced, and any of the following are true:

- The corresponding branch future instruction was not traced.
- The trace for the corresponding branch future instruction might have been lost due to an overflow.
- A *Trace Info element* has been generated between the branch future instruction and the implicit branch.

When a *Resynchronization element* is generated, it must be inserted before the *Atom element* due to the implicit branch. It is permissible to have other *Atom elements* between these two elements.

After a *Resynchronization element*, if conditional tracing is implemented and enabled, the trace unit inserts a *Conditional Flush element* after the *Atom element* for the implicit branch, and before the next of any of the following:

- *Conditional Instruction element.*
- *Conditional Result element*.
- *P0 element.*

After a *Resynchronization element*, if tracing continues after the implicit branch, the trace unit inserts an *Address element* after the *Atom element* for the implicit branch, and before the next P0 element. The *Address element* indicates the target of the implicit

There must not be any *Address elements* between a *Resynchronization element* and the *Atom element* due to the implicit branch.

Resynchronization element analysis

When the trace analyzer processes a *Resynchronization element* and D_BRANCH_INFO.valid is FALSE then it is to abandon analysis and search forward for the next *Address element*. Otherwise the trace analyzer must ignore this element.

Resynchronization elements must be discarded by *Cancel elements*.

5.2.13 Function Return instruction trace element

The Function Return instruction trace element indicates that a function return instruction has caused a pop of data from the stack, if data tracing is implemented and enabled. A Function Return element is generated after the *Atom element* for the instruction.

The Function Return element is a P0 element and the stack pop data transfers for the function return are associated with the Function Return element.

An instruction can be identified as a function return instruction if it loads the FNC_RETURN value into the PC and is one of the following:

- A `POP/LDM` instruction that includes loading the PC.
- An `LDR` instruction that has the PC as the destination.
- A `BX` instruction that is used with any register.

Note

Some function return instructions are also data transfer instructions. The data for these instructions is associated with the instruction, and the function return stack pop operations are associated with the Function Return element.

The Function Return packet is only generated for an Armv8-M PE when data tracing is enabled.

5.2.14 Timestamp instruction trace element

The Timestamp instruction trace element inserts a global timestamp into the instruction trace stream. A Timestamp instruction trace element is generated whenever a *timestamp request* occurs.

The timestamp value corresponds to whichever of the following that was most recently generated:

- *Atom element*.
- *Exception element*.
- Numbered Data Sync Mark element.
- Event element.
- Q element.
- Exception Return element, for Armv6-M, Armv7-M, and Armv8-M profile PEs.

A timestamp request occurs whenever:

- The timestamp event occurs, as set by the TRCTSCTLR.
- The trace unit:
  - Generates a Trace Info instruction trace element.
  - Recovers from a trace buffer overflow.
- The PE:
  - Takes an exception when not in a prohibited region.
  - Returns from an exception handler when not in a prohibited region.
- A flush of the trace unit is requested.
- An `ISB` instruction performs an Instruction Synchronization Barrier operation when not in a prohibited region.

Note

If data tracing is supported and enabled, then whenever a timestamp request occurs, a timestamp is requested in both trace streams. This means that the trace unit generates both a Timestamp instruction trace element and a Timestamp data trace element. See *Timestamp data trace element on page 5-230*.

Certain timestamp request events, such as the execution of `ISB` instructions, exceptions taken, and exception returns, result in a timestamp request regardless of whether the trace unit traces the request.

There is no requirement for a Timestamp element to be generated in the instruction trace stream on each occasion that ViewInst becomes active.

When a trace unit receives a timestamp request then if necessary, for example to avoid a trace buffer overflow, it can delay the generation of a Timestamp instruction trace packet. This 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 stream somewhere near the time of the request.

After the first time ViewInst is enabled, or after an overflow of the instruction trace buffer, the next Timestamp element must not be generated until after the trace unit has generated either:

- An *Atom element*.
- An *Exception element*.
- A Numbered Data Sync Mark element.
- An Event element.
- A Q element.
- An Exception Return element, for Armv6-M, Armv7-M, and Armv8-M profile PEs.

This is so that the timestamp value can correspond to the most recent of these elements.

If a trace unit receives multiple timestamp requests close together then it might not generate a Timestamp instruction trace element for each request. For example, a trace unit can ignore the second request of two successive timestamp requests if both of the following are true:

- The second request is not caused as a result of a trace synchronization request.
- None of the following element types have been generated between the two requests:
  - *Atom element*.

- — *Exception element*.
- — Numbered Data Sync Mark element.
- — Event element.
- — Q element.
- — Exception Return element, for Armv6-M, Armv7-M, and Armv8-M PEs.

A timestamp value of zero indicates that the timestamp value is UNKNOWN. This might occur if the system does not support timestamping or if the timestamp is temporarily unavailable.

When cycle counting is enabled, each Timestamp instruction trace 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. Unlike the Cycle Count element, the:

- Cycle count does not affect the cumulative cycle count.
- Cycle count value can be zero, indicating that no cycles passed between the Cycle Count element and the element with which the Timestamp is associated.

When the trace unit is first enabled, the cycle count might not be known. If cycle counting is enabled and a Timestamp element is generated before any Cycle Count elements, then the Timestamp element must report the cycle count as UNKNOWN, because there are no previous Cycle Count elements. See *Global timestamping on page 6-259*.

When a Cancel element occurs then a trace analyzer can discard Timestamp elements. If a trace analyzer discards an Atom, Q or *Exception element* with which a Timestamp element is associated then the timestamp value might be associated with an incorrect Atom, Q or *Exception element*.

Timestamp Values and Armv8.4-Trace

From Armv8.4-A and above, if the PE implements Armv8.4-Trace these extensions control the timestamp values that are included in the trace stream.

If the trace unit is programmed to trace through context switches then the trace stream might contain more than one context, indicated by a change of Context ID or Virtual Context identifier. The timestamp values in the trace stream might exhibit some of the following behaviors:

- Timestamp values around context switches might be unreliable if the timestamp source is changed:
  - — The ordering of the changes in context information and the timestamp value source are not defined. The timestamp source might change before the Context element is generated.
  - — In the same way, the timestamp source can change after the Context element has been generated for the new context.
- Timestamp values are not guaranteed to monotonically increase within a trace session:
  - — If the trace unit is programmed to trace through a context switch then there is no guarantee that the timestamp value of the new source will be greater than the last timestamp element generated.
- Timestamp values monotonically increase for each context.

Trace analyzers must consider changes in context if they rely on timestamp values to localize the sections of trace stream and when aligning with trace streams from other sources.

5.2.15 Cycle Count instruction trace element

Cycle Count instruction trace elements are associated with Commit elements, so that when a Commit element is generated, a Cycle Count element might also be generated. A Cycle Count element indicates the number of processor clock cycles between the two most recent Commit elements that both had a cycle count value associated with them. Some Commit elements do not have a cycle count value associated with them.

To reduce trace bandwidth, the ETMv4 architecture only requires a Cycle Count element to be generated if the cycle count value exceeds a minimum threshold value at the time when a Commit element is generated. When a Commit element is generated:

- If the cycle count is less than the minimum threshold value, the trace unit does not generate a Cycle Count element.
- If the cycle count is equal to or greater than the minimum threshold value, the trace unit is requested to insert a Cycle Count element.

When a request to insert a Cycle Count element occurs, one of the following occurs:

- The trace unit generates a Cycle Count element immediately or before any future Commit element. 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. Arm recommends this behavior.
- The trace unit delays generation of the Cycle Count element until after one or more further Commit elements have been generated. When the Cycle Count element is generated it contains the value of the cycle counter at the time the most recent Commit element was generated, and the cycle counter is reset. Arm recommends this behavior is either not used or only occurs in rare and non-repetitive circumstances which are implementation specific.

The minimum threshold value is set by programming TRCCCCTLR.THRESHOLD.

The cycle counter has an IMPLEMENTATION DEFINED length between 12 and 20 bits. The cycle counter therefore supports values from 1 to 2^{20}–1. A value of 0 indicates that the cycle count value is UNKNOWN. This might occur when:

- A trace unit generates the first cycle count.
- The cycle counter overflows, indicating the value is greater than the maximum value that can be contained in the cycle counter.
- The processor clock stops, for example, if a *Wait For Interrupt* (WFI) occurs.
- An instruction trace buffer overflow occurs.

To produce a total cycle count, a trace analyzer can cumulatively add the values from all Cycle Count elements.

A trace analyzer must not use the cycle count values in Timestamp elements to produce a total cycle count.

When a Cancel element occurs, a trace analyzer must not discard Cycle Count elements.

5.2.16 Event instruction trace element

The Event element indicates when a programmed event occurs and its payload contains a number to identify the event number.

If an Event element occurs between two P0 elements or at the same time as a second P0 element, a trace unit must insert the Event element no earlier than the first P0 element, and no later than an IMPLEMENTATION DEFINED number of P0 elements after the first P0 element. 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.

See *TRCEVENTCTL0R, Event Control 0 Register on page 7-368* and *TRCEVENTCTL1R, Event Control 1 Register on page 7-368* for information about the programming of arbitrary events.

When a Cancel element occurs, a trace analyzer must not discard Event elements.

5.2.17 Commit instruction trace element

A Commit instruction trace element indicates the number of oldest uncommitted P0 elements that have been committed for execution.

The Commit element commits all P0 element types. These are:

- *Atom elements*.

- Q elements.
- Exception.
- Exception Return elements, for Armv6-M, Armv7-M, and Armv8-M PEs.

A trace unit generates a Commit element when one or more traced P0 elements are committed for execution.

A P0 element for a branch instruction might be mispredicted after it has been committed.

To reduce trace bandwidth, an implementation can combine multiple Commit elements and generate the Commit elements out of order in relation to the other elements. However, at any instance in time, the number of speculative P0 elements must not exceed the maximum speculation depth of the implementation.

If cycle counting is supported and 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. For more information, see *Cycle Count instruction trace element on page 5-216* and *Cycle counting on page 2-81*.

When a Cancel element occurs, a trace analyzer must not discard Commit elements.

5.2.18 Cancel instruction trace element

The Cancel element indicates the number of youngest uncommitted and uncanceled P0 elements that are canceled from execution. A trace unit might cancel elements because:

- A branch is mis-speculated.
- An exception occurs.

The Cancel element cancels all P0 element types. These are:

- *Atom elements*.
- Q elements.
- Exception.
- Exception Return elements, for Armv6-M, Armv7-M, and Armv8-M PEs.

A trace unit generates a Cancel element when one or more P0 elements are canceled.

If a cancelation causes execution to return to a point in the program flow that is not adjacent to a P0 element instruction, then the trace unit must generate an *Exception element* before it generates any P0 elements. It requires an *Exception element*, to indicate which instructions were executed at that point in the program flow.

When a Cancel element occurs, a trace analyzer must discard any Trace On elements it encounters as it discards the number of P0 elements indicated by the Cancel element. 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.

When discarding P0 elements that have been canceled, a trace analyzer must also discard all other element types that occur in the element stream between the Cancel element and the oldest P0 element that the Cancel element cancels. However, the ETMv4 architecture does not permit a trace analyzer to discard certain types of element. Table 5-9 describes the correct behavior on encountering each element during a cancelation operation.

When a P0 element is canceled, the trace unit stops tracing any P1 elements that relate to the right-hand key of that P0 element.

Table 5-9 Discard behavior on cancelation

Element	Behavior on cancelation
Trace Info	Discard if elements earlier in the trace stream than this element are canceled by this cancelation operation
Trace On	Discard
Context	Discard

Table 5-9 Discard behavior on cancelation (continued)

Element	Behavior on cancelation
Address	Discard
Atom	Discard
Exception	Discard
Exception Return	Discard
Commit	Do not discard
Mispredict	Discard
Timestamp	Can be retained when canceling
Data Sync Mark	Do not discard
Cycle Count	Do not discard
Overflow	Do not discard
Conditional (C)	Do not discard
Conditional (R)	Do not discard
Conditional Flush (F)	Discard
Discard	Do not discard
Event	Do not discard
Branch Future Flush	Discard
Resynchronization	Discard

A trace unit must generate Cancel elements as they occur, so that they appear in the element stream in the correct position in relation to other elements. This ensures that a trace analyzer can cancel the appropriate elements.

When a Cancel element occurs, a trace analyzer must discard Cancel elements.

5.2.19 Mispredict instruction trace element

The Mispredict element indicates that the most recent *Atom element* has the incorrect E or N status. For example, if a branch is predicted as taken, it is traced with an E Atom. If the prediction becomes incorrect then a Mispredict element is traced to indicate to a trace analyzer that the E Atom changes to an N Atom.

A trace unit might generate multiple Mispredict elements for the same Atom. A trace analyzer must use each Mispredict element to determine the final status of the Atom. For example, if an E Atom has two Mispredict elements, the first Mispredict element indicates the Atom is an N Atom and the second Mispredict element indicates it is an E Atom.

When a Mispredict element corresponds to an Atom for a direct branch instruction, before the trace analyzer can calculate the target of the direct branch, it must apply any applicable Mispredict elements so that it can determine whether it is an E Atom or an N Atom.

If a trace unit mispredicts only the branch target address then it does not generate a Mispredict element. The trace unit uses an Address element to correct the mispredicted target address. When analyzing a Mispredict element, any Address elements between the mispredicted *Atom element* and the Mispredict element must be discarded.

When a Cancel element occurs, a trace analyzer must discard Mispredict elements.

5.2.20 Conditional Instruction (C) instruction trace element

The Conditional Instruction element, also referred to as a C element, indicates when a conditional non-branch instruction is executed. A trace unit might not generate a C element for every conditional non-branch instruction, see *Trace behavior on tracing conditional instructions on page 2-71* for more information.

If conditional non-branch instructions occur between two P0 elements then a trace unit can generate the C elements for those conditional non-branch instructions, at any of the following times:

- Before it generates the P0 elements.
- Between the P0 elements.
- After it generates the P0 elements.

A C element includes a right-hand key that associates a subsequent Conditional Result element to the C element. The right-hand key uses a different key namespace from the P0 elements and its use is only for associating the Conditional Result elements with the C elements.

When a Cancel element occurs, a trace analyzer must not discard C elements.

5.2.21 Conditional Result (R) instruction trace element

The Conditional Result element, also referred to as an R element, indicates that the result of a Conditional Instruction element is known.

An R element contains a left-hand key, that associates the R element with one or more previous Conditional Instruction elements. The left-hand key uses a different key namespace from the P0 elements and its use is only for associating the R elements with the C elements.

It is IMPLEMENTATION DEFINED whether the result payload of a Conditional Result element contains either:

- The pass or fail result of the conditional instruction.
- The values of the APSR condition flags, which a trace analyzer can then use to compute the pass or fail result of the conditional instruction.

TRCIDR0.CONDTYPE shows whether R elements contain pass or fail results, or the value of the APSR condition flags.

When a Cancel element occurs, a trace analyzer must not discard R elements.

See *Trace behavior on tracing conditional instructions on page 2-71* for more information about the tracing of conditional instructions.

5.2.22 Conditional Flush (F) instruction trace element

The Conditional Flush element, also referred to as an F element, indicates that zero or more Conditional Instruction elements are canceled, along with the Conditional Result elements that are associated with them.

When a Cancel element occurs, a trace analyzer must discard F elements.

See *Trace behavior on tracing conditional instructions on page 2-71* for more information about the tracing of conditional instructions.

5.2.23 Data Synchronization Marker (Data Sync Mark) instruction trace element

The Data Sync Mark element ensures that the data transfers in the data trace stream are associated with the correct instructions. A trace unit generates Data Sync Mark elements only when data tracing is enabled.

The ETMv4 architecture provides the following types of Data Sync Mark elements:

- Numbered Data Sync Mark elements, which are generated periodically to enable coarse correlation of the instruction and data trace streams.
- Unnumbered Data Sync Mark elements, which are output more frequently and enable accurate association of the instruction and data trace streams.

When a Cancel element occurs, a trace analyzer must not discard Data Sync Mark elements.

5.3 Return stack

The ETMv4 architecture includes an optional return stack, that if implemented, can be used to remove address elements from the instruction trace stream for the purpose of optimizing the trace bandwidth.

TRCIDR0.RETSTACK indicates if the return stack is implemented. The depth of the return stack is IMPLEMENTATION DEFINED, though it can be up to 15 entries.

5.3.1 Use of the return stack

The operation of the return stack is as follows:

- Whenever the PE executes a Branch with Link instruction that is taken from outside of a branch broadcast region, the return address of the instruction, including the instruction set state, is pushed onto the return stack. If the return stack is already full, the oldest entry is discarded. A Branch with Link instruction implied by a Q element does not push onto the return stack.

  Note

  This does not apply to Branch with Link instruction that is not taken.

- Whenever an indirect branch is taken from outside of a branch broadcast region, the trace unit compares both the target address of the branch, and the IS indicator for the instruction at that address, with the address and instruction set contained on the top entry of the return stack. This comparison has one of two outcomes:
  - If the address and instruction set is exactly the same as the address and instruction set contained on the top entry of the return stack, the top entry of the stack is removed and the trace unit does not generate an Address element.
  - If the address and instruction set do not match the address and instruction set contained on the top entry of the return stack, the trace unit generates an Address element.

  Note

  This does not apply to an indirect branch that is not taken.

If an instruction is both a Branch with Link instruction and an indirect branch instruction then the order of operations on the return stack is as follows:

1. Push the return address and instruction set state onto the return stack.
2. When the branch target is known, compare the branch target address and instruction set state with the top entry of the return stack and remove if a match occurs.

Note

Previous trace architectures from Arm use a different order of operations.

All comparisons with the top entry of the return stack compare both the target address and target instruction set state.

The return stack must never match on:

- An indirect branch that causes the current context to change, such as a return from an exception.
- An indirect branch that is implied by a Q element.

A trace analyzer contains and maintains a copy of the trace unit return stack, so that when the trace unit traces an indirect branch instruction without an Address element, the trace analyzer knows that the target address is stored in the top entry of the trace analyzer return stack, and can therefore pop the top entry.

The return stack in the trace unit is flushed whenever the trace unit generates a Trace Info element or a Trace On element and on entry to a branch broadcast region. In addition, a trace unit implementation might flush the return stack at any time.

When these flushes occur, there is no requirement for the trace analyzer to be aware that a flush has occurred. This is because even though the contents of the trace unit return stack are flushed, there are no adverse consequences if the contents of the trace analyzer return stack are retained. What happens after a trace unit return stack flush is described as follows:

- In the trace analyzer, those return stack entries that are retained are never used.
- Whenever the PE executes a Branch with Link instruction, the new address and instruction set are pushed onto the trace unit return stack (and therefore the trace analyzer return stack) as before. 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.
- Whenever an address match occurs, the top entry of the trace unit return stack is removed, and the top entry of the trace analyzer return stack is popped, as before.

For a taken indirect branch, it is possible that the target address is predicted incorrectly by the PE. If the incorrect target address is traced with an Address element, this must be corrected with a further Address element with the correct target address. If the return stack is enabled and the incorrect target address does not match the top entry of the return stack, the correct target address must never match the top entry of the return stack, because an explicit Address element must be output to correct the incorrect target address.

When trace stream synchronization occurs, the trace unit must flush the return stack when the Trace Info element is generated. After the Trace Info element, an Address element and a Context element are required but might not be generated immediately. If the Address element and the Context element are not generated before the next Atom or *Exception element*, then any Branch with Link instructions must not push on to the return stack until both the Address element and the Context element have been generated. 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.

Return stack push

When an implicit branch is taken, and when all of the following are true, a return stack push occurs:

- LO_BRANCH_INFO.VALID == 1.
- LO_BRANCH_INFO.LF == 1.
- The corresponding branch future instruction was traced.
- The implicit branch is traced.

The address pushed on to the return stack is:

increment[31 : 0] = if LO_BRANCH_INFO.T16IND then 0x00000002 else 0x00000004

address[31 : 0] = if LO_BRANCH_INFO.END_ADDR[31:1]:0 + *increment*

When an end address match occurs in the trace analyzer, a return stack push occurs to the trace analyzer's return stack when all of the following have occurred:

- The branch future instruction was traced.
- The *Atom element* indicating execution indicates that the implicit branch was taken.
- D_BRANCH_INFO.link == 1.

The address pushed on to the return stack is:

increment[31 : 0] = if D_BRANCH_INFO.t16ind then 0x00000002 else 0x00000004

address[31 : 0] = if D_BRANCH_INFO.end_address[31:1]:0 + *increment*

Return stack pop

When the target of an implicit indirect branch matches the top entry of the return stack then one of the following

occurs:

- An *Address element* is not generated and the top entry of the return stack is removed.
- An *Address element* is generated, indicating the target of the implicit branch, and the return stack state is maintained.

The behavior for popping from the trace analyzer return stack is not changed by implicit branches.

Operation of the trace analyzer return stack

Note

A trace analyzer is not required to be aware of the depth of the trace unit return stack. However, the trace analyzer must implement a return stack with a depth of 15 entries.

The purpose of the trace analyzer return stack is to provide target addresses for indirect branch instructions that are traced without a target address, that is, to provide an address when an indirect branch instruction is traced without an Address element.

The trace analyzer return stack follows the same rules for pushing on to the return stack as the trace unit return stack, so that whenever the trace unit return stack changes, the same change occurs in the trace analyzer return stack. In this way, apart from when a trace unit return stack flush occurs, the trace analyzer return stack is always maintained as a copy of the trace unit return stack.

If the trace unit indicates that the PE has taken an indirect branch, but it does not output an Address element before the next Atom, Q, or *Exception element* to indicate the target of that branch, then the top entry of the trace analyzer return stack is popped and the value that it contains is used as the target address.

When trace stream synchronization occurs, the trace unit must flush the return stack when the Trace Info element is generated. After the Trace Info element, an Address element and a Context element are required but might not be generated immediately. If the Address element and the Context element are not generated before the next Atom or *Exception element*, then any Branch with Link instructions must not push on to the return stack until both the Address element and the Context element have been generated. 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.

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.

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 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, the mispredicted address remains in both return stacks because there is no reason to remove it. There are no adverse consequences of leaving mispredicted addresses in the stacks.

A return stack push does not occur if a Branch with Link instruction is traced with an N Atom, even if the status of the N Atom later changes to be an E *Atom element* as a result of a subsequent Mispredict element.

If more than one Mispredict element is output, 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 branch 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.

The trace analyzer never needs to flush its copy of the return stack. If the trace unit flushes the 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.

The trace analyzer does not need to prevent the return stack from being modified while in a branch broadcast region. The fact that the trace unit flushes the return stack when entering the branch broadcast region ensures that the return stack in the trace unit and the return stack in the trace analyzer remain synchronized.

5.4 Descriptions of data trace elements

The following sections describe:

- *Trace Info data trace element*.
- *Discard data trace element*.
- *Overflow data trace element*.
- *Suppression data trace element on page 5-227*.
- *P1 Data Address (P1) data trace elements on page 5-227*.
- *P2 Data Value (P2) data trace elements on page 5-230*.
- *Timestamp data trace element on page 5-230*.
- *Event data trace element on page 5-231*.
- *Data Synchronization Marker (Data Sync Mark) data trace element on page 5-231*.

5.4.1 Trace Info data trace element

A Trace Info data trace element provides a point in the data trace stream where analysis of the trace stream can begin.

The trace unit generates a Trace Info data trace element whenever a trace synchronization request occurs.

A trace synchronization request automatically occurs:

- At the beginning of each new trace run, that is, the first time tracing starts after the trace unit has been enabled. In this case, the Trace Info element is generated when the trace unit is enabled but before any other trace elements are generated.
- After an overflow of either of the trace unit buffers.

In addition, the trace unit can be programmed to generate trace synchronization requests on a periodic basis, so that the trace streams can be analyzed if either stream has been stored in a circular trace buffer. The field that enables this functionality is TRCSYNCPR.PERIOD.

After the first Trace Info element in a trace run, any other Trace Info elements that the trace unit generates in that run can safely be ignored.

5.4.2 Discard data trace element

A Discard data trace element indicates that the trace unit is unable to generate P2 elements for any P1 elements that have already been traced but that have not yet had any P2 elements associated with them, if they require P2 elements.

This might be because:

- The trace unit has been disabled. In this case:
  - The trace unit cannot trace any data transfers that are in progress.
  - The Discard element is the last element output. All other trace elements must be output before the Discard element.
- A trace buffer overflow occurs. In this case, the trace unit is not able to any trace data transfers that are in progress.
- The PE has been reset. In this case the PE cannot complete any data transfers that might be in progress.

5.4.3 Overflow data trace element

An Overflow data trace element indicates an overflow of the data trace buffer. This means that some of the trace might be lost, and that tracing is inactive until the overflow condition clears.

On a trace buffer overflow:

- Uncommitted elements must be discarded, because the trace unit is unable to generate and output any elements that show whether the uncommitted elements have been committed for execution, or canceled because of mis-speculation.
- If any P1 elements were output before the Overflow data trace element, and if the instruction trace buffer has also overflowed, then those P1 elements might not have a parent P0 element.

After the trace unit recovers from the overflow, if ViewData is:

Active The trace unit generates an Overflow element.

Inactive The trace unit must immediately generate an Overflow element.

Disabled The trace unit must immediately generate an Overflow element before the trace unit is completely disabled.

5.4.4 Suppression data trace element

Some ETMv4 implementations permit the trace unit to discard some of the data trace elements it generates if there is a risk that the data trace buffer in the trace unit might overflow. For an implementation to have this capability, it is required that the Stall Control Register, TRCSTALLCTLR, is implemented. TRCSTALLCTLR contains a field, DATADISCARD, that you can use to choose what types of data trace elements you would prefer to discard. The options are:

- Discard no data, therefore suppress nothing.
- Discard data load transfers, therefore suppress the generation of all P1 and P2 elements that are associated with data loads.
- Discard data store transfers, therefore suppress the generation of all P1 and P2 elements that are associated with data stores.
- Discard both data load and data store transfers, therefore suppress the generation of all P1 and P2 elements for both data loads and data stores.

The process of discarding elements in this way, so that the data trace buffer does not overflow, is called suppression, and the trace unit generates a Suppression element when it discards the first P1 element of the chosen type.

When the trace unit is suppressing P1 and P2 elements, it might still generate P2 elements for older P1 elements, if those older P1 elements were traced before suppression became active.

When the trace unit stops discarding P1 and P2 elements, tracing of these elements resumes. If suppression restarts again later, then the trace unit generates another suppression element for the first discarded P1 element.

A Suppression element is an indicator that some trace has been lost.

A Suppression element is only required for the first discarded P1 Data Address element. However, whenever the data trace stream is synchronized, a new Suppression element must be output for the first discarded P1 Data Address element after the Trace Info element.

It is ID Register 3, TRCIDR3, that indicates whether TRCSTALLCTLR is implemented. See *TRCIDR3, ID Register 3 on page 7-376*, and *TRCSTALLCTLR, Stall Control Register on page 7-405*.

5.4.5 P1 Data Address (P1) data trace elements

P1 Data Address elements, also referred to as P1elements, are generated if either data address tracing is enabled, data value tracing is enabled, or both are enabled. As stated in *Separate instruction and data trace streams on page 2-33*:

- Data address (DA) tracing is enabled if TRCCONFIGR.DA is set to 1.
- Data value (DV) tracing is enabled if TRCCONFIGR.DV is set to 1.

Whenever they are traced, data addresses are always traced as P1 elements, and data values are always traced as P2 elements. Instructions are traced as P0 elements, though not every instruction type is traced as a P0 element. See *About instruction trace P0 elements on page 2-35* and *Relationships between P0, P1, and P2 elements on page 2-38*.

If only data address tracing is enabled, the trace unit generates P1 Data Address elements, also referred to as P1 elements, that contain the addresses of data transfers.

If only data value tracing is enabled, the trace unit generates both P1 elements and P2 Data Value elements, also referred to as P2 elements. However, the P1 elements do not provide the addresses of any data transfers. In this case, P1 elements only provide links between P2 elements and P0 elements. That is, if only data value tracing is enabled, a P1 element provides a link between the data value of a data transfer and the load or store instruction that the data transfer is associated with.

If both data address tracing and data value tracing are enabled, then both P1 and P2 elements are generated, and the P1 elements contain addresses of data transfers, and also provide links between the P2 and P0 elements.

Table 5-10 shows this.

Table 5-10 The generation and content of P1 elements

DA	DV	P1 element generated?	Address provided in the P1 element?	P1 element provides a link between a P2 and P0 element?
0	0	N	-	-
0	1	Y	N	Y
1	0	Y	Y	N
1	1	Y	Y	Y

A P1 element contains:

- The address of a data transfer that the PE has performed as a result of executing a load or store instruction, unless only data address tracing is enabled.
- The endianness of the data transfer.
- A transfer index.
- A left-hand key, so that the element can be associated with its parent P0 element.
- A right-hand key, so that the element can be associated with its child P2 element.

The trace unit generates a new P1 element for each data transfer. This means that if one instruction results in multiple data transfers, a new P1 element is generated for each of those transfers.

When one instruction results in one P0 element and multiple child P1 elements, the child P1 elements all have the same value of left-hand key that associates them with the P0 element. The same value of left-hand key cannot be used again until all the child P1 elements have been output. Similarly, although each P1 element can only be associated with one child P2 element, the value of the right-hand key of the P1 element cannot be used again until the child P2 element has been output.

P1 elements can be traced out of program order, because the key mechanism enables each P1 element to be associated with its parent P0 element, and P0 elements are always traced in program order. A P2 element is always traced after its parent P1 element.

The meaning of a transfer index that is contained in a P1 element depends on the instruction. For instruction types that result in multiple data transfers, the transfer index indicates which part of the instruction the P1 element is associated with. For example, the following instruction results in one P0 element because it is a load instruction:

- `LDM r0, {r2,r5,r6}`

Three child P1 elements are generated because the instruction results in three data transfers. The three P1 elements all have the same left-hand key value so that they can all be associated with the single parent P0 element. If the instruction starts the data loads from base address `0x1000`, the three P1 elements are generated with the transfer indexes shown in Table 5-11.

Table 5-11 Example of three P1 elements generated from one P0 load instruction

Transfer index value	Meaning		
	Address	Offset of the address from the base address	Data value[a]
0	`0x1000`	None	r2
1	`0x1004`	One word	r5
2	`0x1008`	Two words	r6

a. The data values of the data transfers are output only if data value tracing is enabled.

The P1 elements might not be generated in the order shown. The transfer indexes indicate to a trace analyzer which data transfer the P1 element represents.

Note

- The example given in Table 5-11 is based on three word-sized data transfers. Not all data transfers are word-sized. Some might be halfword or doubleword transfers.
- For other instruction types, such as a single-transfer store-exclusive instruction or an instruction that performs both a read and write to a data address, the transfer index has a different meaning.
- For more information, see *P1 element transfer index meanings on page F-477*.

Occasions when P1 elements are traced without the address or endianness of the data transfer

A P1 element might be traced without address or endianness information if:

- Data address tracing is disabled for the particular type of data transfer, or for all data transfers, but data value tracing is enabled.
- The address is not known for the particular data transfer.
- The P1 element is generated to trace the success indicator of a store-exclusive instruction. See *Data trace behavior on tracing store-exclusive instructions on page 2-77*.
- The trace analyzer can infer the address of the data transfer from addresses that are contained in other P1 elements. This scenario might occur if, for example, one instruction results in multiple P1 elements. Figure 5-6 on page 5-230 shows a case where one instruction results in seven P1 elements.
- When TRCCONFIGR.DA is `0b0`, the address and endianness included in a P1 element are UNKNOWN.

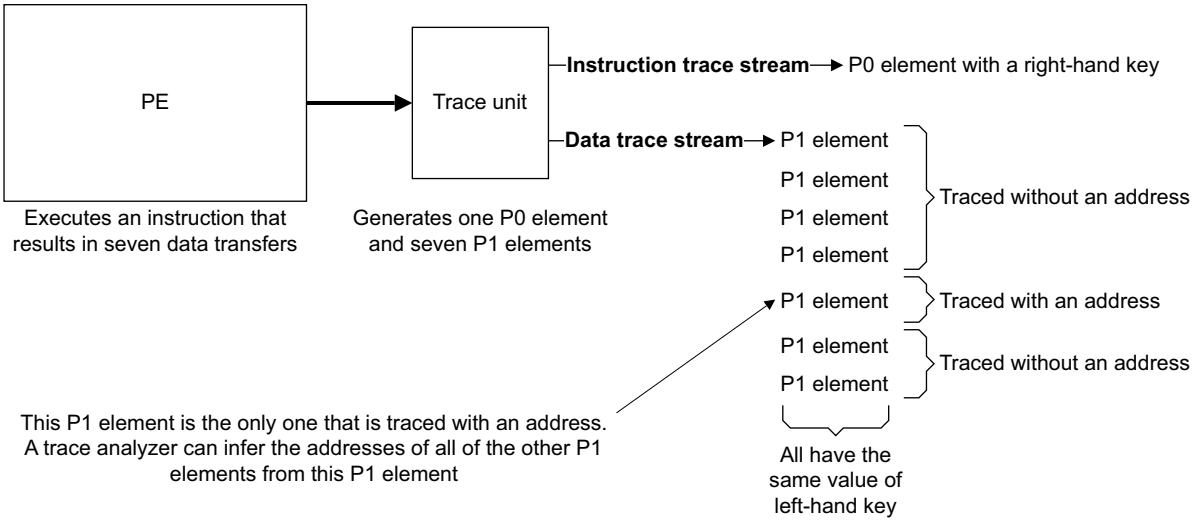


Figure 5-6 An example of when P1 elements might be traced without an address

Whenever a P1 element is traced without the address or endianness of a data transfer, the transfer index is always correct. This is to enable the data value to be located.

For accesses to the PPB space on an Armv6-M, Armv7-M, and Armv8-M PEs, the endianness traced is UNKNOWN, and must always be considered to be little-endian.

5.4.6 P2 Data Value (P2) data trace elements

Note

P2 elements are only generated if data tracing is supported and if data value tracing is enabled. See *Data value tracing on page 2-84*.

A P2 Data Value element, also referred to as a P2 element, contains:

- The data value of a data transfer that the PE has performed as a result of executing a P0 instruction, such as a load or store.
- A left-hand key, so that the P2 element can be associated with its parent P1 element.

Each P2 element only has one parent P1 element. A P2 element must be output before the value of its left-hand key can be reused by another P1 element. A P2 element is always traced after its parent P1 element.

5.4.7 Timestamp data trace element

A Timestamp data trace element inserts a global timestamp value into the data trace stream. A Timestamp data trace element is generated whenever a *timestamp request* occurs.

Whenever a timestamp request occurs, a timestamp is requested in both trace streams. This means that the trace unit generates both a Timestamp data trace element and a Timestamp instruction trace element.

See *Timestamp instruction trace element on page 5-215* for a list of events that result in timestamp requests.

The value of the timestamp is the time the timestamp was inserted into the data trace stream.

Although when a timestamp request occurs, the request is common to both trace streams, each trace stream can deal with the request independently.

The rule for inserting a Timestamp element in the data trace stream is as follows:

- When a trace unit receives a timestamp request then if necessary, for example to avoid a trace buffer overflow, it can delay the generation of a Timestamp data trace packet.

In addition, if a trace unit receives multiple timestamp requests close together then it might not generate a Timestamp data trace element for each request. For example, a trace unit can ignore the second request of two successive timestamp requests if both of the following are true:

- The second request is not caused as a result of a trace synchronization request.
- None of the following element types have been generated between the two requests:
  - *Atom element*.
  - P1 Data Address element.
  - P2 Data Value element.
  - Numbered Data Sync Mark element.
  - Event element.

A timestamp value of zero means that the timestamp value is UNKNOWN. This might be because the timestamp value is temporarily unavailable. Timestamping must always be supported by the trace unit and the system when data tracing is implemented.

5.4.8 Event data trace element

The ETMv4 architecture supports the tracing of *trace unit events* in the trace streams. A trace unit implementation provides support for:

- 1-4 events in the instruction trace stream, that each has a number from 0-3.
- One event in the data trace stream, that is event number zero.

An Event data trace element indicates that trace unit event number zero has occurred. The trace unit event that is event number zero can be chosen using the following procedure:

1. Select the *trace unit resource* that is to be used to activate the trace unit event, by programming a *trace unit resource selector*. A trace unit has 2-32 resource selectors, that each use one of the TRCRSCTLRn registers.
2. Select the programmed resource selector by programming TRCEVENTCTL0R.EVENT0.
3. Set TRCEVENTCTL1R.DATAEN to 1.

For more information, see:

- *Trace unit resources on page 4-139*.
- *Selecting trace unit resources on page 4-171*.
- *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177*.

5.4.9 Data Synchronization Marker (Data Sync Mark) data trace element

Data synchronization markers enable a trace analyzer to synchronize the data trace stream with the instruction trace stream.

Because their purpose is to enable synchronization of the two trace streams, the trace unit generates Data Synchronization Marker elements, also referred to as Data Sync Mark elements, only when data tracing is enabled.

For every Data Sync Mark element output in the data trace stream, there is a matching Data Sync Mark element output in the instruction trace stream.

The ETMv4 architecture provides the following types of Data Sync Mark elements:

- Numbered Data Sync Mark elements, which are output at trace synchronization points in the trace stream.
- Unnumbered Data Sync Mark elements, which are output between two numbered Data Sync Mark elements when required.

See *Synchronizing the instruction and data trace streams on page 2-43* for more information.

Chapter 6
Descriptions of Trace Protocols

This chapter describes the packets that comprise the two trace streams. It contains the following sections:

6.1 About the instruction trace and data trace protocol

An ETMv4 trace unit generates an instruction trace stream, and might generate a data trace stream. See Chapter 2 *About the Trace Streams*. This section describes the protocol that is used for these streams.

The protocol is a byte-based packet protocol, which means that each trace stream is constructed of multiple packets. Each packet contains one or more bytes of data.

A packet consists of a single header byte, followed by zero or more payload bytes. The number of payload bytes present in a packet depends on the packet type. For example:

- A Trace On packet consists of only a header byte, with no payload bytes.
- An A-Sync packet consists of a header byte plus 11 payload bytes.

For some packet types, such as the Commit packet, there is no upper limit on the number of payload bytes the packet can have. The size of each packet is determined when analyzing the packet.

The following section describes the protocol:

- *Packet types*.

6.1.1 Packet types

The ETMv4 architecture defines different *packet types*, each with a unique name. Some packet types have *subtypes*. Each packet type or subtype that can appear in a trace stream, except for extension packets, can be identified by a header that is unique within that stream. For example, the instruction trace packet type that is known as a Short Address packet type has five different subtypes, each with a unique header that identifies a particular instruction set. See *Short Address instruction trace packets on page 6-289*.

Extension packet types on page 6-238 describes how extension packets are identified.

The packet types are grouped into categories that indicate their function. For example:

- The Mispredict packet type is part of the Speculation resolution category, because its function is to signify that the status of a previously traced branch instruction has been mispredicted.
- The A-Sync packet type is part of the Synchronization category, because its function is to enable a trace analyzer to synchronize with the instruction or data trace stream.

Table 6-1 on page 6-235 shows which packet types belong to which category in the instruction trace stream. Table 6-2 on page 6-237 shows which packet types belong to which category in the data trace stream.

Some packet types can occur in both trace streams. Usually the function of the packet types is the same for both trace streams. However, for some of these packet types the function differs slightly, depending on which trace stream the packet is in. For example, a Trace Info packet:

- In the instruction trace stream, provides trace setup information and indicates a point in the instruction trace stream where analysis of the trace stream can begin.
- In the data trace stream, only indicates a point in the data trace stream where analysis of the trace stream can begin.

The format of a packet type can depend on which trace stream it is in. For example, an Unnumbered Data Synchronization Marker packet:

- In the data trace stream, always has the header `0b00000001`.
- In the instruction trace stream, can have either the header `0b00101100`, or a header in the range indicated by `0b001010xx`.

Although the header bytes of the different packet types are unique within a single trace stream, the meaning of some depends on which trace stream it appears in. For example, the header byte `0b00000001`:

- In the instruction trace stream, identifies a Trace Info packet.
- In the data trace stream, identifies an Unnumbered Data Synchronization Marker packet.

This means that a trace analyzer must be aware of which trace stream it is looking at when analyzing the trace, and that the trace streams cannot be combined in a single buffer unless the combination process includes a method of distinguishing the stream from which each packet originates.

Instruction trace stream packet types

Table 6-1 shows which packet types are output in the instruction trace stream, in alphabetical order.

Table 6-1 Packet types that are output in the instruction trace stream

Category	Packet type	Subtypes	Header
Address and context tracing	Address with Context	32-bit IS0 Long	0b10000010
		32-bit IS1 Long	0b10000011
		64-bit IS0 Long	0b10000101
		64-bit IS1 Long	0b10000110
	Context	-	0b1000000x
	Exact Match Address	-	0b10010000 to 0b10010010
	Long Address	32-bit IS0 Long	0b10011010
		32-bit IS1 Long	0b10011011
		64-bit IS0 Long	0b10011101
		64-bit IS1 Long	0b10011110
	Short Address	IS0 Short	0b10010101
		IS1 Short	0b10010110
Atom packets	Atom Format 1	-	0b1111011x
	Atom Format 2	-	0b110110xx
	Atom Format 3	-	0b11111xxx
	Atom Format 4	-	0b110111xx
	Atom Format 5	-	0b11110101, 0b11010101, 0b11010110, 0b11010111
	Atom Format 6	-	0b11000000 to 0b11010100 0b11100000 to 0b11110100

Table 6-1 Packet types that are output in the instruction trace stream (continued)

Category	Packet type	Subtypes	Header
Conditional instruction tracing	Conditional Flush	-	0b01000011
	Conditional Instruction Format 1	-	0b01101100
	Conditional Instruction Format 2	-	0b01000000, 0b01000001, 0b01000010
	Conditional Instruction Format 3	-	0b01101101
	Conditional Result Format 1	Single payload sequence	0b0110111x
		Double payload sequence	0b011010xx
	Conditional Result Format 2	-	0b01001xxx
	Conditional Result Format 3	-	0b0101xxxx
	Conditional Result Format 4	-	0b010001xx
Cycle counting	Cycle Count Format 1	-	0b0000111x
	Cycle Count Format 2	-	0b0000110x
	Cycle Count Format 3	-	0b0001xxxx
Data synchronization markers	Numbered Data Synchronization Marker	-	0b00100xxx
	Unnumbered Data Synchronization Marker	-	0b00101000, 0b00101001, 0b00101010, 0b00101011, 0b00101100
Ignore packets	Ignore	-	0b01110000
Event tracing	Event	-	0b01110001 to 0b01111111
Exception	Exception	-	0b00000110
	Exception Return	-	0b00000111
Function	Function Return	-	0b00000101
Global timestamping	Timestamp	-	0b0000001x
Q packets	Q	-	0b1010xxxx
Speculation resolution	Cancel Format 1	-	0b0010111x
	Cancel Format 2	-	0b001101xx
	Cancel Format 3	-	0b00111xxx
	Commit	-	0b00101101
	Mispredict	-	0b001100xx

Table 6-1 Packet types that are output in the instruction trace stream (continued)

Category	Packet type	Subtypes	Header
Synchronization	A-Sync (extension packet)[a]	-	0b00000000
	Branch Future Flush (extension packet)[a]	-	0b00000000
	Discard (extension packet)[a]	-	0b00000000
	Overflow (extension packet)[a]	-	0b00000000
	Trace Info	-	0b00000001
	Trace On	-	0b00000100
	Resynchronization	-	0b00001000

a. This is part of the Extension packets. See *Extension packets in the instruction trace stream on page 6-252*.

Data trace stream packet types

Table 6-2 shows which packets are output in the data trace stream, in alphabetical order.

Table 6-2 Packet types that are output in the data trace stream

Category	Packet type	Subtypes	Header
Data synchronization markers	Numbered Data Synchronization Marker (extension packet)[a]	-	0b00000000
	Unnumbered Data Synchronization Marker	-	0b00000001
Ignore packets	Ignore	-	0b00000101
Event tracing	Event	-	0b00000100
Global Timestamping	Timestamp	-	0b00000010
P1 data address tracing	P1 Format 1	-	0b0111xxxx
	P1 Format 2	-	0b10xxxxxx
	P1 Format 3	-	0b000101xx
	P1 Format 4	-	0b0110xxxx
	P1 Format 5	-	0b11111xxx
	P1 Format 6	-	0b1111011x
	P1 Format 7	-	0b11110101
P2 data value tracing	P2 Format 1	-	0b0010xxxx
	P2 Format 2	-	0b00110xxx
	P2 Format 3	-	0b010xxxxx
	P2 Format 4	-	0b000100xx
	P2 Format 5	-	0b00011xxx
	P2 Format 6	-	0b00111xxx
Suppression	Suppression	-	0b00000011

Table 6-2 Packet types that are output in the data trace stream (continued)

Category	Packet type	Subtypes	Header
Synchronization	A-Sync (extension packet)	-	0b00000000
	Discard (extension packet)[a]	-	0b00000000
	Overflow (extension packet)[a]	-	0b00000000
	Trace Info (extension packet)[a]	-	0b00000000

a. This is part of the Extension packets. See *Extension packets in the instruction trace stream on page 6-252*.

Extension packet types

Most packet types in a trace stream can be identified from their unique header byte. However, some packet types share the same header byte as other packet types, and for these, the first payload byte is required to identify the packet type. Although these packets, like any other, are grouped into categories according to their function, they are also known as Extension packets. All Extension packets, regardless of which trace stream they are in, use the header byte 0b00000000, and the first payload byte then identifies the packet type.

Table 6-3 shows what packet types are Extension packets, for both the instruction trace stream and the data trace stream.

Table 6-3 Packet types that are Extension packets, for each trace stream

Trace stream	Category	Packet type
Instruction trace stream	Synchronization	A-Sync, Alignment Synchronization, packet
		Discard packet
		Overflow packet
		Branch Future Flush packet
Data trace stream	Synchronization	A-Sync, Alignment Synchronization, packet
		Trace Info packet
		Discard packet
		Overflow packet
	Data synchronization markers	Numbered Data Synchronization Marker packet

For more information, see:

- *Extension packets in the instruction trace stream on page 6-252*.
- *Extension packets in the data trace stream on page 6-308*.

6.2 Trace analyzer state between receiving packets

The ETMv4 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.

The information that a trace analyzer must retain is described in the following two sections:

- *Trace analyzer state between receiving instruction trace packets*.
- *Trace analyzer state between receiving data trace packets on page 6-241*.

6.2.1 Trace analyzer state between receiving instruction trace packets

The information to be retained is shown in Table 6-4 and Table 6-5 on page 6-241.

The following types and enumerations are used:

- `enumeration atom {N, E};`
- `enumeration security_level {SECURE, NONSECURE};`
- `type address_reg_t is (bits(64) address, bits(2) IS);`

Table 6-4 shows information that dynamically changes during a trace run.

Table 6-4 Instruction trace information that dynamically changes, that a trace analyzer must retain between packets

Name	Description
`bits(64) timestamp`	The most recently broadcast global timestamp value. This is updated by the following packet types: • Timestamp. • Trace Info.

`address_reg_t address_regs[0]`
`address_reg_t address_regs[1]`
`address_reg_t address_regs[2]`
	Three address registers that store the three most recently broadcast instruction addresses and instruction sets. These are updated by the following packet types:
	• Address with Context.
	• Exact Match Address.
	• Exception.
	• Long Address.
	• Short Address.
	• Q.
	• Trace Info.

`bits(32) context_id`
`bits(32) vmid`
`bits(2) ex_level`
`security_level security`
`boolean sixty_four_bit`
	Context registers that store the most recently broadcast context values. These are updated by the following packet types:
	• Address with Context.
	• Context.
	• Exception.
	• Trace Info.

Table 6-4 Instruction trace information that dynamically changes, that a trace analyzer must retain between packets

Name	Description
`integer curr_spec_depth`	The speculation depth. This is updated by the following packet types: • All types of Atom packet. • All types of Cycle count packet. • Cancel Format 1. • Cancel Format 2. • Cancel Format 3. • Commit. • Discard. • Exception. • Exception Return, when tracing Armv6-M, Armv7-M, or Armv8-M profile PEs. • Overflow. • Function Return. • Q. • Trace Info. In addition, the following packet types might also update `curr_spec_depth`: • Mispredict. • Unnumbered Data Sync Mark.
`integer p0_key`	The right-hand key value of the next P0 element. This is updated by the following packet types: • Trace Info. • All types of Atom packet. • Exception. • Exception Return, when tracing Armv6-M, Armv7-M, or Armv8-M profile PEs. • Cancel Format 1. • Cancel Format 2. • Cancel Format 3. • Q. In addition, the following packet types might also update `p0_key`: • Mispredict. • Unnumbered Data Sync Mark.
`integer cond_c_key`	The right-hand key value of the most recently traced Conditional Instruction element. This is updated by Conditional Instruction, Conditional Result, and Trace Info packet types.
`integer cond_r_key`	The left-hand key value of the most recently traced Conditional Result element. This is updated by Conditional Result and Trace Info packet types.

Table 6-5 shows information that is static for a particular implementation. This information can be found in trace ID Registers 8-13 in Chapter 7 *Register Descriptions*.

Table 6-5 Instruction trace information that is specific to an implementation, that a trace analyzer must retain between packets

Name	Description
`integer p0_key_max`	The number of P0 right-hand keys implemented. TRCIDR9.NUMP0KEY shows this. The value of `p0_key_max` is at least one. Therefore, an implementation contains at least one P0 right-hand key that can be used. The value of `p0_key` is between zero and `p0_key_max`−1. The value of `p0_key_max` is also the number of left-hand keys for P1 elements.
`integer cond_key_max_incr`	The number of normal right-hand keys that are implemented for Conditional Instruction elements. TRCIDR12.NUMCONDKEY shows the total number of right-hand keys that are implemented for Conditional Instruction elements, and TRCIDR13.NUMCONDSPEC shows the number of special right-hand keys that are implemented for Conditional Instruction elements. Therefore, the number of normal right-hand keys that are implemented for Conditional Instruction elements, `cond_key_max_incr`, can be calculated from TRCIDR12.NUMCONDKEY – TRCIDR13.NUMCONDSPEC.
`integer max_spec_depth`	The maximum speculation depth. TRCIDR8.MAXSPEC shows this.
`integer cc_threshold`	Cycle count threshold value[a]. This value is static during a trace run, because it is a threshold value that is set by programming TRCCCCTLR.THRESHOLD. A trace analyzer obtains the value from a Trace Info packet.

a. This value is static during a trace run. All other values mentioned in Table 6-4 on page 6-239, and the contents of all registers mentioned, change dynamically during a trace run.

6.2.2 Trace analyzer state between receiving data trace packets

The information that is required is shown in Table 6-6 on page 6-242 and Table 6-7 on page 6-244.

The following enumeration is used:

- `enumeration endian {LITTLE, BIG};`

Table 6-6 shows information that is provided by trace packets, and that dynamically changes during a trace run.

Table 6-6 Information about the data trace that dynamically changes, that the trace analyzer must retain between receiving packets

Name	Description
`bits(64) timestamp`	The most recently broadcast global timestamp value. This is updated by the following packets: • Timestamp. • Trace Info.

`bits(64) address_regs[0]`
`bits(64) address_regs[1]`
`bits(64) address_regs[2]`
	Three address registers which store recently broadcast data addresses. These are updated by the following packet types:
	• P1 Format 1.
	• P1 Format 2.
	• P1 Format 3.
	• P1 Format 4.
	• P1 Format 7.
	• Trace Info.

`endian endianness`	The last broadcast endianness for a P1 Data Address element. This is the endianness of the data address, and it is updated by the following packet types: • P1 Format 1. • P1 Format 2. • P1 Format 3. • P1 Format 4. • Trace Info.
`integer p1_left_key`	The left-hand key value of a P1 element. This is updated by the following packet types: • P1 Format 1. • P1 Format 2. • P1 Format 3. • P1 Format 4. • P1 Format 5. • P1 Format 6. • P2 Format 5. • P2 Format 6. • Trace Info.

Table 6-6 Information about the data trace that dynamically changes, that the trace analyzer must retain between receiving packets (continued)

Name	Description
`integer p1_right_key`	The right-hand key value of a P1 element. This is updated by the following packet types: • P1 Format 1[a]. • P1 Format 2. • P1 Format 3. • P1 Format 4. • P1 Format 5. • P1 Format 6[a]. • P2 Format 5. • P2 Format 6. • Trace Info.
`integer p2_left_key`	The left-hand key value of a P2 element. This is updated by the following packet types: • P2 Format 1. • P2 Format 2. • P2 Format 3. • P2 Format 4. • P2 Format 5. • P2 Format 6. • Trace Info.
`integer p1_index`	The index for a P1 element. This is updated by the following packet types: • P1 Format 1. • P1 Format 2. • P1 Format 3. • P1 Format 4. • P1 Format 5. • P1 Format 6. • P2 Format 5. • P2 Format 6. • Trace Info.

a. These packet types only update `p1_right_key` if the key value provided by the packet is not a special key.

Table 6-7 on page 6-244 shows information that is static for a particular implementation. This information can be found in the ID Registers 8-13, see *TRCIDR8, ID Register 8 on page 7-384* onwards.

Table 6-7 Information about the data trace that is specific to an implementation, that the trace analyzer must retain between receiving packets

Name	Description
`integer p1_right_key_max`	The number of normal right-hand keys that are implemented for P1 elements. The packet descriptions use normal key values from zero to `p1_right_key_max`−1. Key values equal to or greater than `p1_right_key_max` are special keys. TRCIDR10.NUMP1KEY shows the total number of right-hand keys that are implemented for P1 elements, and TRCIDR11.NUMP1SPC shows the number of special keys that are implemented for P1 elements. Therefore, the number of normal right-hand keys that are implemented for P1 elements, p1_right_key_max, can be calculated from TRCIDR10.NUMP1KEY – TRCIDR11.NUMP1SPC. This value is also the number of normal left-hand keys for P2 elements.
`integer p1_left_key_max`	This value is also the number of P0 right-hand keys, which are stored in `p0_key_max` for the instruction trace stream. See Table 6-5 on page 6-241. The number of left-hand keys that are implemented for P1 elements, as TRCIDR9.NUMP0KEY shows.

6.3 Packet header encodings summary tables

The following sections contain two tables, that summarize:

- *Instruction trace packet header encodings on page 6-246*.
- *Data trace packet header encodings on page 6-249*.

6.3.1 Instruction trace packet header encodings

Table 6-8 Instruction trace packet header encodings, in byte order

Category	Header byte	Packet name	Payload	Purpose
Extension packets	0b00000000	A-Sync	11 bytes	Identifies a packet boundary.
		Discard	1 byte	Indicates that tracing has become inactive.
		Overflow	1 byte	Indicates that a trace unit buffer overflow has occurred.
		Branch Future Flush	1 byte	Indicates a Branch Future Flush element.
Synchronization	See the Extension packets category[a]	A-Sync	11 bytes	See A-Sync in the Extension packets category.
		Discard	1 byte	See Discard in the Extension packets category.
		Overflow	1 byte	See Overflow in the Extension packets category.
	0b00000001	Trace Info	At least 1 byte[b]	Provides trace setup information.
	0b00001000	Resynchronization	None	Indicates a Resynchronization element.
Global timestamping	0b0000001x	Timestamp	1-11 bytes	Contains the value of the timestamp.
Synchronization (continued)	0b00000100	Trace On	None	Indicates that there has been a discontinuity in the trace stream.
Function Return	0b00000101	Function Return	None	Indicates that a function return instruction has caused a pop of data from the stack.
Exceptions	0b00000110	Exception	3-12 bytes	Indicates that an exception has occurred.
	0b00000111	Exception Return	None	Indicates a return from an exception handler.
Reserved	0b000010xx	-	-	Reserved.
Cycle counting	0b0000110x	Cycle Count Format 2	1 byte	Indicates a number of processor clock cycles between two Commit elements.
	0b0000111x	Cycle Count Format 1	At least 1 byte[b]	
	0b0001xxxx	Cycle Count Format 3	None	
Data synchronization markers	0b00100xxx	Numbered Data Sync Mark	None	Enables approximate correlation of the instruction trace stream with the data trace stream.
	0b001010xx 0b00101100	Unnumbered Data Sync Mark	None	Enables accurate synchronization of the instruction trace stream with the data trace stream.
Speculation resolution	0b00101101	Commit	At least 1 byte[b]	Indicates a number of Commit elements.

Table 6-8 Instruction trace packet header encodings, in byte order (continued)

Category	Header byte	Packet name	Payload	Purpose
Speculation resolution (continued)	0b0010111x	Cancel Format 1	At least 1 byte[b]	Indicates one or more Cancel elements followed by one Mispredict element.
	0b001100xx	Mispredict	None	Indicates 0-2 E or N *Atom elements* followed by one Mispredict element.
	0b001101xx	Cancel Format 2	None	Indicates zero or more E or N *Atom element*s followed by one Cancel element and one Mispredict element.
	0b00111xxx	Cancel Format 3	None	Indicates zero or one E *Atom element* followed by 2-5 Cancel elements and one Mispredict element.
Conditional instructions tracing	0b0100000x 0b01000010	Conditional Instruction Format 2	None	Indicates 1-2 C elements. The packet also contains information about the keys for these elements.
	0b01000011	Conditional Flush	None	Indicates a Conditional Flush element.
	0b0100010x 0b01000110	Conditional Result Format 4	None	Indicates one R element, whose key is one less than for the previous R element.
	0b01000111	-	-	Reserved.
	0b0100100x 0b01001010	Conditional Result Format 2	None	Indicates one token, that indicates one or more C elements followed by an R element.
	0b01001011	-	-	Reserved.
	0b0100110x 0b01001110	Conditional Result Format 2	None	Indicates one token, that indicates or more C elements followed by an R element.
	0b01001111	-	-	Reserved.
	0b0101xxxx	Conditional Result Format 3	1 byte	Indicates one or more tokens. Each token indicates one or more C elements followed by an R element.
	0b01100000 to 0b01100111	-	-	Reserved.
	0b011010xx	Conditional Result Format 1	At least 1 byte[b]	Indicates zero or more C elements followed by an R element. This packet type can contain two sets of payload bytes.
	0b01101100	Conditional Instruction Format 1	At least 1 byte[b]	Indicates one C element, and contains the right-hand key value for that C element.
Conditional instructions tracing (continued)	0b01101101	Conditional Instruction Format 3	1 byte	Indicates 1-64 C elements. The packet also contains information about the keys for these elements.
	0b0110111x	Conditional Result Format 1	At least 1 byte[b]	Indicates zero or more C elements followed by an R element. This packet type can contain two sets of payload bytes.
Ignore packets	0b01110000	Ignore	None	The Ignore packet is ignored by a trace analyzer, and might be used to pad the trace stream to a convenient boundary.

Table 6-8 Instruction trace packet header encodings, in byte order (continued)

Category	Header byte	Packet name	Payload	Purpose
Event tracing	0b01110001 to 0b01111111	Event	None	Indicates 1-4 Event elements.
Address and context tracing	0b1000000x	Context	0-9 bytes	Contains information about the context in which instructions are being executed.
	0b10000010 to 0b10000011 and 0b10000101 to 0b10000110	Address with Context	4 address bytes or 8 address bytes and 1-6 context bytes	Contains both the address and instruction set of the next instruction to be executed, and in addition, contains new context information.
	0b10001xxx	-	-	Reserved.
	0b10010000 to 0b10010010	Exact Match Address	None	Indicates that the address and instruction set of the next instruction to be executed is identical to that contained in an Address packet that has recently been output.
	0b10010011 to 0b10010100	-	-	Reserved.
	0b10010101 to 0b10010110	Short Address	1 byte or 2 bytes	Contains the address and instruction set of the next instruction to be executed. This packet type can contain up to 17 bits of the address.
	0b10010111 to 0b10011001	-	-	Reserved
	0b10011010 to 0b10011011 and 0b10011101 to 0b10011110	Long Address	4 bytes or 8 bytes	Contains the address and instruction set of the next instruction to be executed. This packet type can contain up to 64 bits of the address.
	0b10011100	-	-	Reserved
	0b10011111	-	-	Reserved
Q packets	0b1010xxxx	Q	At least 1 byte[b]	Contains a count of instructions and the address at which tracing resumes.
Reserved	0b1011xxxx	-	-	-

Table 6-8 Instruction trace packet header encodings, in byte order (continued)

Category	Header byte	Packet name	Payload	Purpose
Atom packets	0b11000000 to 0b11010100	Atom Format 6	None	Indicates 4-24 *Atom elements*.
	0b11010101 to 0b11010111	Atom Format 5	None	Indicates a sequence of five *Atom elements*.
	0b110110xx	Atom Format 2	None	Indicates two *Atom elements*.
	0b110111xx	Atom Format 4	None	Indicates a sequence of four *Atom elements*.
Atom packets (continued)	0b11100000 to 0b11110100	Atom Format 6	None	Indicates 4-24 *Atom elements*.
	0b11110101	Atom Format 5	None	Indicates a sequence of five *Atom elements*.
	0b1111011x	Atom Format 1	None	Indicates one *Atom element.*
	0b11111xxx	Atom Format 3	None	Indicates three *Atom elements*.

a. These synchronization packets are Extension packets. A trace analysis tool requires the first two bytes of an Extension packet, the header byte and the first payload byte, to identify the packet type. All other packet types can be identified from only their header bytes.

b. Where the payload for a packet is denoted as at least 1 byte, there is no upper limit on the number of payload bytes.

6.3.2 Data trace packet header encodings

Table 6-9 Data trace packet header encodings, in byte order

Category	Header byte	Packet name	Payload	Purpose
Extension packets	0b00000000	A-Sync	11 bytes	Identifies a packet boundary.
		Trace Info	At least 1byte[b]	Provides a point in the data trace stream where analysis of the trace stream can begin.
		Discard	1 byte	Indicates that tracing has become inactive.
		Overflow	1 byte	Indicates that a trace unit buffer overflow has occurred.
		Numbered Data Sync Mark	1 byte	Enables approximate correlation of the data trace stream with the instruction trace stream.
Data synchronization markers	See the Extension packets category.[a]	Numbered Data Sync Mark	1 byte	See Numbered Data Sync Marker in the Extension packets category.
	0b00000001	Unnumbered Data Sync Mark	None	Enables accurate synchronization of the data trace stream with the instruction trace stream.

Table 6-9 Data trace packet header encodings, in byte order (continued)

Category	Header byte	Packet name	Payload	Purpose
Global timestamping	0b00000010	Timestamp	1-9 bytes	Contains the value of the timestamp.
Synchronization	See the Extension packets category.[a]	A-Sync	11 bytes	See A-Sync in the Extension packets category.
		Trace Info	At least 1 byte[b]	See Trace Info in the Extension packets category.
		Discard	1 byte	See Discard in the Extension packets category.
		Overflow	1 byte	See Overflow in the Extension packets category.
Suppression	0b00000011	Suppression	None	Indicates that some of the data trace has been lost, because there is a risk that the data trace buffer in the trace unit might overflow.
Event tracing	0b00000100	Event	None	Indicates one Event element.
Ignore packets	0b00000101	Ignore	None	The Ignore packet is ignored by a trace analyzer, and might be used to pad the trace stream to a convenient boundary.
Reserved	0b00000110 to 0b00001111	-	-	Reserved.
P2 data value packets	0b000100xx	P2 Format 4	None	Indicates one P2 element, when the data value is in the range 1-4. Also indicates the left-hand key value.
P1 data address packets	0b000101xx	P1 Format 3	1 byte	Contains bits[9:2] of the word-aligned address of one P1 element. Also indicates the key and transfer index values.
P2 data value packets (continued)	0b00011xxx	P2 Format 5	4 bytes or 8 bytes	Indicates one P2 element, when the data value is either 32bits or 64 bits. Also indicates up to four P1 elements when the trace analyzer does not require the address information contained in those elements.
	0b0010xxxx	P2 Format 1	At least 1 byte[b]	Contains the full data value and full left-hand key value for one P2 element.
	0b00110xxx	P2 Format 2	2 bytes	Indicates one P2 element, when the data value is 16 bits or less. Also indicates the left-hand key value.
	0b00111xxx	P2 Format 6	8 bytes	Indicates two P2 elements, when the data value contained in both is 32 bits. Also indicates up to four P1 elements when the trace analyzer does not require the address information that is contained in those elements. See *P2 Format 6 data trace packet on page 6-332*.
	0b010xxxxx	P2 Format 3	4 bytes or 8 bytes	Indicates one P2 element, when the data value is either 32 or 64 bits. Also indicates the left-hand key value.

Table 6-9 Data trace packet header encodings, in byte order (continued)

Category	Header byte	Packet name	Payload	Purpose
P1 data address packets (continued)	0b0110xxxx	P1 Format 4	None	Contains bits[3:2] of the halfword-aligned address of one P1 element. Also indicates the key and transfer index values.
	0b0111xxxx	P1 Format 1	1 byte	Contains the full address, full right-hand and left-hand key values, and full transfer index value, of one P1 element.
	0b10xxxxxx	P1 Format 2	None	Contains bits[5:2] of the word-aligned address of one P1 element. Also indicates the key and transfer index values.
	0b11000000 to 0b11110100	-	-	Reserved.
	0b11110101	P1 Format 7	1 byte	Updates the address given in the most recently traced P1 element. Bits[63:56] can be updated.
	0b1111011x	P1 Format 6	At least 1 byte[b]	Indicates one P1 element, when the trace analyzer does not require the address information contained in that element. Also contains a full right-hand key value.
	0b11111xxx	P1 Format 5	None	Indicates the key and index values for 1-4 P1 elements, when the trace analyzer does not require the address information contained in those elements.

a. These packets are Extension packets. A trace analysis tool requires the first two bytes of an Extension packet, the header byte and the first payload byte, to identify the packet type. All other packet types can be identified from only their header byte.

b. Where the payload for a packet is denoted as at least 1 byte, there is no upper limit on the number of payload bytes.

6.4 Descriptions of instruction trace packets

The following sections describe the packets that comprise the instruction trace stream:

- *Extension packets in the instruction trace stream*.
- *Packets associated with synchronization between the trace unit and a trace analyzer on page 6-253*.
- *Global timestamping on page 6-259*.
- *Packets associated with exceptions on page 6-261*.
- *Cycle Count packets on page 6-267*.
- *Data Synchronization Marker (Data Sync Mark) instruction trace packets on page 6-270*.
- *Speculation resolution packets on page 6-271*.
- *Packets associated with tracing conditional instructions on page 6-275*.
- *Ignore packet on page 6-284*.
- *Event tracing instruction trace packet on page 6-285*.
- *Address and Context tracing packets on page 6-285*.
- *Atom instruction trace packets on page 6-298*.
- *Q instruction trace packet on page 6-304*.

6.4.1 Extension packets in the instruction trace stream

In Table 6-8 on page 6-246, three of the packet types that are in the synchronization category, the A-Sync packet, the Discard packet, and the Overflow packet, are extension packets. In general, a packet type can be identified from its unique header byte. However, in the case of an extension packet, the header byte defines the packet as an extension packet, and it is the first payload byte that identifies the packet type. The header byte of an extension packet, regardless of what type of packet it is, always has the value `0b00000000`, as shown in Table 6-10.

Table 6-10 Extension packets

Packet name	Header byte	First payload byte	Purpose
-	`0b00000000`	`0bxxxxxxx0`	Reserved, except `0b00000000`
A-Sync		`0b00000000`	Identifies a packet boundary
-		`0b00000001`	Reserved
Discard		`0b00000011`	Indicates that tracing has become inactive
Overflow		`0b00000101`	Indicates a trace unit buffer overflow
Branch Future Flush		`0b00000111`	Indicates a Branch Future Flush
-		`0b00001xx1`	Reserved
-		`0b0001xxx1`	Reserved
-		`0b001xxxx1`	Reserved
-		`0b01xxxxx1`	Reserved
-		`0b1xxxxxx1`	Reserved

A trace analysis tool therefore requires the first two bytes of an extension packet to identify the packet type, whereas for all other packet types, a trace analysis tool requires only the header byte to identify the packet type.

6.4.2 Packets associated with synchronization between the trace unit and a trace analyzer

The packet types that comprise this category are:

- *Alignment Synchronization (A-Sync) instruction trace packet*.
- *Trace Info instruction trace packet on page 6-254*.
- *Trace On instruction trace packet on page 6-257*.
- *Discard instruction trace packet on page 6-258*.
- *Overflow instruction trace packet on page 6-258*.

Alignment Synchronization (A-Sync) instruction trace packet

A trace analyzer uses an Alignment Synchronization instruction trace packet, also known as an A-Sync instruction trace packet, to synchronize with the instruction trace stream. This packet type is only used for synchronization purposes and does not indicate any trace elements.

Note

An A-Sync instruction trace packet is an extension packet. See *Extension packets in the instruction trace stream on page 6-252*.

The A-Sync instruction trace packet is a unique sequence of bits that identifies the boundary of another packet. The unique sequence is a header byte, `0b00000000`, followed by ten payload bytes of `0b00000000` and one final payload byte of `0b10000000`, as shown in Figure 6-1. Any byte that follows this unique sequence of bits is the header byte of a new packet.

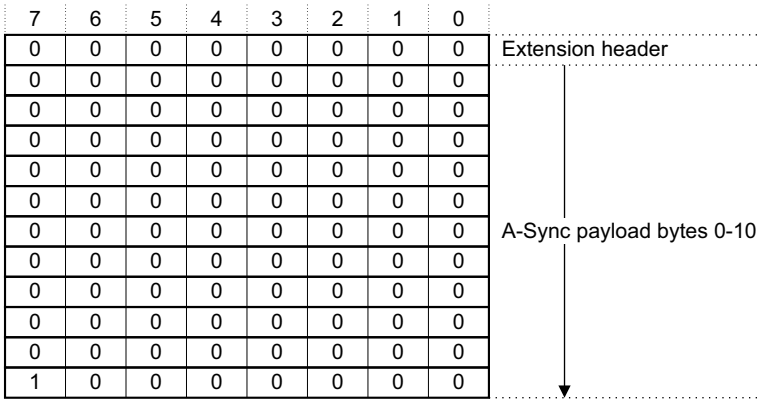


Figure 6-1 A-Sync instruction trace packet

Whenever the trace unit is first enabled, at the beginning of each new trace run, the first packet output in the instruction trace stream is an A-Sync instruction trace packet. Therefore, on enabling the trace unit this packet type must be the first packet that a trace analyzer searches for, so that it can identify where the next packet starts and when to start decompression of the instruction trace stream. The packet type that follows the A-Sync instruction trace packet is a Trace Info instruction trace packet, that contains information about the programming of the trace and provides a point in the instruction trace stream where analysis of the trace stream can begin. See *Trace Info instruction trace packet on page 6-254*.

An A-Sync instruction trace packet is also output:

- Periodically, based on trace synchronization requests. That is, the trace unit can be programmed to generate trace synchronization requests on a periodic basis, so that the trace streams can be analyzed if either trace stream has been stored in a circular trace buffer. In addition, the number of bytes of trace that are output between the trace synchronization requests can be specified. The field that enables this functionality is TRCSYNCPR.PERIOD.
- After a trace unit buffer overflow. If an overflow happens, a trace synchronization request automatically occurs.

Note

The data trace stream also contains A-Sync packets, and whenever a trace synchronization request occurs, synchronization is requested in both trace streams. However, the trace unit might not output an A-Sync packet in each of the streams simultaneously. For example, if outputting an A-Sync packet in one of the trace streams would risk an overflow of one of the trace unit buffers, an A-Sync packet might appear in one trace stream before the other.

For more information about synchronizing a trace analyzer with the trace streams, see *Synchronization with a trace analyzer on page 2-65*.

Trace Info instruction trace packet

A Trace Info instruction trace packet indicates to a trace analyzer that the trace unit has generated a Trace Info instruction trace element. See *Trace Info instruction trace element on page 5-188*.

A Trace Info packet contains information about the setup of the trace. This information shows:

- Whether load instructions are traced explicitly, and whether store instructions are traced explicitly.
- Whether cycle counting is enabled, and if enabled, the cycle count threshold.
- What is enabled regarding the tracing of conditional non-branch instructions.
- The value of the right-hand key for the next P0 element.
- What the speculation depth is.

After the trace unit is first enabled, an A-Sync packet is output, followed by a Trace Info packet. When a trace analyzer has found the A-Sync packet, it must search for the Trace Info packet to obtain information about the setup of the trace, then it can begin analyzing program execution when a Context packet and Address packet are output. A Context packet contains information about the context in which instructions are being executed, and an Address packet indicates a point in the program code where the analysis of program execution is to begin. For a description of the Context and Address instruction trace elements, see *Context instruction trace element on page 5-211* and *Address instruction trace element on page 5-209*. Address and Context instruction trace packets are described in *Address and Context tracing packets on page 6-285*.

A Trace Info packet consists of a header byte, `0b00000001`, plus a variable number of payload sections, where each payload section is made up of a number of bytes. The payload sections are:

1. Payload control, PLCTL. This section is always present.
2. Trace Info, INFO. This section might be present.
3. P0 key, KEY. This section might be present.
4. Speculation depth, SPEC. This section might be present.
5. Cycle count threshold, CYCT. This section might be present.

The first payload section, PLCTL, is always present and indicates which of the other payload sections are also present. For example, bit[0] indicates if the INFO payload section is present. If bit[0] of PLCTL is set to 1, then the INFO payload section is present. If bit[0] of PLCTL is set to 0, then the INFO payload section is not present.

Each byte in each payload section contains a continuation bit. This is the C field that is bit[7] in each payload byte. If this bit is set to 1, then another byte follows in that section. Otherwise, if the continuation bit is set to 0, the byte is the last byte in the section. However, other sections might follow, depending on which sections are indicated as present in PLCTL. Figure 6-2 on page 6-255 shows the format of a Trace Info packet in the instruction trace stream.

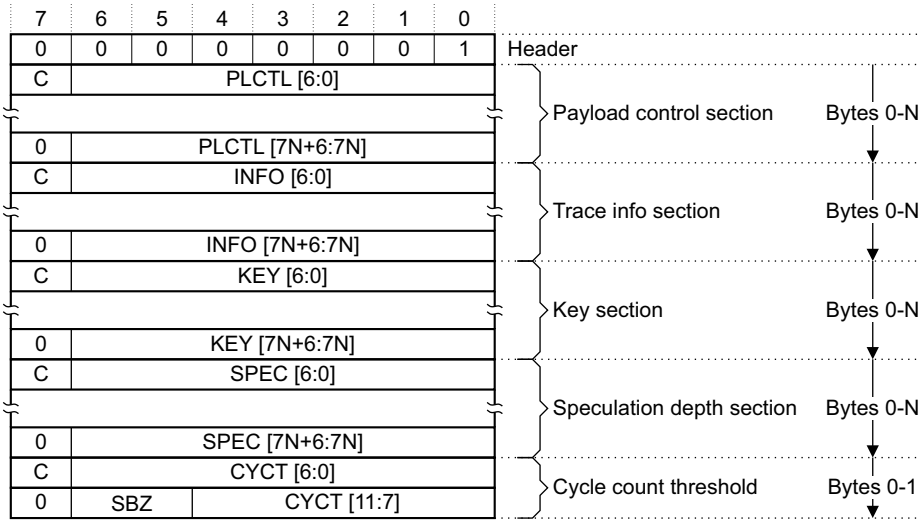


Figure 6-2 Trace Info instruction trace packet

The fields in the payload sections of a Trace Info packet are:

PLCTL This is the payload control field. The bits in this field indicate which other payload sections are present, as follows:

[0] INFO section.

[1] KEY section.

[2] SPEC section.

[3] CYCT section.

The possible values are:

0 The section is not present.

1 The section is present.

All other bits in this field are reserved.

If any bits of the PLCTL field are not output, their value is zero. A trace unit must not output more than 1 PLCTL field in a Trace Info packet.

INFO This field contains information about the programming of the trace, that is set before a trace run. See Table 6-11 on page 6-256 for a description of bits[5:0] in this field. All other bits are reserved.

If any bits of the INFO field are not output, their value is zero.

A trace unit must not output more than one INFO field in a Trace Info packet.

KEY The KEY payload section might not be included in the Trace Info packet if either:

- Data tracing is not implemented.
- Data tracing is implemented but not enabled.

See *Separate instruction and data trace streams on page 2-33*.

If the KEY payload section is present in either of these scenarios, then it must not be relied on.

If data tracing is implemented and enabled, the value contained in this field is the value of the right-hand key for the next P0 element. For example, if the value of the right-hand key for the next P0 element is the number three, then this field consists of one byte that is `0b00000011`.

If any bits of the KEY field are not output, their value is zero.

A trace unit must not output more KEY bytes than are required to indicate `p0_key_max`–1. For example, if `p0_max_key` is 32, then no more than one KEY byte must be output.

Note

If data tracing is supported and enabled but no KEY field is present, then it means that the value of the key for the next P0 element is zero.

SPEC This is the speculation depth field, and the value given in it, `curr_spec_depth`, is the number of P0 elements that are speculative.

If the SPEC payload section is not present in the packet, then the value of `curr_spec_depth` is zero.

If any bits of the SPEC field are not output, their value is zero.

A trace unit must not output more SPEC bytes than are required to indicate `max_spec_depth`. For example, if `max_spec_depth` is 32, then no more than one SPEC byte must be output.

CYCT The value shown in this field is the cycle count threshold value, `cc_threshold`. If cycle counting is enabled, this is the threshold above which Cycle Count packets are output. The threshold value can be set by programming the TRCCCCTLR.

If cycle counting is disabled, then the CYCT payload section is usually not present in the packet. However, there are occasions when the CYCT section might be present when cycle counting is disabled, and if so, the value of CYCT must be ignored.

If any bits of the CYCT field are not output, their value is zero.

C The continuation bit. If a byte in a section has this bit set to 1, then another byte follows in the same section. If a byte in a section has this bit set to 0, then it is the last byte in the section.

Table 6-11 INFO field in the Trace Info instruction trace packet

INFO bits	Name	Description
[0]	`cc_enabled`	This shows whether cycle counting is enabled: **0** Cycle counting is disabled. If the CYCT section is present in the packet, it must be ignored. **1** Cycle counting is enabled, and the value given in the CYCT field is the cycle count threshold value, `cc_threshold`.
[3:1]	`cond_enabled`	These show what is programmed with regard to conditional non-branch instruction tracing. The possible values are: `0b000` Tracing of conditional non-branch instructions is disabled. `0b001` Conditional load instructions are traced. `0b010` Conditional store instructions are traced. `0b011` Conditional load and store instructions are traced. `0b100` Reserved. `0b101` Reserved. `0b110` Reserved. `0b111` All conditional non-branch instructions are traced.
[4]	`p0_load`	This shows whether load instructions in the trace stream are traced explicitly: **0** Load instructions are not traced explicitly. **1** Load instructions are traced explicitly. That is, load instructions result in P0 elements.
[5]	`p0_store`	This shows whether store instructions in the trace stream are traced explicitly: **0** Store instructions are not traced explicitly. **1** Store instructions are traced explicitly. That is, store instructions result in P0 elements.

The TraceInfoPacket() function for the instruction trace stream is:

```
// TraceInfoPacket()
//=================

TraceInfoPacket()
    timestamp = 0;
    for I = 0 to 2
        address_regs[I].address = 0;
        address_regs[I].IS = 0;
    context_id = 0;
    vmid = 0;
    ex_level = 0;
    security = SECURE;
    sixty_four_bit = 0;
    cc_threshold = if INFO.cc_enabled then UInt(CYCT) else 0;
    curr_spec_depth = UInt(SPEC);
    p0_key = UInt(KEY);
    cond_c_key = 0;
    cond_r_key = 0;
    emit(trace_info_element(INFO.cc_enabled,
            cc_threshold,
            INFO.cond_enabled,
            INFO.p0_load,
            INFO.p0_store,
            curr_spec_depth
           ) );
```

Trace On instruction trace packet

A Trace On instruction trace packet indicates to a trace analyzer that the trace unit has generated a Trace On instruction trace element. See *Trace On instruction trace element on page 5-190*.

The Trace On packet indicates that there has been a discontinuity in the instruction trace stream. It is output whenever a gap occurs, after the gap occurs. This means that a Trace On packet is output:

- When the trace unit is first enabled, after the first A-Sync and Trace Info packets but before any packet types that indicate any P0 elements.
- After an overflow of either trace buffer in the trace unit. 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 trace 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 stream. The Trace On packet must be output before any packet types that indicate any P0 elements.

A Trace On packet consists of only a header byte, as shown in Figure 6-3.

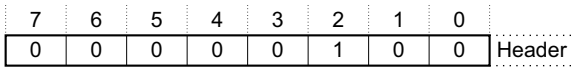


Figure 6-3 Trace On instruction trace packet

The TraceOnPacket() function is:

```
//TraceOnPacket()
//==============

TraceOnPacket()
    emit(trace_on_element());
    emit(conditional_flush_element());
    emit(branch_future_flush_element());
```

Discard instruction trace packet

A Discard instruction trace packet indicates to a trace analyzer that the trace unit has generated a Discard instruction trace element. See *Discard instruction trace element on page 5-191*.

Note

A Discard packet in the instruction trace stream is an extension packet. See *Extension packets in the instruction trace stream on page 6-252*.

A Discard instruction trace packet indicates that tracing has become inactive while uncommitted P0 elements remain. For example, if tracing becomes inactive as a result of a trace buffer overflow, and some speculative P0 elements remain, then a Discard packet is output to signal that these speculative elements must be discarded because their statuses cannot be resolved.

A Discard instruction trace packet consists of a header byte plus one payload byte, as shown in Figure 6-4.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Extension header
0	0	0	0	0	0	1	1	Identifies the packet type as a Discard packet

Figure 6-4 Discard instruction trace packet

The `DiscardPacket()` function for the instruction trace stream is:

```
//DiscardPacket()
//==============

DiscardPacket()
    emit(discard_element());
    emit(conditional_flush_element());
    curr_spec_depth = 0;
```

Overflow instruction trace packet

An Overflow instruction trace packet indicates to a trace analyzer that the trace unit has generated all of the following:

- An Overflow element. See *Overflow instruction trace element on page 5-191*.
- A Discard element. See *Discard instruction trace element on page 5-191*.
- An F element. See *Conditional Flush (F) instruction trace element on page 5-220*.

Note

An Overflow instruction trace packet is an extension packet. See *Extension packets in the instruction trace stream on page 6-252*.

An Overflow instruction trace packet is output whenever the instruction trace buffer overflows, which means that some of the trace might be lost, and that tracing is inactive until the overflow condition clears.

If part of the trace stream is lost, then some status information for uncommitted P0 elements might be lost, and some Conditional Result (R) elements for some Conditional Instruction (C) elements might also be lost.

When an overflow occurs, the trace unit outputs an Overflow packet and then it must output packets, in the following order:

1. An Event or Discard packet, or another Overflow packet. This step is optional.
2. An A-Sync packet, so that the trace analyzer can resynchronize with the instruction trace stream.
3. A Trace Info packet, to provide the trace analyzer with up-to-date information about the trace, such as the speculation depth and the value of the key for the next P0 element.
4. A Trace On packet, to indicate a gap in the trace stream. The Trace On packet must be output before any packet types that indicate any P0 elements.

For a full description of trace unit behavior on a trace buffer overflow, see *Trace unit behavior on a trace buffer overflow on page 3-106*.

An Overflow instruction trace packet consists of a header byte plus one payload byte, as shown in Figure 6-5.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Extension header
0	0	0	0	0	1	0	1	Identifies the packet type as an Overflow packet

Figure 6-5 Overflow instruction trace packet

The `OverflowPacket()` function for the instruction trace stream is:

```
//OverflowPacket()
//===============

OverflowPacket()
    emit(overflow_element());
    emit(discard_element());
    emit(conditional_flush_element());
    curr_spec_depth = 0;
```

Resynchronization packet

The Resynchronization packet indicates a *Resynchronization element*. See *Resynchronization element on page 5-213*.

A Resynchronization packet consists of a header byte, as shown in Figure 6-6.

7	6	5	4	3	2	1	0	
0	0	0	0	1	0	0	0	Header

Figure 6-6 Resynchronization packet

The `ResynchronizationPacket()` function for the instruction trace stream is:

```
// ResynchronizationPacket()
// =========================

ResynchronizationPacket()
    emit(resynchronization_element());
```

6.4.3 Global timestamping

There is one packet type in this category:

- *Timestamp instruction trace packet*.

Timestamp instruction trace packet

If global timestamping is supported and enabled, the trace unit generates timestamp requests whenever certain events occur. Whenever a timestamp request is generated, the trace unit generates a Timestamp instruction trace element that results in a Timestamp instruction trace packet. See *Timestamp instruction trace element on page 5-215*.

A Timestamp instruction trace packet consists of:

- A one-byte header. This byte is always present.
- 1-9 bytes of timestamp value. This section is always present.
- If cycle counting is supported and enabled, 1-3 bytes of cycle count value. When cycle counting is enabled, the Timestamp packet must include a cycle count section, unless the cycle count is UNKNOWN.

The ETMv4 architecture permits maximum timestamp values of either 48 bits or 64 bits. If the PE does not implement Armv8.4-Trace, whether an implementation supports a maximum timestamp value of 48 bits or 64 bits is IMPLEMENTATION DEFINED. If Armv8.4-Trace is implemented, the size of the timestamp values traced must be 64 bits.

TRCIDR0.TSSIZE shows which maximum size is implemented.

Figure 6-7 shows the format of the Timestamp packet.

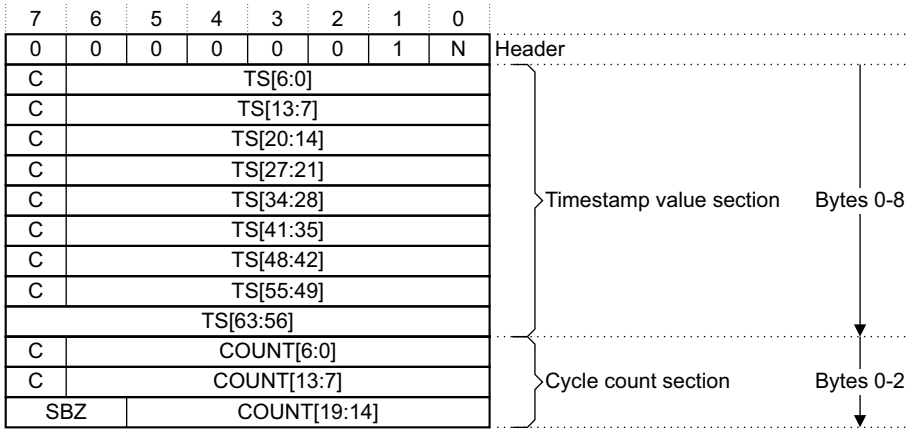


Figure 6-7 Timestamp instruction trace packet

The fields in the Timestamp packet are:

N The N bit in the Timestamp packet header indicates if the cycle count bytes are present in the packet:

0 Cycle count bytes are not present. The cycle count is UNKNOWN.

1 Cycle count bytes are present.

TS The Timestamp packet header is always followed by at least one byte of timestamp value. The timestamp value is compressed, so that the trace unit generates only enough bytes of timestamp to output the least significant bits that have changed since the value given in the previous Timestamp packet. Therefore, if any bits in this field are not output, they are either:

- The same value as they were in the previous Timestamp packet.
- Zero, if they have not been output since the most recent Trace Info packet.

COUNT If cycle counting is enabled, then the timestamp section of the packet is followed by at least one byte of cycle count. The value given in this field is the number of processor clock cycles between the most recent Cycle Count element, and the Atom, Exception, Numbered Data Sync marker or Event element that the timestamp value given in the TS field corresponds to.

A trace unit must not output more COUNT bytes than is required to indicate the maximum value of the cycle counter. For example, if the cycle counter is 12 bits, no more than two COUNT bytes must be output.

Note

Unlike the Cycle Count packets:

- The value of the cycle count given in a Timestamp packet is COUNT, that is, the cycle count is not offset by the cycle count threshold.
- COUNT does not affect the cumulative cycle count total.
- The value of COUNT can be zero.

C The continuation bit. If a byte in a section has this bit set to 1, then another byte follows in the same section. If a byte in a section has this bit set to 0, then it is the last byte in the section.

The `TimestampPacket()` function for the instruction trace stream is:

```
//TimestampPacket()
//================

TimestampPacket()
    timestamp = replace timestamp with new bits from TS, leaving other bits unchanged
    if N then
        emit(timestamp_element(UInt(timestamp),UInt(COUNT)));
    else
        emit(timestamp_element(UInt(timestamp),UNKNOWN));
```

6.4.4 Function Return packets

There is one packet type in this category:

- Function Return.

Function Return packet

The Function Return packet indicates the occurrence of a Function Return element. The Function Return packet is only generated for an Armv8-M PE, and only when data tracing is enabled.

If the Function Return element exceeds the maximum speculation depth, then the Function Return packet also implies a Commit element.

A Function Return packet consists of only a header byte, as shown in Figure 6-8.

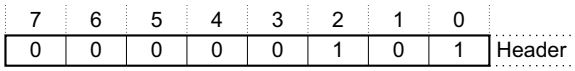


Figure 6-8 Function Return instruction trace packet

The `FunctionReturnPacket()` function for the instruction trace stream is:

```
//FunctionReturnPacket()
//======================

FunctionReturn ()
    emit(function_return_element(p0_key));
    p0_key = (p0_key + 1) MOD p0_key_max;
    curr_spec_depth = curr_spec_depth + 1;
    if (curr_spec_depth > max_spec_depth) then
        emit(commit_element(1));
        curr_spec_depth = curr_spec_depth - 1;

    emit(branch_future_flush_element());
```

6.4.5 Packets associated with exceptions

The packet types that comprise this category are:

- *Exception instruction trace packet on page 6-262*.
- *Handling Exception instruction trace packets on page 6-266*.
- *Exception Return instruction trace packet on page 6-266*.

Exception instruction trace packet

Whenever the PE takes an exception, the trace unit generates an *Exception element*.

For more information, see *Exception instruction trace element on page 5-196*.

An Exception packet contains information on:

- The type of exception. For example, it might be a PE reset, an IRQ, or another type of exception. Possible exception types for Arm PEs can be found in Table 6-12 on page 6-263 and in Table 6-13 on page 6-264.
- An Address packet.
- Whether an Address element and an optional Context element are implied before the *Exception element*.

The Exception packet contains an address. This means that execution has continued from the target of the most recent P0 element, up to, but not including, that address, and a trace analyzer must analyze each instruction in this range.

The Exception packet might also indicate that an Address element and an optional Context element are generated before the *Exception element*. This might be used when no instructions have executed between the target of the previous P0 element and the exception. In this scenario, the target address and context of the previous P0 element is the same as the preferred exception return address. When this occurs, the Exception packet includes a single address field that indicates both the target of the previous P0 element and the preferred exception return address.

If the *Exception element* means that the maximum P0 speculation depth is exceeded, then the Exception packet also implies a Commit element.

Note

The address that is provided in an Exception packet is the preferred exception return address. See Table 5-4 on page 5-198 for a summary of the preferred exception return addresses for each exception type.

The format of an Exception packet is shown in Figure 6-9.

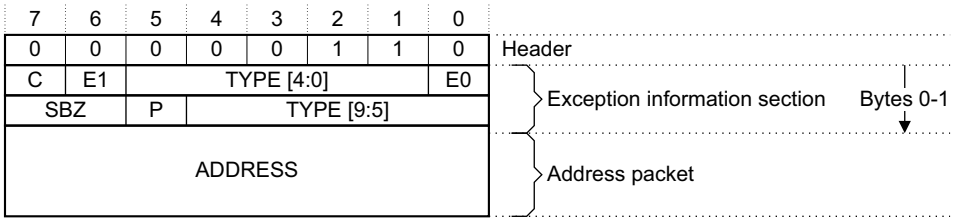


Figure 6-9 Exception instruction trace packet

The fields in an Exception packet are:

E0 When combined with the E1 field, the E0 field indicates how the Address field is interpreted. The description of the E1 field contains more information.

E1 When combined with the E0 field, the E1 field identifies the elements that are indicated by this packet. The E1 and E0 fields are combined to form a 2-bit field [E1:E0]. The valid encodings for this field are:

- `0b00` Reserved.
- `0b01` An ADDRESS field follows the Exception information bytes. The Address packet indicates the preferred exception return address of the exception.
- `0b10` An ADDRESS field follows the Exception information bytes. The Address packet indicates the preferred exception return address of the exception.

  Furthermore, an Address element and optional Context element are output before the *Exception element*. The address and context in the Address and Context elements are the values provided in the ADDRESS field of this packet, and are therefore the same as the preferred exception return address of the exception.
- `0b11` Reserved.

TYPE The exception type. The possible values for this field are shown in Table 6-12 for Armv7-A/R and Armv8-A and Armv8-R PEs, and are shown in Table 6-13 on page 6-264 for Armv6-M, Armv7-M, and Armv8-M PEs.

If any bits of the TYPE field are not output, their value is zero.

P This bit indicates if there is a serious fault pending, for Armv6-M, Armv7-M, and Armv8-M PEs.

`0b0` There is no serious fault pending.

`0b1` There is a serious fault pending.

This bit is always `0b0` for Armv8-A, Armv8-R, Armv7-A and Armv7-R PEs.

See *Additional information for tracing exceptions on Armv6-M, Armv7-M, and Armv8-M on page 5-200* and *Further information for tracing exceptions on Armv8-M on page 5-205* for more details on pending serious faults.

ADDRESS This field is the address for an *Exception element*, and it is always present in the packet. The E0 and E1 fields indicate how to interpret this field. The ADDRESS field takes the form of one of the Address packets, so that the field itself takes the form of a Short Address packet, a Long Address packet, an Exact Match Address packet, or an Address with Context packet, complete with header. See *Address and Context tracing packets on page 6-285* for descriptions of these packet types.

The trace unit always updates its stored `address_regs` and might update its stored context registers based on the contents of the ADDRESS field, in the same way as it would from a normal Address or Address with Context packet. See *Trace analyzer state between receiving instruction trace packets on page 6-239* for more information.

C The first payload byte in the packet is the first exception information byte, and this has a continuation bit, C. If C is set to 1, then a second exception information byte follows. Otherwise, if C is set to 0, there are no more exception information bytes in the packet. However, in either case, an ADDRESS field follows the exception information section.

Table 6-12 Possible values for the TYPE field in an Exception instruction trace packet, for Armv7-A/R and Armv8-A/R PEs

TYPE[4:0][a]	Exception type
`0b00000`	PE reset
`0b00001`	Debug halt
`0b00010`	Call
`0b00011`	Trap
`0b00100`	System error
`0b00101`	Reserved
`0b00110`	Inst debug
`0b00111`	Data debug
`0b01000`	Reserved
`0b01001`	Reserved
`0b01010`	Alignment
`0b01011`	Inst fault
`0b01100`	Data fault
`0b01101`	Reserved
`0b01110`	IRQ

Table 6-12 Possible values for the TYPE field in an Exception instruction trace packet, for Armv7-A/R and Armv8-A/R PEs (continued)

TYPE[4:0][a]	Exception type
0b01111	FIQ
0b10000 to 0b10111	Reserved for IMPLEMENTATION DEFINED exceptions
0b11000 to 0b11111	Reserved

a. TYPE[9:5] are always 0b00000 for Armv7-A/R and Armv8-A/R PEs.

Table 6-13 Possible values for the TYPE field in an Exception instruction trace packet, for Armv6-M, Armv7-M, and Armv8-M PEs

TYPE[9:0]	Exception
0b0000000000	Reserved
0b0000000001	PE reset
0b0000000010	NMI
0b0000000011	HardFault
0b0000000100	MemManage
0b0000000101	BusFault
0b0000000110	UsageFault
0b0000000111	SecureFault
0b0000001000	Reserved
0b0000001001	Reserved
0b0000001010	Reserved
0b0000001011	SVC
0b0000001100	Debug Monitor
0b0000001101	Reserved
0b0000001110	PendSV
0b0000001111	SysTick
0b0000010000	IRQ0
0b0000010001	IRQ1
0b0000010010	IRQ2
0b0000010011	IRQ3
0b0000010100	IRQ4
0b0000010101	IRQ5
0b0000010110	IRQ6
0b0000010111	IRQ7

Table 6-13 Possible values for the TYPE field in an Exception instruction trace packet, for Armv6-M, Armv7-M, and Armv8-M PEs (continued)

TYPE[9:0]	Exception
0b0000011000	Debug halt
0b0000011001	Lazy FP push
0b0000011010	Lockup
0b0000011011	Reserved
0b0000011100	Reserved
0b0000011101	Reserved
0b0000011110	Reserved
0b0000011111	Reserved
0b0000100000 to 0b0000101111	Reserved for IMPLEMENTATION DEFINED exceptions
0b0000110000 to 0b0111111111	Reserved
0b1000000000 to 0b1000000111	Reserved
0b1000001000 to 0b1111101111	IRQ8 to IRQ495
0b1111110000 to 0b1111111111	Reserved

Note

Some implementations might have IMPLEMENTATION DEFINED exceptions. Therefore, some of the encodings in Table 6-12 on page 6-263 and Table 6-13 on page 6-264 are reserved for this scenario. Arm does not intend to use these encodings in the future. However, Arm does reserve all the other encodings that are denoted as reserved in these tables, and therefore these must not be used by an implementation.

For all exceptions taken from a misaligned address, the bottom bits of the preferred exception return address are always traced as zero:

- For AArch64 A64 and AArch32 A32, address bits[1:0] are always traced as 0b00.
- For AArch32 T32, address bit[0] is always traced as 0.

When a PE Reset exception is traced, the ADDRESS field in the Exception packet is valid, and correctly updates the recently traced address registers in the same way as all other Address packets. See *Address and Context tracing packets on page 6-285* for more information. However, the address provided with the *Exception element* is UNKNOWN, as is the address and context provided with any Address or Context elements implied before the *Exception element*.

The ExceptionPacket() function is:

```
//ExceptionPacket()
//==============

ExceptionPacket()
    bits(2) EE = E1:E0;
    case of EE
        when '01'
            address_packet(false);
            handle_exception(UInt(TYPE), address_regs[0].address);
        when '10'
            // Handle Address field as an Address packet,
            // including emitting necessary Address or Context elements
            address_packet(true);
            handle_exception(UInt(TYPE), address_regs[0].address);
```

Handling Exception instruction trace packets

The ExceptionPacket() function calls an exception handler.

The `handle_exception(integer type, bits(64) address)` function is:

```
//handle_exception(integer type, bits(64) address)
//===============================================================

handle_exception(integer type, bits(64) address, bit pending)
    emit(exception_element(type, address, p0_key, pending));
    emit(conditional_flush_element());
    p0_key = (p0_key + 1) MOD p0_key_max; curr_spec_depth = curr_spec_depth + 1;
    if (curr_spec_depth > max_spec_depth) then
        emit(commit_element(1));
        curr_spec_depth = curr_spec_depth - 1;
    if ((type != IMPDEF_EXCEPTIONS) &&
        (type != Lazy FP push) &&
        (type != Lockup)) then
        emit(branch_future_flush_element());
```

Exception Return instruction trace packet

Note

For a description of the Exception Return trace element, see *Exception Return instruction trace element on page 5-208*.

The trace unit outputs an Exception Return packet whenever it generates an Exception Return element. An Exception Return element is generated whenever the PE executes an instruction that is classified as an exception return instruction.

For Armv6-M, Armv7-M, and Armv8-M PEs, if the Exception Return element exceeds the maximum P0 speculation depth, then the Exception Return packet also implies a Commit element.

Appendix F *Instruction Categories* shows the instructions that are classified as exception return instructions.

The Exception Return packet consists of only the exception return header, as shown in Figure 6-10.

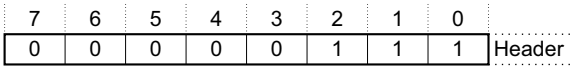


Figure 6-10 Exception Return instruction trace packet

The `ExceptionReturn()` function depends on the PE architecture.

For Armv7-A/R and Armv8-A/R PEs, the function is:

```
//ExceptionReturnPacket()
//=======================

ExceptionReturn()
    emit(exception_return_element());
```

For Armv6-M, Armv7-M, and Armv8-M PEs, the function is:

```
//ExceptionReturnPacket()
//=======================

ExceptionReturn()
    emit(exception_return_element(p0_key));
    p0_key = (p0_key + 1) MOD p0_key_max;
    curr_spec_depth = curr_spec_depth + 1;
    if (curr_spec_depth > max_spec_depth) then
        emit(commit_element(1));
```

```
        curr_spec_depth = curr_spec_depth - 1;

    emit(branch_future_flush_element());
```

6.4.6 Cycle Count packets

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. If cycle counting is supported, and it has been enabled by setting TRCCONFIGR.CCI to 1 and programming TRCCCCTLR, then the trace unit outputs Cycle Count packets that contain processor clock cycle count values.

Cycle Count packets are associated with Commit elements, so that when a Commit element is generated, a Cycle Count element might also be generated. See *Cycle Count instruction trace element on page 5-216*. 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. If the threshold value is not reached when a Commit element is generated, then a Cycle Count element is not generated. However, when a Commit element is generated, if 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, and the cycle count value that is contained in that packet is associated with the Commit element that triggered it.

A Cycle Count packet is therefore only output if:

- Cycle counting is supported and enabled.
- A Commit element is generated.
- At the time when the Commit element is generated, the cycle count value is greater than, or equal to, the threshold value that is programmed in TRCCCCTLR.THRESHOLD.

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.

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.

There are three formats for Cycle Count packets:

Cycle Count Format 1 instruction trace packet

Indicates a large cycle count value and zero or more Commit elements. If any Commit elements are indicated by the packet, then the cycle count value corresponds to the last Commit element indicated.

Cycle Count Format 2 instruction trace packet

Indicates a medium cycle count value and zero or more Commit elements. If any Commit elements are indicated by the packet, then the cycle count value corresponds to the last Commit element indicated.

Cycle Count Format 3 instruction trace packet

Indicates a small cycle count value and one or more Commit elements. If any Commit elements are indicated by the packet, then the cycle count value corresponds to the last Commit element indicated.

Cycle Count Format 1 instruction trace packet

This packet is output if the cycle count value is large and there are a zero or more Commit elements. A Cycle Count Format 1 packet consists of a header byte plus a variable number of payload bytes, as shown in Figure 6-11 on page 6-268. There is no upper limit on the number of payload bytes a Cycle Count Format 1 packet can have.

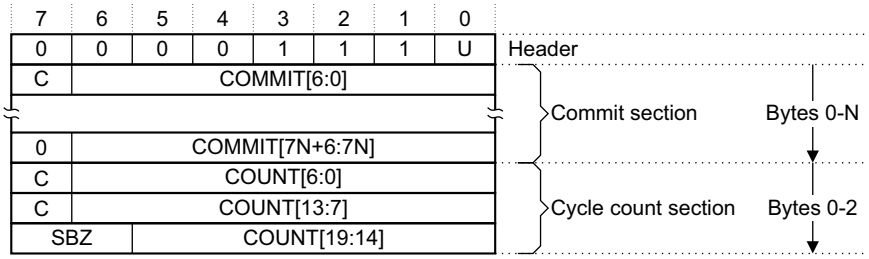


Figure 6-11 Cycle Count Format 1 instruction trace packet

The fields in a Cycle Count Format 1 packet are:

U This bit indicates if the cycle count value is UNKNOWN.

- **0** The cycle count is known.
- **1** The cycle count is UNKNOWN.

If the cycle count value is UNKNOWN, the cycle count field is not present in the packet.

COMMIT The value that is given in this field is the number of Commit elements that the Cycle Count Format 1 packet indicates.

This field is only present in the packet if TRCIDR0.COMMOPT==0. When TRCIDR0.COMMOPT==1, the Commit section is not present and zero Commit elements are indicated.

When this field is present, if any bits of it are not output, their value is zero.

A trace unit must not output more COMMIT bytes than are required to indicate the maximum speculation depth. For example, if `max_spec_depth` is 32, no more than one COMMIT byte must be output.

COUNT This field indicates the number of processor clock cycles that have occurred from the last Commit element that had a Cycle Count packet associated with it, to the last Commit element that is indicated in the present Cycle Count Format 1 packet.

The value given in this field is an increment on the cycle count threshold value, so that the actual cycle count value is calculated from `cc_threshold`+COUNT. This helps to reduce trace bandwidth but it means that to determine the actual cycle count value, the threshold value is required. The threshold value can be found from the Trace Info instruction trace packet.

A trace unit must not output more COUNT bytes than are required to indicate the maximum value of the cycle counter. For example, if the cycle counter is 12 bits, no more than two COUNT bytes must be output.

C This is a continuation bit. If a byte in a section has this bit set to 1, then another byte follows in the same section. If a byte in a section has this bit set to 0, then it is the last byte in the section.

Note

If a byte in the Commit section has C set to 0, and there is no cycle count payload section in the packet, the byte is the last byte in the packet.

The `CycleCountFormat1Packet()` function is:

```
//CycleCountFormat1Packet()
//========================

CycleCountFormat1Packet()
    If (COMMIT > 0)
        emit(commit_element(COMMIT));
    curr_spec_depth = curr_spec_depth - COMMIT;
    if (U) then
        emit(cycle_count_element(UNKNOWN));
    else
```

```
        emit(cycle_count_element(COUNT + cc_threshold));
```

Cycle Count Format 2 instruction trace packet

Note

For a description of the Cycle Count element, see *Cycle Count instruction trace element on page 5-216*.

This packet indicates a medium cycle count value and that zero or more Commit elements have occurred. A Cycle Count Format 2 packet consists of a header byte plus one payload byte, as shown in Figure 6-12.

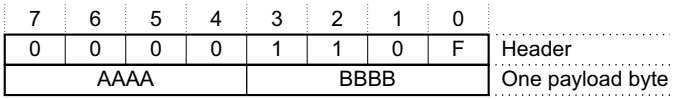


Figure 6-12 Cycle Count Format 2 instruction trace packet

The fields in the Cycle Count Format 2 packet are:

F This bit affects the meaning of the value shown in the AAAA field:

- **0** Read the AAAA field as AAAA+1.
- **1** Read the AAAA field as `max_spec_depth`+AAAA−15.

BBBB The field named BBBB indicates the cycle count value. The value given in this field is an increment on the cycle count threshold value, so that the actual cycle count value is calculated from `cc_threshold`+BBBB. The threshold value can be found from the Trace Info instruction trace packet.

AAAA The number of Commit elements that are indicated by this packet. The value of the F bit is required to interpret the meaning of this field.

The number of Commit elements must not be negative.

The `CycleCountFormat2Packet()` function is:

```
//CycleCountFormat2Packet()
//========================
CycleCountFormat2Packet()

    if (F) then
        commit_count = max_spec_depth + AAAA - 15;
    else
        commit_count = AAAA + 1;
    if (commit_count > 0)
        emit(commit_element(commit_count));
    curr_spec_depth = curr_spec_depth - commit_count;
    emit(cycle_count_element(cc_threshold + BBBB));
```

Cycle Count Format 3 instruction trace packet

Note

For a description of the Cycle Count element, see *Cycle Count instruction trace element on page 5-216*.

A Cycle Count Format 3 packet consists of only a header byte, and indicates a small cycle count value and that one or more Commit elements have occurred. Figure 6-13 shows the format of a Cycle Count Format 3 packet.

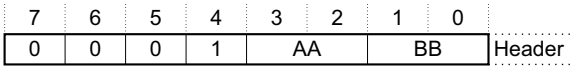


Figure 6-13 Cycle Count Format 3 instruction trace packet

The fields in the Cycle Count Format 3 packet are:

AA The number of Commit elements that are indicated by this packet. To interpret this field requires an addition of one, so that the number of Commit elements the packet indicates is AA+1.

This field is only used if TRCIDR0.COMMOPT==0. If TRCIDR0.COMMOPT==1 then zero Commit elements are indicated and this field is SBZ.

BB Indicates the cycle count value. The value that is given in this field is an increment on the cycle count threshold value, so that the actual cycle count value is calculated from `cc_threshold`+BB. The threshold value can be found from the Trace Info instruction trace packet.

The `CycleCountFormat3Packet()` function is:

```
//CycleCountFormat3Packet()
//=========================

CycleCountFormat3Packet()
    if (!TRCIDR0.COMMOPT) then
        emit(commit_element(AA + 1));
        curr_spec_depth = curr_spec_depth - (AA + 1);
    emit(cycle_count_element(cc_threshold + BB));
```

6.4.7 Data Synchronization Marker (Data Sync Mark) instruction trace packets

Data synchronization markers enable a trace analyzer to synchronize the data trace stream with the instruction trace stream. Therefore, the trace unit only generates Data Synchronization Marker elements, also known as Data Sync Mark elements, if data tracing is supported and enabled. When Data Sync Mark elements are output, they are output in both trace streams.

There are two forms of Data Sync Mark elements:

- Numbered Data Sync Mark elements. Each of these results in a Numbered Data Sync Mark packet that, as its name implies, contains a number that enables correlation between the two trace streams.
- Unnumbered Data Sync Mark trace elements. These result in Unnumbered Data Sync Mark packets. An Unnumbered Data Sync Mark packet occurs in the trace stream between two Numbered Data Sync Mark packets, which is true for both trace streams. Unnumbered Data Sync Mark packets enable a more accurate correlation of the two streams.

For more information, see *Synchronizing the instruction and data trace streams on page 2-43*.

In addition, see *Data Synchronization Marker (Data Sync Mark) instruction trace element on page 5-220*.

Numbered Data Synchronization Marker (Numbered Data Sync Mark) instruction trace packet

A numbered data synchronization marker provides an approximate correlation of the instruction trace stream with the data trace stream. The format of a Numbered Data Sync Mark packet is as shown in Figure 6-14.

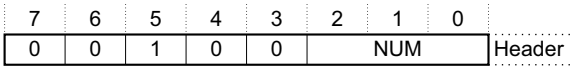


Figure 6-14 Numbered Data Sync Mark instruction trace packet

The NUM field contains the number of the Data Sync Mark element.

The `NumberedDataSynchronizationMarkerPacket()` function for the instruction trace stream is:

```
//NumberedDataSynchronizationMarkerPacket()
//=========================================

NumberedDataSynchronizationMarkerPacket()
    emit(numbered_sync_marker_element(NUM));
```

Unnumbered Data Synchronization Marker (Unnumbered Data Sync Mark) instruction trace packet

When used in conjunction with Numbered Data Synchronization Marker packets, this packet type provides an accurate correlation of the data trace stream with the instruction trace stream. For more information, see *Synchronizing the instruction and data trace streams on page 2-43*.

The presence of an Unnumbered Data Sync Mark packet indicates that zero to four *Atom elements* have occurred, followed by an Unnumbered Data Sync Mark element.

An Unnumbered Data Sync Mark packet consists of only a header byte, as shown in Figure 6-15.

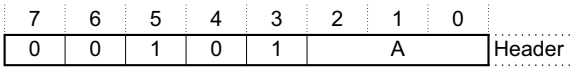


Figure 6-15 Unnumbered Data Sync Mark instruction trace packet

The value given in the A field is the number of *Atom elements* that occurred before the Unnumbered Data Sync Mark element was generated. In the A field, only values from `0b000` to `0b100` are permitted. All other values are not permitted because the header byte would then resemble header bytes of other packet types.

The `UnnumberedDataSynchronizationMarkerPacket()` function for the instruction trace stream is:

```
//UnnumberedDataSynchronizationMarkerPacket()
//===========================================

UnnumberedDataSynchronizationMarkerPacket()
    for I = 0 to UInt(A) - 1
        handle_atom(E);
    emit(sync_marker_element());
```

The handle_atom(atom type) function is defined in *Handling Atom instruction trace packets on page 6-299*.

6.4.8 Speculation resolution packets

The ETMv4 architecture supports the speculative execution of instructions by a PE.

An ETMv4 trace unit traces speculatively executed instructions in the same way as all other instructions, so that they all appear in the instruction trace stream.

Whenever the instruction trace stream shows any speculative execution, the trace unit generates elements to resolve the status of each speculatively executed instruction. These elements are:

- Commit elements. See *Commit instruction trace element on page 5-217*.
- Cancel elements. See *Cancel instruction trace element on page 5-218*.
- Mispredict elements. See *Mispredict instruction trace element on page 5-219*.

Speculation resolution packets are output when the trace generates these element types. There are five different types of speculation resolution packet. Each different packet type indicates a different mix, or different quantities, of these four element types. Table 6-14 provides a summary of the elements that are indicated by each of the speculation resolution packet types.

Table 6-14 Elements indicated by each type of speculation resolution packet

Packet name	Elements indicated
Commit	One or more Commit elements.
Cancel Format 1	One or more Cancel elements. This packet can also indicate the presence of a Mispredict element.
Cancel Format 2	The trace unit has generated 0-2 *Atom elements*, followed by one Cancel element and one Mispredict element.
Cancel Format 3	The trace unit has generated 0-1 *Atom elements*, followed by 2-5 Cancel elements.
Mispredict	The trace unit has generated zero or more *Atom elements*, followed by one Mispredict element.

The following sections describe these packet types:

- *Commit instruction trace packet*.
- *Cancel Format 1 instruction trace packet*.
- *Cancel Format 2 instruction trace packet on page 6-273*.
- *Cancel Format 3 instruction trace packet on page 6-274*.
- *Mispredict instruction trace packet on page 6-274*.

Note

Other packet types can also indicate some speculation resolution. For example, Cycle Count packets, Atom packets, and Exception packets, also indicate Commit elements.

Commit instruction trace packet

If a Commit packet is output, then the trace unit has generated one or more Commit elements. A Commit packet consists of a header byte plus a variable number of payload bytes, as shown in Figure 6-16. There is no upper limit on the number of payload bytes that a Commit packet can have.

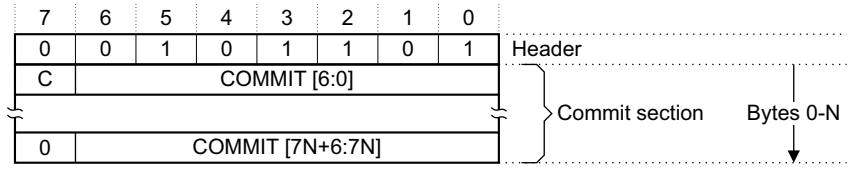


Figure 6-16 Commit instruction trace packet

The fields in a Commit packet are:

COMMIT The value that is given in this field is the number of Commit elements that this packet indicates. A value of zero Commit elements is not permitted.

If any bits of the COMMIT field are not output, their value is zero.

A trace unit must not output more COMMIT bytes than are required to indicate `max_spec_depth`. For example, if `max_spec_depth` is 32, no more than one COMMIT byte must be output.

C The continuation bit indicates if there is another COMMIT byte in the packet. If C is set to 1, then another COMMIT byte follows. Otherwise, if C is set to 0, no more COMMIT bytes follow.

The `CommitPacket()` function is:

```
//CommitPacket()
//==============

CommitPacket()
    emit(commit_element(COMMIT));
    curr_spec_depth = curr_spec_depth - COMMIT;
```

Cancel Format 1 instruction trace packet

If a Cancel Format 1 packet is output, then the trace unit has generated one or more Cancel elements. In addition, a Mispredict element might also have been generated after one or more Cancel elements. A Commit packet consists of a header byte plus a variable number of payload bytes, as shown in Figure 6-17 on page 6-273. There is no upper limit on the number of payload bytes that a Cancel Format 1 packet can have.

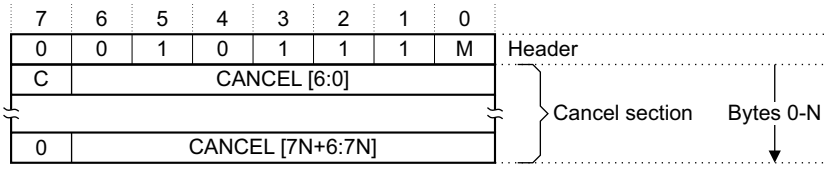


Figure 6-17 Cancel Format 1 instruction trace packet

The fields in a Cancel Format 1 packet are:

M This is the mispredict bit. It indicates whether a Mispredict element occurred after the Cancel elements:

- **0** No Mispredict element occurred.
- **1** A Mispredict element occurred after the Cancel elements.

CANCEL The value that is given in this field is the number of Cancel elements that this packet indicates. A value of zero Cancel elements is not permitted.

If any bits of the CANCEL field are not output, their value is zero.

A trace unit must not output more CANCEL bytes than are required to indicate `max_spec_depth`. For example, if `max_spec_depth` is 32, no more than one CANCEL byte must be output.

C The continuation bit indicates if there is another CANCEL byte in the packet. If C is set to 1, then another CANCEL byte follows. Otherwise, if C is set to 0, no more CANCEL bytes follow.

The `CancelFormat1Packet()` function cancels part of the speculation depth, rewinds some of the P0 keys, and might emit a Mispredict element.

```
// CancelFormat1Packet()
//======================

CancelFormat1Packet()
    emit(cancel_element(CANCEL));
    curr_spec_depth = curr_spec_depth - CANCEL;
    p0_key = (p0_key - CANCEL) MOD p0_key_max;
    if (M) then
        emit(mispredict_element());
        emit(conditional_flush_element());
```

Cancel Format 2 instruction trace packet

The trace unit outputs a Cancel Format 2 packet if it generates 0-2 *Atom elements* followed by one Cancel element and one Mispredict element. This is a single-byte packet, so it consists of only a header as shown in Figure 6-18.

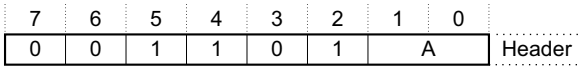


Figure 6-18 Cancel Format 2 instruction trace packet

The A field indicates the number of *Atom elements* that occurred before the Cancel element was generated. The possible values are:

`0b00` No *Atom elements* occurred.

`0b01` One E *Atom element* has occurred.

`0b10` Two E *Atom elements* have occurred.

`0b11` One N *Atom element* has occurred.

The `CancelFormat2Packet()` function is:

```
//CancelFormat2Packet()
//====================

CancelFormat2packet()
```

```
    case of A
        when '01'
            handle_atom(E);
        when '10'
            handle_atom(E);
            handle_atom(E);
        when '11'
            handle_atom(N);
    emit(cancel_element(1));
    p0_key = (p0_key - 1) MOD p0_key_max;
    curr_spec_depth = curr_spec_depth - 1;
    emit(mispredict_element());
    emit(conditional_flush_element());
```

The handle_atom(atom type) function is defined in *Handling Atom instruction trace packets on page 6-299*.

Cancel Format 3 instruction trace packet

The trace unit outputs a Cancel Format 3 packet if it generates zero or one E *Atom elements,* followed by 2-5 Cancel elements and one Mispredict element. This is a single-byte packet, so it consists of only a header as shown in Figure 6-19.

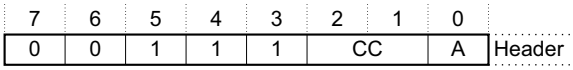


Figure 6-19 Cancel Format 3 instruction trace packet

The fields in a Cancel Format 3 packet are:

A This bit indicates if the packet signifies zero E *Atom elements*, or one E *Atom element*:

- **0** No E *Atom elements* occurred.
- **1** One E *Atom element* has occurred.

CC This field indicates the number of Cancel elements. The number of Cancel elements is CC+2.

The `CancelFormat3Packet()` function is:

```
//CancelFormat3Packet()
//====================

CancelFormat3Packet()
    if (A) then
        handle_atom(E);
    emit(cancel_element(CC + 2));
    p0_key = (p0_key - (CC + 2)) MOD p0_key_max;
    curr_spec_depth = curr_spec_depth - (CC + 2);
    emit(mispredict_element);
    emit(conditional_flush_element());
```

The handle_atom(atom type) function is defined in *Handling Atom instruction trace packets on page 6-299*.

Mispredict instruction trace packet

The trace unit outputs a Mispredict packet if it generates zero or more *Atom elements*, followed by one Mispredict element. This is a single-byte packet, as shown in Figure 6-20.

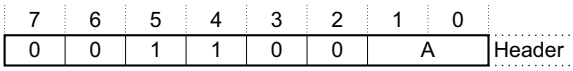


Figure 6-20 Mispredict instruction trace packet

The A field indicates the number of *Atom elements* that occurred before the Mispredict element was generated. The possible values are:

`0b00` No *Atom elements* occurred.

`0b01` One E *Atom element* has occurred.

`0b10` Two E *Atom elements* have occurred.

`0b11` One N *Atom element* has occurred.

The `MispredictPacket()` function is:

```
//MispredictPacket()
//==================

MispredictPacket()
    case A of
        when '01'
            handle_atom(E);
        when '10'
            handle_atom(E);
            handle_atom(E);
        when '11'
            handle_atom(N);
    emit(mispredict_element);
    emit(conditional_flush_element());
```

The handle_atom(atom type) function is defined in *Handling Atom instruction trace packets on page 6-299*.

6.4.9 Packets associated with tracing conditional instructions

Conditional instructions are traced in one of two ways, depending on whether they are branch instructions:

- All conditional branch instructions are traced using *Atom elements*, that have an E or N status. See *Atom instruction trace element on page 5-192*.
- If tracing of conditional non-branch instructions is supported and enabled, then conditional non-branch instructions are traced using Conditional Instruction (C) elements, Conditional Result (R) elements, and Conditional Flush (F) elements.

This section describes the packet types that are associated with tracing conditional non-branch instructions, that is, the packet types that C, R, and F elements are encoded into. For a description of Atom packets, see *Atom instruction trace packets on page 6-298*.

The purpose of each element type is as follows:

- A C element is generated when the PE executes a conditional non-branch instruction. See *Conditional Instruction (C) instruction trace element on page 5-220*.
- An R element is generated whenever the result of a conditional non-branch instruction is known. See *Conditional Result (R) instruction trace element on page 5-220*.
- An F element is generated when zero or more C elements are canceled because they do not have a corresponding R element. See *Conditional Flush (F) instruction trace element on page 5-220*.

For a description of how these elements relate to each other, and how they are arranged in the trace stream, see *Trace behavior on tracing conditional instructions on page 2-71*.

The packets associated with C, R, and F elements are:

- Conditional Instruction packets. The trace unit outputs these when it generates C elements. There are three types of Conditional Instruction packets:
  - Conditional Instruction Format 1
  - Conditional Instruction Format 2
  - Conditional Instruction Format 3.
- Conditional Result packets. A Conditional Result packet can indicate either one R element, or a combination of C and R elements, depending on the packet type. The packet types in this category are:
  - Conditional Result Format 1

  - Conditional Result Format 2
  - Conditional Result Format 3
  - Conditional Result Format 4.

- The Conditional Flush packet. The trace unit outputs this packet type when zero or more C elements are canceled. This can happen for example, if the PE cancels some speculative instructions because of mis-speculation. In this case, some C elements that have previously been generated might no longer be relevant. In addition, it might be impossible to resolve the status of these C elements, because the speculative execution might not have progressed that far.

Table 6-15 provides a summary of the header encodings for all of these packet types.

Table 6-15 Packet header encodings summary table for packets associated with tracing conditional instructions

Header encoding	Packet name
0b010000xx (not 0b01000011)	Conditional Instruction Format 2
0b01000011	Conditional Flush
0b010001xx (not 0b01000111)	Conditional Result Format 4
0b01000111	Reserved
0b01001xxx (not 0b01001x11)	Conditional Result Format 2
0b01001x11	Reserved
0b0101xxxx	Conditional Result Format 3
0b01100xxx	Reserved
0b011010xx	Conditional Result Format 1
0b01101100	Conditional Instruction Format 1
0b01101101	Conditional Instruction Format 3
0b0110111x	Conditional Result Format 1

Because C and R elements contain keys, the trace analyzer must have the values of these keys so that it can associate the R elements with the correct C elements. As the trace unit outputs the instruction trace stream, it uses compression techniques to reduce the trace bandwidth, and one of these techniques involves representing the value of a C or R element key as an increment on the previous C or R key value. This means that the trace analyzer must store the most recent key value that it has decoded for a C element, and the most recent key value that it has decoded for an R element, so that it can determine the values for the keys belonging to the next C and R elements that it receives.

The key system uses an IMPLEMENTATION DEFINED number of keys. These consist of normal keys and special keys:

- The number of normal keys is `cond_key_max_incr`.

  These are numbered from zero to `cond_key_max_incr`−1.

- The number of special keys is `num_cond_key_spc`. This value is defined by TRCIDR13.NUMCONDSPC.

  The special keys are numbered from `cond_key_max_incr` to (`cond_key_max_incr`+`num_cond_key_spc`−1).

Typically, the key belonging to the next element is an incremental offset from the last key in the normal key space. If the last key is at the last possible normal key space, `cond_key_incr`−1, and the key belonging to the next element is incremented by one, then the key numbering wraps and the key for the next element is at normal key space number zero.

Sometimes, the key cannot be output using an incremental value. In these cases the key must be explicitly traced, and the trace unit might use one of the special key spaces to do this.

In the pseudocode, the value for a C element right-hand key is stored in cond_c_key, and the value for an R element left-hand key is stored in cond_r_key.

Handling conditional keys

The is_cond_key_special(integer key) function, used to determine if a key is a special key or a normal key in Conditional Instruction Format 1 and Conditional Result Format 1 packets, is:

```
//is_cond_key_special(integer key)
//================================

is_cond_key_special(integer key)
    if (key >= cond_key_max_incr)
        return true;
    else
        return false;
```

Conditional instruction packets

Conditional instruction packets are output when the trace unit generates C elements. There are three types of packet in this category:

- Conditional Instruction Format 1. This indicates that the trace unit has generated one C element.
- Conditional Instruction Format 2. This indicates that the trace unit has generated 1-2 C elements.
- Conditional Instruction Format 3. This indicates that the trace unit has generated 1-64 C elements.

Conditional Instruction Format 1 instruction trace packet

This type of packet indicates that the trace unit has generated one C element. The packet consists of a single header byte, plus one or more payload bytes that contain the value of the right-hand key of the element. The packet format is shown in Figure 6-21.

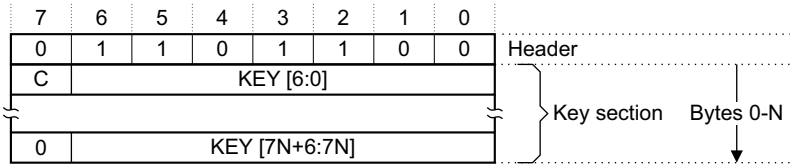


Figure 6-21 Conditional Instruction Format 1 instruction trace packet

The value given in the KEY field is the right-hand key value for the C element. Any bits that are not output have the value 0.

A trace unit must not output more KEY bytes than are required to indicate TRCIDR12.NUMCONDKEY. For example, if TRCIDR12.NUMCONDKEY is 32, no more than one KEY byte must be output.

The C bit is a continuation bit that indicates if there is another KEY byte in the packet. If C is set to 1, then another KEY byte follows. Otherwise, if C is set to 0, no more KEY bytes follow.

The ConditionalInstructionFormat1Packet() function is:

```
//ConditionalInstructionFormat1Packet()
//=====================================

ConditionalInstructionFormat1Packet()
    integer this_key = KEY;
    // If the key is special, we do not copy it to cond_c_key,
    // but we do still increment cond_c_key to skip over the value.
    // Otherwise we update cond_c_key with KEY.
    if (is_cond_key_special(KEY))
        cond_c_key = (cond_c_key + 1) MOD cond_key_max_incr;
    else
        cond_c_key = KEY;
    emit(conditional_instruction_element(this_key));
```

The is_cond_key_special(integer key) function is defined in *Handling conditional keys on page 6-277*.

Conditional Instruction Format 2 instruction trace packet

A Conditional Instruction Format 2 packet signifies either one or two C elements. This type of packet consists of a single header byte, as shown in Figure 6-22.

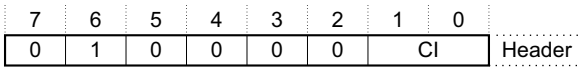


Figure 6-22 Conditional Instruction Format 2 instruction trace packet

The CI field indicates the number of C elements and what the value of the right-hand key is:

`0b00` A single C element has been generated. The element has a right-hand key value of `cond_c_key`+1.

`0b01` A single C element has been generated. The element has a right-hand key value of `cond_c_key`.

`0b10` Two C elements have been generated. The first element has a right-hand key value of `cond_c_key`+1. The value of the key for the second element is the same as the value of the key for the first element.

`0b11` This value is not permitted in a Conditional Instruction Format 2 packet. If the CI field has the value `0b11`, then the packet is not a Conditional Instruction Format 2 packet but is instead a Conditional Flush packet. See *Conditional Flush instruction trace packet on page 6-279* for more information.

The `ConditionalInstructionFormat2Packet()` function is:

```
//ConditionalInstructionFormat2Packet()
//=====================================

ConditionalInstructionFormat2Packet()
    case of CI
        when '00'
            cond_c_key = (cond_c_key + 1) MOD cond_key_max_incr;
            emit(conditional_instruction_element(cond_c_key));
        when '01'
            emit(conditional_instruction_element(cond_c_key))
        when '10'
            cond_c_key = (cond_c_key + 1) MOD cond_key_max_incr;
            emit(conditional_instruction_element(cond_c_key));
            emit(conditional_instruction_element(cond_c_key));
```

Conditional Instruction Format 3 instruction trace packet

A Conditional Instruction Format 3 packet indicates that 1-64 C elements have occurred, each with a right-hand key value that has been incremented by one from the key value associated with the previous C element.

In addition, this packet might also indicate the presence of a final C element, whose right-hand key value is identical to the key value provided with the most recent C element. The most recent C element could be the last one that is indicated by the Format 3 packet, or, if the Format 3 packet only indicates the final C element, then the most recent C element is the last one indicated by a previous packet type that indicates C elements.

A Conditional Instruction Format 3 packet consists of a single header byte plus a single payload byte, as shown in Figure 6-23.

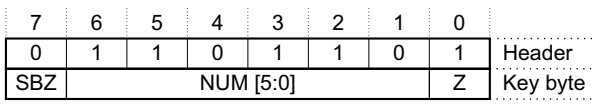


Figure 6-23 Conditional Instruction Format 3 instruction trace packet

The fields in a Conditional Instruction Format 3 packet are:

Z Indicates if a final C element is present:

0 There is no final C element.

1 One more C element, the final C element, is indicated in addition to those indicated in the NUM field. The right-hand key value of the final C element is identical to the key value for last C element indicated in the NUM field, or, if the NUM field does not signify any C elements, the key value is identical to the key value for the last C element indicated by a previous packet.

NUM The value given in this field is the number of C elements that the packet signifies. Each C element has a right-hand key that is incrementally one more than the right-hand key provided with the previous C element.

The payload byte that results from the NUM and Z fields both being set to zero is a reserved value, and must not be used.

The `ConditionalInstructionFormat3Packet()` function is:

```
//ConditionalInstructionFormat3Packet()
//=====================================

ConditionalInstructionFormat3Packet()
    for I = 1 to UInt(NUM)
        cond_c_key = (cond_c_key + 1) MOD cond_key_max_incr;
        emit(conditional_instruction_element(cond_c_key));
    if (Z) then
        emit(conditional_instruction_element(cond_c_key));
```

Conditional Flush instruction trace packet

A Conditional Flush packet is output whenever a Conditional Flush element is generated. A Conditional Flush element indicates that any C elements that have not been resolved by an R element must be discarded. The Conditional Flush packet consists of only a header byte, as shown in Figure 6-24.

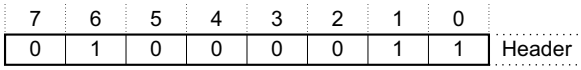


Figure 6-24 Conditional Flush instruction trace packet

The `ConditionalFlushPacket()` function is:

```
//ConditionalFlushPacket()
//========================

ConditionalFlushPacket()
    emit(conditional_flush_element());
```

Conditional Result packets

Conditional Result packets signify either:

- A number of C elements plus one R element.
- One R element.

Whether a Conditional Result packet signifies only one R element, or a number of C elements and one R element, depends on the packet type. There are four types of packet in this category:

- Conditional Result Format 1. This indicates either one or two groups of elements. Each group contains zero or more C elements plus one R element.
- Conditional Result Format 2. This indicates one or more C elements, plus one R element.
- Conditional Result Format 3. This indicates one or more tokens. Each token indicates one or more C elements plus one R element.
- Conditional Result Format 4. This indicates that the trace unit has generated one R element.

Each of the packet types indicates one R element. An R element contains:

- A left-hand key value.
- A result payload, that either indicates:
  - The status of the APSR condition flags.
  - Whether the instruction passed or failed the condition code check.

  Which of these the packet contains is IMPLEMENTATION DEFINED. TRCIDR0.CONDTYPE shows what it implemented. For a description of the algorithms associated with each of these, see *About the generation of Conditional Instruction (C) elements on page 2-73*.

Note

When tracing the APSR condition flags, an R element payload does not necessarily contain correct values for all four flags. Instead, the payload contains correct values for only those particular flags that are required to determine the results of the associated C elements, but for the other flags, that is, those flags whose status is not required, the values might be reported in the payload as either 0 or 1. See *The algorithm for tracing the APSR condition flag values on page 2-73*.

When tracing conditional pass or fail results, the result payload of an R element indicates whether the conditional instruction passed or failed the condition code check.

To reduce trace bandwidth, some types of Conditional Result packets make use of token fields, that contain tokens to represent either the APSR condition flag values or the pass or fail result. In addition, in all but the Conditional Instruction Format 1 packet type, changes to the C element right-hand keys and R element left-hand keys are traced as an increment on the previous key value rather than traced explicitly. This also helps to reduce trace bandwidth.

The tokens that make up token fields can be either two or four bits wide. Within token fields, the order of the tokens is from the oldest to the newest, with the oldest tokens located in the least significant bits of the field. This means that a trace analyzer must begin analyzing a token field from bit[0] to extract the tokens.

2-bit tokens can have values of either `0b00`, `0b01`, or `0b10`. When first analyzing the token field, if a value of `0b11` is encountered in the bottom two bits, then this signifies a 4-bit token. All 4-bit tokens have `0b11` as their bottom two bits. This means that the size of a token, and therefore the boundary to the next token, can be determined from the bottom two bits, because if the bottom two bits have the value `0b11`, the token is four bits wide. If the bottom two bits have any other value, that is, `0b00`, `0b01`, or `0b10`, then the token is two bits wide. A token value of `0b11` indicates that the token is extended to four bits wide.

One of the packet types, the Conditional Result Format 3 packet, contains a token field that is twelve bits wide. This field can contain a maximum of six tokens, if those tokens are all 2-bit tokens. If there are not enough tokens to fill this field, then the remaining space is filled with null tokens. In this case, if the field requires only two bits of padding, the value `0b11` is used. If four or more bits of padding are required, then multiple tokens of the value `0b11` are used. Null tokens might also appear anywhere in the 12-bit field.

Decoding of the token values is given in Table 6-16 if tracing the APSR condition flag values, and in Table 6-17 on page 6-281 if tracing the pass or fail result.

Note

When tracing the pass or fail result, all tokens are two bits wide. No 4-bit tokens are used when tracing the pass or fail result.

Table 6-16 Conditional result token description, when tracing the APSR condition flag values

Token index	Token encoding	APSR values indicated
2-bit tokens		
0	`0b00`	C flag set
1	`0b01`	N flag set

Table 6-16 Conditional result token description, when tracing the APSR condition flag values

Token index	Token encoding	APSR values indicated
2	0b10	Z and C flags set
-	0b11	Signifies a 4-bit token[a]
4-bit tokens		
3	0b0011	N and C flags set
4	0b0111	No flags set
5	0b1011	Z flag set
-	0b1111	Null, no R element indicated, used for padding

a. If 0b11 is encountered as the two most significant bits in the token field of a Conditional Result Format 3 packet, then 0b11 has been used as padding and does not signify a 4-bit token.

Table 6-17 Conditional result token description, when tracing the pass or fail result

Token index	Token encoding	Pass or fail value
0	0b00	Fail
1	0b01	Pass
-	0b11	Null, no R element indicated

Conditional Result Format 1 instruction trace packet

A Conditional Result Format 1 packet indicates that zero or more C elements have occurred, followed by a single R element.

This packet type consists of a single header byte plus one or more payload bytes. However, two different headers are used. Both headers identify the packet as a Conditional Result Format 1 packet, but while one header indicates that there is one set of payload bytes in the packet, the other header indicates that there are two sets of payload bytes in the packet, as shown in Figure 6-25 on page 6-282.

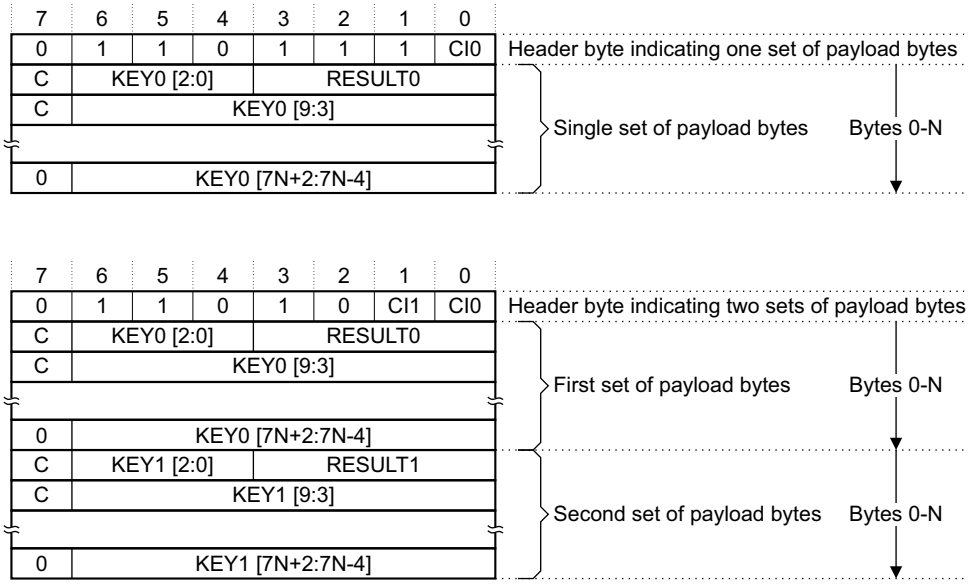


Figure 6-25 Conditional Result Format 1 instruction trace packet

The fields in a Conditional Result Format 1 packet are:

CIn Indicates whether this set of payload bytes signifies only a single R element, or one or more C elements in addition to the single R element:

- **0** This set of payload bytes indicates only a single R element.
- **1** This set of payload bytes indicates one or more C elements followed by a single R element.

If the value of the CIn bit is 1, KEYn must not be a special key.

RESULTn If tracing APSR condition code flag values, this field contains the flag values. The bits are allocated as follows:

- **[0]** This bit has the same value as the V flag in the APSR.
- **[1]** This bit has the same value as the C flag in the APSR.
- **[2]** This bit has the same value as the Z flag in the APSR.
- **[3]** This bit has the same value as the N flag in the APSR.

If tracing conditional pass or fail results, bits[3:1] of this field are SBZ, and bit[0] indicates whether the instruction passed or failed the condition code check, as follows:

- **0** The instruction failed the condition code check.
- **1** The instruction passed the condition code check.

KEYn The value given in this field is the value of the left-hand key of the R element. Any bits that are not output have the value zero. If the CIn bit is set to 1, KEYn must not be a special key.

A trace unit must not output more KEY bytes than are required to indicate TRCIDR12.NUMCONDKEY. For example, if TRCIDR12.NUMCONDKEY is 32, no more than one KEY byte must be output.

C The continuation bit indicates if there is another byte in the set of payload bytes. If C is set to 1, then another byte follows. Otherwise, if C is set to 0, no more bytes follow in that set.

The `ConditionalResultFormat1Packet()` function is:

```
//ConditionalResultFormat1Packet()
//================================

ConditionalResultFormat1Packet()
    for I = 0 to (Number of payloads present - 1)
```

```
this_r_key = KEY(I);
// If the key is special, we do not copy it to cond_r_key,
// but we do still increment cond_r_key to skip over the value.
// Otherwise we update cond_r_key with KEY(I).
if (is_cond_key_special(this_r_key))
    cond_r_key = (cond_r_key + 1) MOD cond_key_max_incr;
else
    cond_r_key = this_r_key;
result = RESULT(I);
if (CI(I)) then
    repeat
        cond_c_key = (cond_c_key + 1) MOD cond_key_max_incr;
        emit(conditional_instruction_element(cond_c_key));
    until (cond_c_key == cond_r_key);
emit(conditional_result_element(this_r_key,result));
```

The is_cond_key_special(integer key) function is defined in *Handling conditional keys on page 6-277*.

Conditional Result Format 2 instruction trace packet

A Conditional Result Format 2 packet indicates that one or more C elements have occurred, followed by a single R element. This packet type consists of only a header, see Figure 6-26.

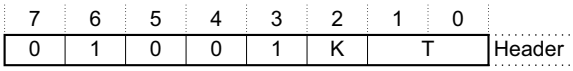


Figure 6-26 Conditional Result Format 2 instruction trace packet

The fields in a Conditional Result Format 2 packet are:

K This bit indicates what the key increment is:

- **0** cond_r_key is +1.
- **1** cond_r_key is +2.

T This is a 2-bit token. The permissible values for this token are `0b00`, `0b01`, and `0b10`. See Table 6-16 on page 6-280 or Table 6-17 on page 6-281 for the meaning of these values.

The `ConditionalResultFormat2Packet()` function is:

```
//ConditionalResultFormat2Packet()
//================================

ConditionalResultFormat2Packet()
    cond_r_key = (cond_r_key + 1 + K) MOD cond_key_max_incr;
    repeat
        cond_c_key = (cond_c_key + 1) MOD cond_key_max_incr;
        emit(conditional_instruction_element(cond_c_key));
    until (cond_c_key == cond_r_key);
    result = interpet T as a token;
    emit(conditional_result_element(cond_r_key,result));
```

Conditional Result Format 3 instruction trace packet

A Conditional Result Format 3 packet contains one or more tokens. Each token indicates one or more C elements followed by a single R element. This packet type consists of single header byte, plus one payload byte, as shown in Figure 6-27.

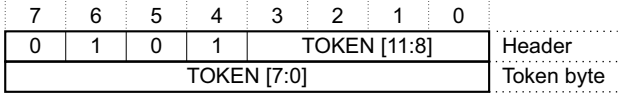


Figure 6-27 Conditional Result Format 3 instruction trace packet

The TOKEN field is a 12-bit field that can contain multiple tokens, that can consist of either 2-bit tokens, 4-bit tokens, or a combination of both. If the tokens do not fill the field, then the field is padded with null tokens. If the field requires only two bits of padding, the value `0b11` is used. If four or more bits of padding are required, then multiple tokens of the value `0b11` are used.

Null tokens might appear anywhere in the TOKEN field. Analysis of the TOKEN field then progresses to the next token.

The TOKEN field must be analyzed from bit[0] to extract the tokens.

The `ConditionalResultFormat3Packet()` function is:

```
//ConditionalResultFormat3Packet()
//================================

ConditionalResultFormat3Packet()
    tokens[] = analyze TOKEN[11:0] to extract a list of tokens
    for (each token in tokens[])
        if (token != NULL) then
            cond_r_key = (cond_r_key + 1) MOD cond_key_max_incr;
            repeat
                cond_c_key = (cond_c_key + 1) MOD cond_key_max_incr;
                emit(conditional_instruction_element(cond_c_key));
            until (cond_c_key == cond_r_key);
            result = interpret token;
            emit(conditional_result_element(cond_r_key,result));
```

Conditional Result Format 4 instruction trace packet

A Conditional Result Format 4 packet indicates that a single R element has occurred. The key for this element, `cond_r_key`, is one less that it was for the previous R element.

A Conditional Result Format 4 packet is a single-byte packet, so consists of only a header as shown in Figure 6-28.

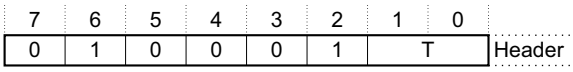


Figure 6-28 Conditional Result Format 4 instruction trace packet

The T field in a Conditional Result Format 4 packet is a token field. Only 2-bit token values of `0b00`, `0b01`, or `0b10` are permitted. See Table 6-16 on page 6-280 or Table 6-17 on page 6-281 for the meaning of these values.

The `ConditionalResultFormat4Packet()` function is:

```
//ConditionalResultFormat4Packet()
//================================

ConditionalResultFormat4Packet()
    cond_r_key = (cond_r_key - 1) MOD cond_key_max_incr;
    result = interpet T as a token;
    emit(conditional_result_element(cond_r_key,result));
```

6.4.10 Ignore packet

An Ignore packet is a single-byte packet that is ignored by a trace analyzer. The header value `0x70` is allocated for the Ignore packet in the instruction trace protocol, as shown in Figure 6-29. This header value is Reserved in ETMv4.2 or earlier.

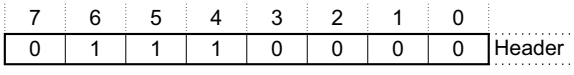


Figure 6-29 Ignore instruction trace packet

Ignore packets might be used to pad the trace stream to a convenient boundary. To avoid inefficient use of trace capture bandwidth or storage, Arm recommends that Ignore packets are not used frequently.

6.4.11 Event tracing instruction trace packet

An Event packet indicates that 1-4 Event elements have been generated. See *Event instruction trace element on page 5-217*. This is a single-byte packet, as shown in Figure 6-30.

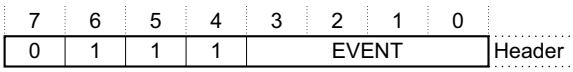


Figure 6-30 Event instruction trace packet

The EVENT field contains one bit per event, so that up to four trace unit events can be indicated as present. The possible values for each bit are:

0 This event did not occur.

1 This event occurred.

A value of `0b0000` for the EVENT field is not permitted.

The `EventTracingPacket()` function for the instruction trace stream is:

```
//EventTracingPacket()
//====================

EventTracingPacket()
    for I = 0 to 3
        if (EVENT<I>) then
            emit(event_element(I));
```

6.4.12 Address and Context tracing packets

Address and Context packets indicate, respectively, that Address and Context elements have occurred. Address elements contain the instruction address and an indication of the instruction set for the next instruction that is executed by the PE. Context elements contain information about the context in which instructions are being executed. The context includes, for example, the Security state of the PE, or if the PE complies with the Armv8 architecture, whether the PE is in the AArch32 or AArch64 Execution state.

See:

- *Address instruction trace element on page 5-209*.
- *Context instruction trace element on page 5-211*.

Types of Address and Context packet

There are three types of Address packet:

- The Short Address packet. There are five different formats for this packet type.
- The Long Address packet. There are six different formats for this packet type.
- An Exact Match Address packet. There is one format for this packet type.

There is only one type of Context packet, named Context packet, and in addition, there is an Address with Context packet. This last packet type indicates that both an Address and a Context element have occurred, and there are six different formats for this packet type. If the trace unit outputs either a Context packet, or an Address with Context packet, then it is an indication that some of the context might have changed.

Contents of Address and Context packets

Short Address and Long Address packets contain:

- The address of the next instruction executed.
- The instruction set for the next instruction executed.

Exact Match Address packets contain:

- An indication that the address and instruction set for the next instruction are the same as an address and instruction set provided in a previous packet.

Context packets contain:

- The Exception level. See *Context ID tracing on page 2-81*.
- Excepting Armv8-M PEs, the Security state of the PE. For Armv8-M PEs, the trace unit does not distinguish between the Secure and Non-secure states, and treats both states as Secure state.
- Whether the PE is in AArch32 or AArch64 state.
- The Context ID value. For Arm PEs, this is the value of the current *Context ID Register* (CONTEXTIDR).
- The *Virtual context identifier* value. See *Virtual context identifier tracing on page 2-81*, and Table 6-18.

Address with Context packets contain:

- The address of the next instruction executed.
- The instruction set for the next instruction executed.
- The Exception level.
- Excepting Armv8-M PEs, the Security state of the PE. For Armv8-M PEs, the trace unit does not distinguish between the Secure and Non-secure states, and treats both states as Secure state.
- Whether the PE is in AArch32 or AArch64 state.
- The Context ID value. See *Context ID tracing on page 2-81*.
- The Virtual context identifier value. See *Virtual context identifier tracing on page 2-81*, and Table 6-18.

Table 6-18 The Size of the Virtual context identifier field in trace packets

Value of V field	TRCIDR2.VMIDSIZE	Size of Virtual context identifier field
0	N/A	Virtual context identifier field is not present.
N/A	`0b00000`	Virtual context identifier field is never present. The V field must be zero in this scenario.
1	`0b00001`	Always 1 byte.
1	`0b00010`	Always 2 bytes.
1	`0b00100`	Always 4 bytes.
1	Other	UNKNOWN

Compression techniques used when generating Address packets

The trace unit stores up to three recent address values, plus an indication of the instruction set for each of those address values, in a queue. The queue entries are `address_regs[0]`, `address_regs[1]`, and `address_regs[2]`. Each time a new Address packet is output, the address and instruction set indicated by that packet are stored at the top of the queue in entry `address_regs[0]`, and the contents of the other entries are pushed downwards, so that the values that were stored in `address_regs[0]` are moved into `address_regs[1]`, the values that were stored in `address_regs[1]` are moved into `address_regs[2]`, and the values that were stored in `address_regs[2]` are discarded.

Before generating an Address packet, the trace unit compares the new address value with each of the three stored addresses, and if it exactly matches any one of them, an Exact Match Address packet is output instead of an Address packet. Information is given in the Exact Match Address packet that indicates which of the three stored addresses is an exact match. In this way, the amount of trace generated is minimized, because an Exact Match Address packet is much smaller than either a Short or Long Address packet type.

Note

There is no requirement for an implementation to implement all three address stores.

Another technique that is used to minimize the amount of trace that is generated involves only including in Address packets the bits that have changed since the most recently traced address. That is, Address packets only contain the appropriate number of least significant bits that are required to signify any changes from the value that is stored in `address_regs[0]`.

In summary, the trace unit operates as follows:

1. The three most recent address values, together with an indication of their instruction sets, are stored in a queue, where the most recent is stored in `address_regs[0]` and the oldest is stored in `address_regs[2]`. As each new Address packet is generated, the address value and instruction set are stored at the top of the queue, in `entry address_regs[0]`, and the contents of the other entries are pushed downwards until the contents that were in the third entry, `address_regs[2]`, are discarded.
2. Before generating a new Address packet, the trace unit compares the new address with each of the values that are stored in the queue. If an exact match is detected, an Exact Match Address packet is output.
3. If, when comparing the address and instruction set values, an exact match does not exist, then the trace unit outputs either a Short Address packet, or a Long Address packet. That Address packet contains only the number of least significant bits that are required to indicate any bits that differ from those stored at the top queue entry, in `address_regs[0]`.

A trace analyzer must maintain a copy of the queue, `address_regs[0]`, `address_regs[1]`, and `address_regs[2]`, so that it can decode the address and instruction set. See *Trace analyzer state between receiving packets on page 6-239*.

Note

The contents of the three stored queue entries are reset whenever a Trace Info packet is output. When the queue entries are reset, all three of the addresses are set to zero, and all three instruction set indicators are set to IS0.

Decoding the instruction set from an Address packet

An Address packet indicates to a trace analyzer that the trace unit has generated an Address instruction trace element. An Address instruction trace element contains the instruction address, and an indication of the instruction set, for the next instruction that is executed by the PE.

In the Address packet, the header byte indicates what the instruction set is. However, to fully decode the instruction set from an Address packet header, the 64-bit state of the PE is also required. This is indicated by the sixty_four_bit state in the trace analyzer. The value of the sixty_four_bit state is updated by Context packets, Address with Context packets, and Trace Info packets. If required, a Context packet that changes the value of the sixty_four_bit state might be output before or after an Address packet, and the instruction set can only be determined when both the Address and Context packets have been received. See *Context instruction trace packet on page 6-292*.

Table 6-19 shows how to decode the instruction set from the state of the SF bit and the header byte of an Address packet.

Table 6-19 Instruction set encodings

SF bit value[a]	Instruction set[b]	Alignment	Equivalent in Armv7-A/R	Equivalent in Armv7-M	Equivalent in Armv8
0	IS0	Word-aligned	Arm	-	A32
0	IS1	Halfword-aligned	Thumb	Thumb	T32
1	IS0	Word-aligned	-	-	A64

a. The state of the SF bit is given in the most recent Context packet.
b. This information is obtained from the header byte of the Address packet.

Note

All other SF and IS encodings are Reserved,

Not all the instruction set encodings shown in Table 6-19 on page 6-287 can indicate the least significant bits of the address. For example:

- The instruction set named IS0 can only support word-aligned instruction sets. If an Address packet signifies IS0, bits[1:0] of the address always have the value `0b00`.
- IS1 can only support halfword-aligned, and word-aligned, instruction sets. If an Address packet signifies IS1 then bit[0] of the address is always 0.

For branches to misaligned program addresses, the bottom bits of the address are always traced as zero:

- For AArch64 A64 and AArch32 A32 the address bits[1:0] are always traced as `0b00`.
- For AArch32 T32, address bit[0] is always traced as 0.

Summary of Address and Context packet header encodings

Table 6-20 provides a summary of the header encodings for all of the types of Address and Context tracing packets.

Table 6-20 Packet header encodings summary table for Address and Context tracing packets

Header encoding	Packet name	Purpose
`0b1001000x` `0b10010010`	Exact Match Address	Indicates when the new address exactly matches one of those stored in the address queue
`0b10010011`		Reserved
`0b10010100`		Reserved
`0b10010101` `0b10010110`	Short Address	Indicates that up to 17 bits of the address value have changed from that stored at the top queue entry, in address_regs[0]
`0b10010111` `0b1001100x`		Reserved

`0b1001101x`
`0b10011101`
`0b10011110`
	Long Address	Indicates that up to 64 bits of the address value have changed from that stored at the top queue entry, in address_regs[0]

`0b10011100` `0b10011111`	-	Reserved
`0b1000000x`	Context	Indicates a Context element

`0b1000001x`
`0b10000101`
`0b10000110`
	Address with Context	Indicates that up to 64 bits of the address value have changed from that stored in address_regs[0], plus a Context element

`0b10000100` `0b10000111`	-	Reserved

Whenever the trace analyzer processes an Address or the ADDRESS field of an Exception packet, it uses the `AddressPacket()` function defined in the following pseudocode. When an Address packet is processed, the emit parameter is always set to TRUE. When an Exception packet is processed, the emit parameter is set to TRUE in the case of an Exception packet when the E1:E0 field of the packet is set to `0b10`, and to FALSE in all other cases.

The `AddressPacket()` function is:

```
//AddressPacket(boolean emit)
//===========================

AddressPacket(boolean emit)
    case header of
        when '10010000' ExactMatchAddressPacket(emit);
        when '10010001' ExactMatchAddressPacket(emit);
```

```
        when '10010010' ExactMatchAddressPacket(emit);
        when '10010101' ShortAddressPacket(emit);
        when '10010110' ShortAddressPacket(emit);
        when '10011010' LongAddressPacket(emit);
        when '10011011' LongAddressPacket(emit);
        when '10011101' LongAddressPacket(emit);
        when '10011110' LongAddressPacket(emit);
        when '10000000' ContextPacket(emit);
        when '10000001' ContextPacket(emit);
        when '10000010' AddressWithContextPacket(emit);
        when '10000011' AddressWithContextPacket(emit);
        when '10000101' AddressWithContextPacket(emit);
        when '10000110' AddressWithContextPacket(emit);
```

Short Address instruction trace packets

A Short Address packet contains the address of the next instruction to be executed, and provides an indication of the instruction set used when executing that instruction. There are two different formats of Short Address packet, each identified by a different header as shown in Figure 6-31. The packets consist of either a header byte plus one payload byte, or a header byte plus two payload bytes. It is the header byte that identifies which instruction set the packet indicates.

A Short Address packet type can indicate up to 17 bits of the address, depending on the instruction set.

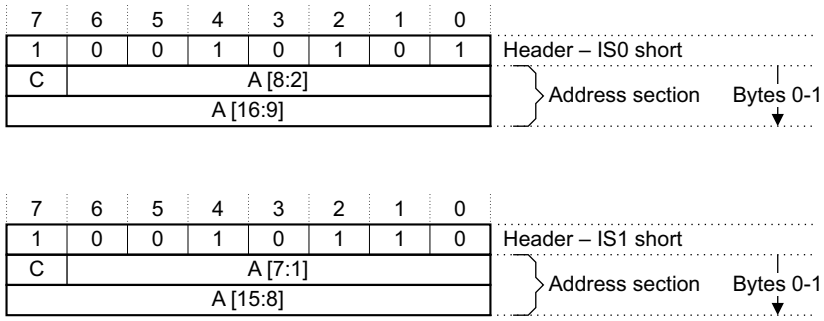


Figure 6-31 Short Address packets

The fields in Short Address packets are:

A The value in this field is the address of the next instruction executed by the PE.

If the instruction set is IS0, bits[1:0] always have the value `0b00`.

If the instruction set is IS1, bit[0] is 0.

If any address bits are not output, their value is the same as shown in the top entry on the address queue, that is, their value is the same as in `address_regs[0]`.

C The continuation bit indicates if there is another A byte in the packet. If C is set to 1, then another A byte follows. Otherwise, if C is set to 0, no more A bytes follow.

The `ShortAddressPacket()` function is:

```
//ShortAddressPacket()
//====================

ShortAddressPacket()
    bits(64) address = address_regs[0];
    bits(2) IS;
    case header of
        when '10010101'
            IS = 0;
            address = address & ~0x1FF;
            address = address | (A<8:2> << 2);
            if (C) then
```

```
            address = address & ~0x1FE00;
            address = address | (A<16:9> << 9);
        when '10010110'
            IS = 1;
            address = address & ~0xFF;
            address = address | (A<7:1> << 1);
            if (C) then
                address = address & ~0xFF00;
                address = address | (A<15:8> << 8);
    if (emit) then
        emit(address_element(address,IS));
    update_address_regs(address,IS);
```

The `update_address_regs()` function is defined in *Additional Address packets pseudocode on page 6-298*.

Long Address instruction trace packets

There are four different formats of Long Address packet. Two of these can indicate up to 32 bits of the address for the next instruction to be executed, and the other two can indicate up to 64 bits of the address. There are always four payload bytes in the packets that can indicate up to 32 bits, and there are always eight payload bytes in the packets that can indicate up to 64 bits. The header bytes identify the packet format and instruction set. The different packet formats are shown in Figure 6-32 and Figure 6-33 on page 6-291.

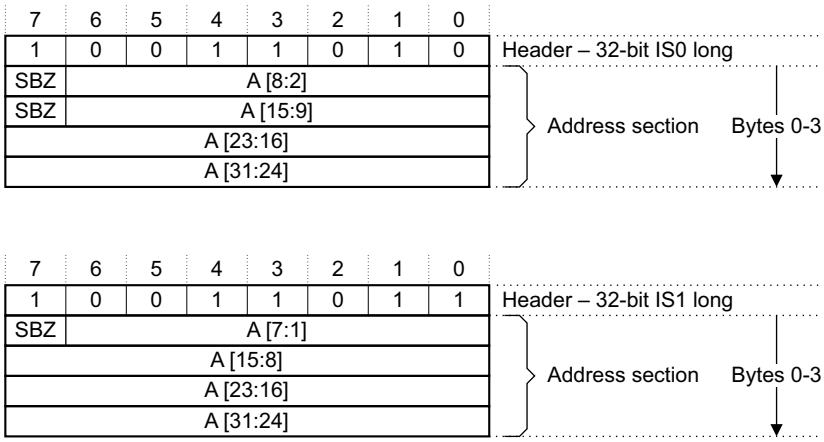


Figure 6-32 Long Address packets that can indicate up to 32 bits of the instruction address

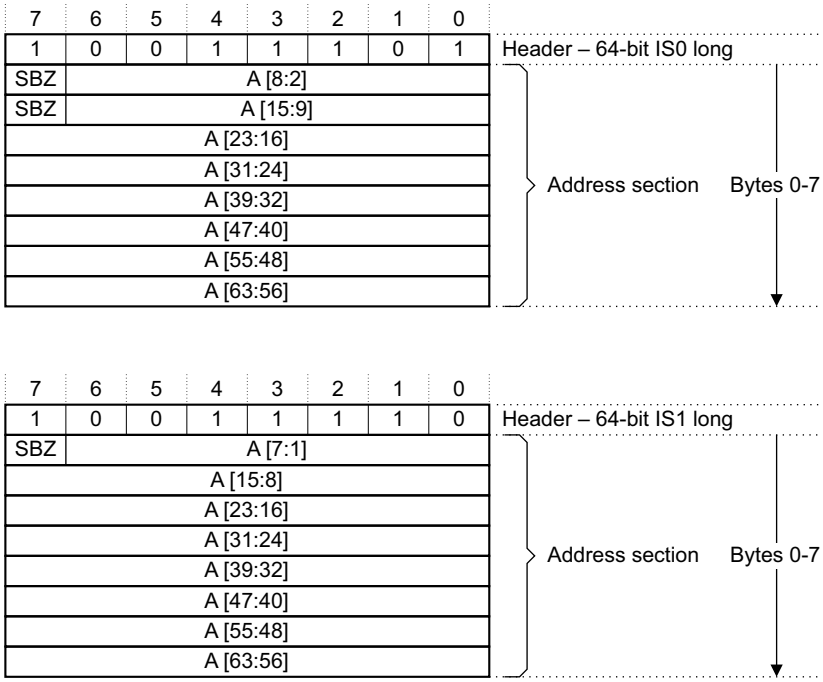


Figure 6-33 Long Address packets that can indicate up to 64 bits of the instruction address

The fields in Long Address packets are:

A The value in this field is the address of the next instruction that is executed by the PE. For those packets shown in Figure 6-32 on page 6-290, up to 32 bits of the address can be provided, depending on the instruction set. For those packets shown in Figure 6-33, up to 64 bits of the address can be provided, depending on the instruction set.

If the instruction set is IS0, bits[1:0] always have the value `0b00`.

If the instruction set is IS1, bit[0] is 0.

If any address bits are not output, their value is the same as shown in the top entry on the address queue, that is, their value is the same as in `address_regs[0]`.

The `LongAddressPacket()` function is:

```
//LongAddressPacket(boolean emit)
//===============================

LongAddressPacket(boolean emit)
    bits(64) address = address_regs[0];
    bits(2) IS;
    case header of
        when '10011010'
            IS = 0;
            address = address & ~0xFFFFFFFF;
            address = address | (A<8:2> << 2) | (A<15:9> << 9) |
                                (A<23:16> << 16) | (A<31:24> << 24);
        when '10011011'
            IS = 1;
            address = address & ~0xFFFFFFFF;
            address = address | (A<7:1> << 1) | (A<15:8> << 8) |
                                (A<23:16> << 16) | (A<31:24> << 24);
        when '10011101'
            IS = 0;
            address = 0x0;
            address = address (A<8:2> << 2) | (A<15:9> << 9) |
                              (A<23:16> << 16) | (A<31:24> << 24) |
                              (A<39:32> << 32) | (A<47:40> << 40) |
```

```
                        (A<55:48> << 48) | (A<63:56> << 56);
        when '10011110'
            IS = 1;
            address = 0x0;
            address = address (A<7:1> << 1) | (A<15:8> << 8) |
                        (A<23:16> << 16) | (A<31:24> << 24) |
                        (A<39:32> << 32) | (A<47:40> << 40) |
                        (A<55:48> << 48) | (A<63:56> << 56);
    if (emit) then
        emit(address_element(address,IS));
    update_address_regs(address,IS);
```

The update_address_regs() function is defined in *Additional Address packets pseudocode on page 6-298*.

Exact Match Address instruction trace packet

Unlike a Short Address or Long Address packet, an Exact Match Address packet does not contain an address field or indicate an instruction set. If the trace unit outputs an Exact Match Address packet, then the address and instruction set for the next instruction to be executed exactly matches one of the entries in the address queue. That is, the address and instruction set are exactly the same as that stored in either address_regs[0], address_regs[1], or address_regs[2].

An Exact Match Address packet is a single-byte packet, so consists of only a header as shown in Figure 6-34.

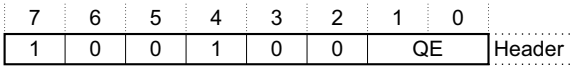


Figure 6-34 Exact Match Address instruction trace packet

The QE field indicates the queue entry that contains the exact match. The possible values are:

0b00 The address and instruction set stored in address_regs[0] is an exact match.

0b01 The address and instruction set stored in address_regs[1] is an exact match.

0b10 The address and instruction set stored in address_regs[2] is an exact match.

0b11 Reserved.

The ExactMatchPacket() function is:

```
//ExactMatchPacket(boolean emit)
====================

ExactMatchPacket(boolean emit)
    if(emit) then
        emit(address_element(address_regs[QE].address,address_regs[QE].IS));
    update_address_regs(address_regs[QE].address,address_regs[QE].IS);
```

The update_address_regs() function is defined in *Additional Address packets pseudocode on page 6-298*.

Context instruction trace packet

A Context packet indicates that the trace unit has generated a Context element. A Context element contains information about the context that instructions are being executed in. This might include:

- The Context ID value.
- The Virtual context identifier value.
- The Security state.
- The Exception level.
- Whether the PE is in AArch32 state or AArch64 state.

If the trace unit outputs a Context packet, then it is an indication that some or all of the context information might have changed. See *Context instruction trace element on page 5-211*.

A Context packet consists of a single header byte, plus 0-9 payload bytes.

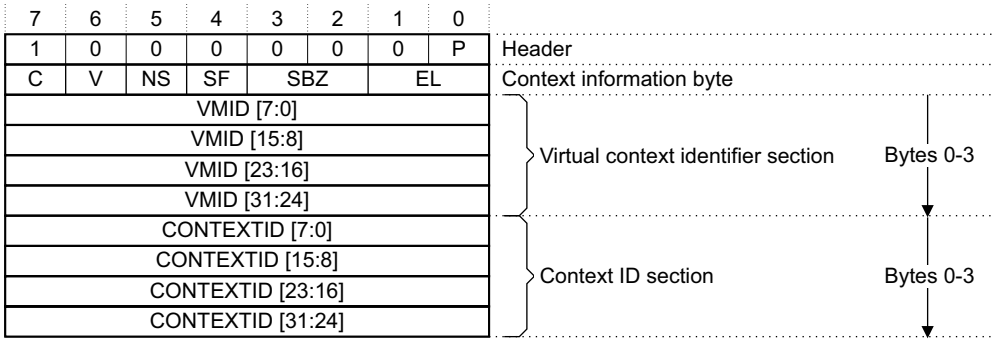


Figure 6-35 Context instruction trace packet

The fields in the Context packet are:

P This bit indicates if the packet has a payload or not:

0 There is no payload, so the packet consists of only a header byte. In this case, the context is the same as the most recently traced context.

1 A payload is present in the packet. The payload consists of at least an information byte. The information byte contains more information about the payload.

EL Indicates the Exception level. The possible values are:

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

For more information about the Exception levels, see *Context instruction trace element on page 5-211*.

SF Indicates whether the PE is in AArch32 state or AArch64 state:

0 The PE is in AArch32 state.

1 The PE is in AArch64 state.

NS Indicates the PE Security state:

0 The PE is in Secure state.

1 The PE is in Non-secure state.

For Armv8-M PEs, the current Security state is not presented in the instruction trace stream. This field is always traced as zero, indicating Secure state.

V Indicates whether the Virtual context identifier section is present in the packet.

0 Virtual context identifier bytes are not present.

1 Virtual context identifier bytes are present.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is `0b0`.
- The VMID field must contain a value of zero.

If TRCIDR2.VMIDSIZE is not one of the supported values, the size of the Virtual context identifier field is UNKNOWN.

C Indicates whether the Context ID section is present in the packet:

0 The Context ID bytes are not present.

1 The Context ID bytes are present.

If Context ID tracing is disabled, then one of the following must occur:

- This field is `0b0`.

- The CONTEXTID field must contain a value of zero.

If TRCIDR2.CIDSIZE is not one of the supported values, the size of the CONTEXTID field is UNKNOWN.

VMID This field contains the Virtual context identifier value. See Table 6-18 on page 6-286.

CONTEXTID This field contains the Context ID value. For Arm PEs, this is the value of the current CONTEXTIDR.

The `ContextPacket()` function is:

```
//ContextPacket(boolean emit)
//===============

ContextPacket(boolean emit)
    if (P) then
        if (C) then
            context_id<31:0> = CONTEXTID<31:0>;
        if (V) then
             if (TRCIDR2.VMIDSIZE == b00001) then
                 vmid<7:0> = VMID<7:0>;
             elsif (TRCIDR2.VMIDSIZE == b00010) then
                 vmid<15:0> = VMID<15:0>;
             elsif (TRCIDR2.VMIDSIZE == b00100) then
                 vmid<31:0> = VMID<31:0>;
        ex_level<1:0> = EL<1:0>;
        security = if NS then NONSECURE else SECURE;
        sixty_four_bit = SF;
    if (emit) then
        emit(context_element(context_id,vmid,ex_level,security,sixty_four_bit));
```

Address with Context instruction trace packets

If, for the next instruction that is executed by the PE, the address cannot be inferred and the context has changed, then an Address with Context packet might be output. An Address with Context packet therefore indicates that the trace unit has generated a Context element plus an Address element.

There are four different formats of Address with Context packet. Two of these formats, which are shown in Figure 6-36 on page 6-295, can indicate up to 32 bits of the address for the next instruction to be executed, and the other two formats, which are shown in Figure 6-37 on page 6-296, can indicate up to 64 bits of the address.

All formats of Address with Context packets have a context section, that contains the following bytes:

- A context information byte. This byte is always present.
- VMID bytes. These bytes might be present.
- Context ID bytes. These bytes might be present.

The header byte of each packet identifies the packet format and instruction set.

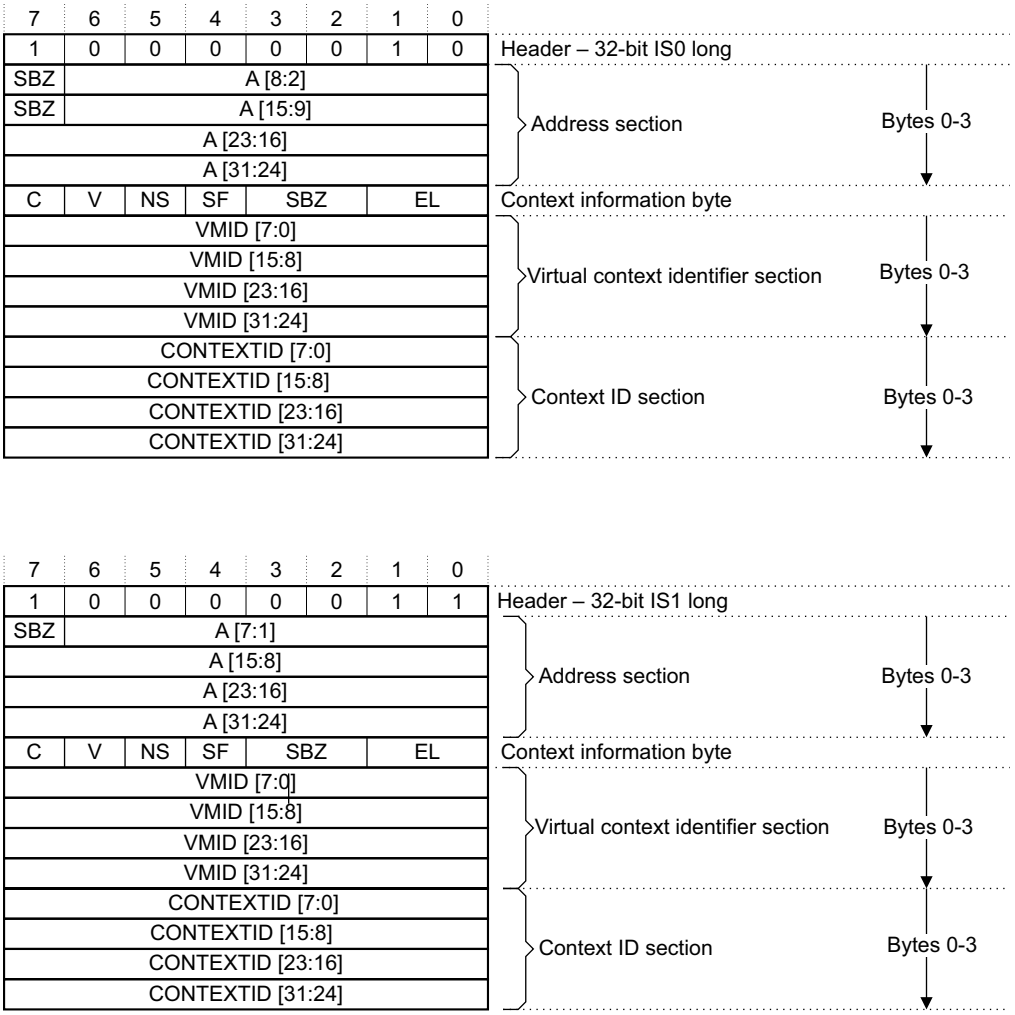


Figure 6-36 Address with Context Instruction trace packets that can indicate up to 32 bits of the address

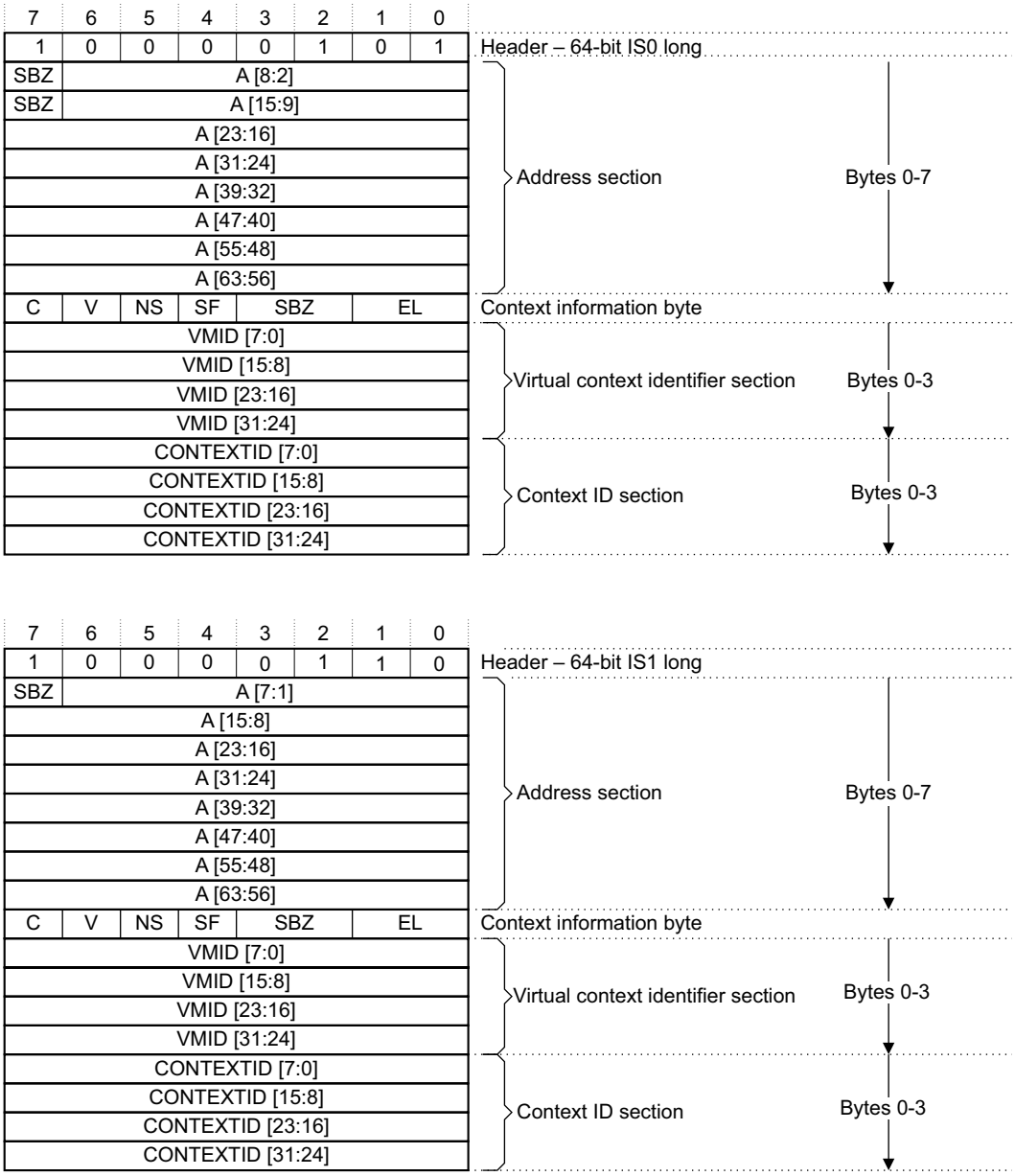


Figure 6-37 Address with Context Instruction trace packets that can indicate up to 64 bits of the address

The fields in Address with Context packets are the same as those described in *Long Address instruction trace packets on page 6-290*, and those described in *Context instruction trace packet on page 6-292*. They are repeated here for clarity:

A The value in this field is the address of the next instruction that is executed by the PE. For those packets shown in Figure 6-36 on page 6-295, up to 32 bits of the address can be provided, depending on the instruction set. For those packets shown in Figure 6-37, up to 64 bits of the address can be provided, depending on the instruction set.

If the instruction set is IS0, bits[1:0] always have the value `0b00`.

If the instruction set is IS1, bit[0] is 0.

If any address bits are not output, their value is the same as shown in the top entry on the address queue, that is, their value is the same as in `address_regs[0]`.

EL Indicates the Exception level. The possible values are:

`0b00` EL0.

`0b01` EL1.

`0b10` EL2.

`0b11` EL3.

For more information about Exception levels, see *Context instruction trace element on page 5-211*.

SF Indicates whether the PE is in AArch32 state or AArch64 state:

0 The PE is in AArch32 state.

1 The PE is in AArch64 state.

NS Indicates the PE Security state:

0 The PE is in Secure state.

1 The PE is in Non-secure state.

For Armv8-M PEs, the current Security state is not presented in the instruction trace stream. This field is always traced as zero, indicating Secure state.

V Indicates whether the Virtual context identifier section is present in the packet:

0 The Virtual context identifier section is not present.

1 The Virtual context identifier section is present.

If Virtual context identifier tracing is disabled, then one of the following must occur:

- This field is `0b0`.
- The VMID field must contain a value of zero.

C Indicates whether the Context ID section is present in the packet:

0 The Context ID bytes are not present.

1 The Context ID bytes are present.

If Context ID tracing is disabled, then one of the following must occur:

- This field is `0b0`.
- The CONTEXTID field must contain a value of zero.

VMID This field contains the Virtual context identifier value. See *Virtual context identifier tracing on page 2-81*.

CONTEXTID This field contains the Context ID value. See *Context ID tracing on page 2-81*.

The `AddressWithContextPacket()` function is:

```
//AddressWithContextPacket(boolean emit)
//======================================

AddressWithContextPacket(boolean emit)
    if (C) then
        context_id<31:0> = CONTEXTID<31:0>;
    if (V) then
         if (TRCIDR2.VMIDSIZE == b00001) then
             vmid<7:0> = VMID<7:0>;
         elsif (TRCIDR2.VMIDSIZE == b00010) then
             vmid<15:0> = VMID<15:0>;
         elsif (TRCIDR2.VMIDSIZE == b00100) then
             vmid<31:0> = VMID<31:0>;
    ex_level<1:0> = EL<1:0>;
    security = if NS then NONSECURE else SECURE;
    sixty_four_bit = SF;
    if (emit) then
        emit(context_element(context_id,vmid,ex_level,security,sixty_four_bit));
    bits(64) address = address_regs[0];
    bits(2) IS;
    case header of
```

```
        when '10000010'
            IS = 00;
            address = address & ~0xFFFFFFFF;
            address = address | (A<8:2> << 2) | (A<15:9> << 9) |
                                (A<23:16> << 16) | (A<31:24> << 24);
        when '10000011'
            IS = 01;
            address = address & ~0xFFFFFFFF;
            address = address | (A<7:1> << 1) | (A<15:8> << 8) |
                                (A<23:16> << 16) | (A<31:24> << 24);
        when '10000101'
            IS = 0;
            address = 0x0;
            address = address | (A<8:2> << 2) | (A<15:9> << 9) |
                                (A<23:16> << 16) | (A<31:24> << 24) |
                                (A<39:32> << 32) | (A<47:40> << 40) |
                                (A<55:48> << 48) | (A<63:56> << 56);
        when '10000110'
            IS = 1;
            address = 0x0;
            address = address | (A<7:1> << 1) | (A<15:8> << 8) |
                                (A<23:16> << 16) | (A<31:24> << 24) |
                                (A<39:32> << 32) | (A<47:40> << 40) |
                                (A<55:48> << 48) | (A<63:56> << 56);
    if (emit) then
        emit(address_element(address,IS));
    update_address_regs(address,IS);
```

The `update_address_regs()` function is defined in *Additional Address packets pseudocode*.

Additional Address packets pseudocode

The `update_address_regs()` function is:

```
//update_address_regs(bits(64) address, bits(2) IS)
//================================================

update_address_regs(bits(64) address, bits(2) IS)
    address_regs[2] = address_regs[1];
    address_regs[1] = address_regs[0];
    address_regs[0].address = address;
    address_regs[0].IS = IS;
```

6.4.13 Atom instruction trace packets

Atom packets indicate that *Atom elements* have occurred. See *Atom instruction trace element on page 5-192*. Instruction types that result in *Atom elements* are:

- All direct branch instructions.
- All indirect branch instructions.
- Instruction Synchronization Barrier.
- Load instructions, if data tracing is supported and the tracing of data loads is enabled.
- Store instructions, if data tracing is supported and the tracing of data stores is enabled.

This means that every time the PE executes one of these types of instruction, the trace unit generates an *Atom element*. See Appendix F *Instruction Categories*for information on the instruction classifications for ETMv4.

Atom elements contain either an E or an N status, and:

- With regard to all direct and indirect branch instructions:

  E Atom Indicates that the instruction executed was a taken branch. In this case, execution has continued to the target of that branch.

  N Atom Indicates that the instruction executed was a not-taken branch.

- With regard to load or store instructions, when explicit tracing of data transfer instructions is enabled:

  E Atom Only E *Atom elements* are generated. Execution has continued to the next instruction after the load or store.

- With regard to `ISB` instructions:

  E Atom Indicates that the `ISB` performed an Instruction Synchronization Barrier operation.

  N Atom Indicates that the `ISB` did not perform an Instruction Synchronization Barrier operation.

- With regard to conditional instructions:

  — For conditional branches:

    E Atom Indicates that the instruction executed was a taken branch. In this case, execution has continued to the target of that branch.

    N Atom Indicates that the instruction executed was a not-taken branch.

  — For all other conditional instructions, no *Atom elements* are generated. These instructions are traced with Conditional Instruction elements, and the relevant Conditional Result element gives the result. See *Conditional instruction packets on page 6-277* and *Conditional Result packets on page 6-279*.

    Also see *Trace behavior on tracing conditional instructions on page 2-71* for an explanation of how conditional non-branch instructions are traced.

There are six different formats of Atom packet. Each packet format consists of only a header. Table 6-21 provides a summary of the six Atom packets.

If an *Atom element* means that the maximum P0 speculation depth is exceeded, then the Atom packet also implies Commit elements to restore the speculation depth to the maximum.

Note

Although the purpose of Atom packets is to indicate *Atom elements*, other types of packet can also indicate *Atom elements*. These are:

- *Unnumbered Data Synchronization Marker (Unnumbered Data Sync Mark) instruction trace packet on page 6-271*.
- *Cancel Format 2 instruction trace packet on page 6-273*.
- *Cancel Format 3 instruction trace packet on page 6-274*.
- *Mispredict instruction trace packet on page 6-274*.

Table 6-21 Packet header encodings summary table for Atom packets

Packet name	Header encoding	Description
Atom Format 1	`0b1111011x`	Indicates a single E or N *Atom element*
Atom Format 2	`0b110110xx`	Indicates a combination of 2 *Atom elements*
Atom Format 3	`0b11111xxx`	Indicates a combination of 3 *Atom elements*
Atom Format 4	`0b110111xx`	Indicates a combination of 4 *Atom elements*
Atom Format 5	`0b11110101` and `0b11010101-0b11010111`	Indicates a combination of 5 *Atom elements*
Atom Format 6	`0b11000000-0b11010100` and `0b11100000-0b11110100`	Indicates 4-24 *Atom elements*

Handling Atom instruction trace packets

The handle_atom(atom type) function is:

```
//handle_atom()
//======================
```

```
handle_atom(atom_type)
    emit(atom_element(atom_type, p0_key));
    p0_key = (p0_key + 1) MOD p0_key_max;
    curr_spec_depth = curr_spec_depth + 1;
    if (curr_spec_depth > max_spec_depth) then
        emit(commit_element(1));
        curr_spec_depth = curr_spec_depth - 1;
```

Atom Format 1 instruction trace packet

An Atom Format 1 packet indicates a single *Atom element*. The packet format is shown in Figure 6-38.

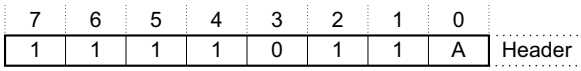


Figure 6-38 Atom Format 1 instruction trace packet

The A bit in the packet identifies whether the *Atom element* carries an E status or an N status:

0 The *Atom element* carries an N status.

1 The *Atom element* carries an E status.

The `AtomFormat1Packet()` function is:

```
//AtomFormat1Packet()
//==================

AtomFormat1Packet()
    if (A) then
        handle_atom(E);
    else
        handle_atom(N);
```

Atom Format 2 instruction trace packet

An Atom Format 2 packet indicates two *Atom elements*. The packet format is shown in Figure 6-39.

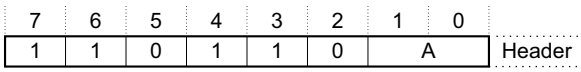


Figure 6-39 Atom Format 2 instruction trace packet

Each bit contained in the A field represents one *Atom element*. The least significant bit represents the oldest *Atom element*. The values of these bits indicate whether the *Atom elements* have an E status or an N status. The possible values for each bit are:

0 The *Atom element* has an N status.

1 The *Atom element* has an E status.

The `AtomFormat2Packet()` function is:

```
//AtomFormat2Packet()
//==================

AtomFormat2Packet()
    for I = 0 to 1
        if (A<I>) then
            handle_atom(E);
        else
            handle_atom(N);
```

Atom Format 3 instruction trace packet

An Atom Format 3 packet indicates three *Atom elements*. The packet format is shown in Figure 6-40 on page 6-301.

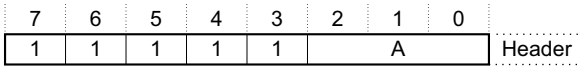


Figure 6-40 Atom Format 3 instruction trace packet

Each bit contained in the A field represents one *Atom element*. The *Atom elements* are ordered by age, with the least significant bit representing the oldest *Atom element*.

The values of the bits indicate whether the *Atom elements* contain an E status or an N status. The possible values for each bit are:

0 The *Atom element* contains an N status.

1 The *Atom element* contains an E status.

The `AtomFormat3Packet()` function is:

```
//AtomFormat3Packet()
//==================

AtomFormat3Packet()
    for I = 0 to 2
        if (A<I>) then
            handle_atom(E);
        else
            handle_atom(N);
```

Atom Format 4 instruction trace packet

An Atom Format 4 packet indicates four *Atom elements*. The packet format is shown in Figure 6-41.

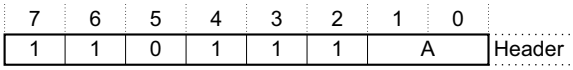


Figure 6-41 Atom Format 4 instruction trace packet

In this packet, the *Atom elements* are represented in a different way to how they are represented in an Atom Format 1 packet, an Atom Format 2 packet, or an Atom Format 3 packet. Instead of each bit in the A field representing one *Atom element,* an Atom Format 4 packet uses the values of the two bits in the A field to indicate a sequence of *Atom elements*, as follows:

`0b00` The sequence is:
1. N *Atom element*.
2. E *Atom element*.
3. E *Atom element*.
4. E *Atom element*.

`0b01` The sequence is:
1. N *Atom element*.
2. N *Atom element*.
3. N *Atom element*.
4. N *Atom element*.

`0b10` The sequence is:
1. N *Atom element*.
2. E *Atom element*.
3. N *Atom element*.
4. E *Atom element*.

`0b11` The sequence is:
1. E *Atom element*.
2. N *Atom element*.
3. E *Atom element*.
4. N *Atom element*.

The AtomFormat4Packet() function is:

```
//AtomFormat4Packet()
//==================

AtomFormat4Packet()
    case A of
        when '00'
            handle_atom(N);
            handle_atom(E);
            handle_atom(E);
            handle_atom(E);
        when '01'
            handle_atom(N);
            handle_atom(N);
            handle_atom(N);
            handle_atom(N);
        when '10'
            handle_atom(N);
            handle_atom(E);
            handle_atom(N);
            handle_atom(E);
        when '11'
            handle_atom(E);
            handle_atom(N);
            handle_atom(E);
            handle_atom(N);
```

Atom Format 5 instruction trace packet

An Atom Format 5 packet indicates five *Atom elements*. The packet format is shown in Figure 6-42.

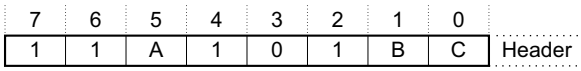


Figure 6-42 Atom Format 5 instruction trace packet

The A, B, and C bits, when put together as ABC, indicate a sequence of *Atom elements*. The possible values for ABC are:

`0b101` The sequence is:
1. N *Atom element*.
2. E *Atom element*.
3. E *Atom element*.
4. E *Atom element*.
5. E *Atom element*.

`0b001` The sequence is:
1. N *Atom element*.
2. N *Atom element*.
3. N *Atom element*.
4. N *Atom element*.
5. N *Atom element*.

`0b010` The sequence is:
1. N *Atom element*.
2. E *Atom element*.
3. N *Atom element*.
4. E *Atom element*.
5. N *Atom element*.

`0b011` The sequence is:
1. E *Atom element*.

2. N *Atom element*.
3. E *Atom element*.
4. N *Atom element*.
5. E *Atom element*.

Only the values `0b101`, `0b001`, `0b010`, and `0b011` are permitted for ABC. All other values are not permitted.

The `AtomFormat5Packet()` function is:

```
//AtomFormat5Packet()
//==================

AtomFormat5Packet()
    case ABC of
        when '101'
            handle_atom(N);
            handle_atom(E);
            handle_atom(E);
            handle_atom(E);
            handle_atom(E);
        when '001'
            handle_atom(N);
            handle_atom(N);
            handle_atom(N);
            handle_atom(N);
            handle_atom(N);
        when '010'
            handle_atom(N);
            handle_atom(E);
            handle_atom(N);
            handle_atom(E);
            handle_atom(N);
        when '011'
            handle_atom(E);
            handle_atom(N);
            handle_atom(E);
            handle_atom(N);
            handle_atom(E);
```

Atom Format 6 instruction trace packet

An Atom Format 6 packet indicates that a series of E *Atom elements* have been generated, followed by one final *Atom element* whose status might be either E or N. This packet can indicate between four and 24 *Atom elements*.

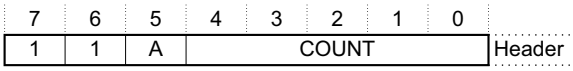


Figure 6-43 Atom Format 6 instruction trace packet

The fields in the Atom Format 6 packet are as follows:

COUNT The value contained in this field is the number of E *Atom elements* that were generated before the final *Atom element* was generated. The number of initial E Atoms is COUNT + 3. Only values from `0b00000` to `0b10100` are permitted.

A Indicates whether the final *Atom element* carries an E or an N status. The possible values for this bit are:

- **0** The *Atom element* carries an E status.
- **1** The *Atom element* carries an N status.

The `AtomFormat6Packet()` function is:

```
//AtomFormat6Packet()
//==================
```

```
AtomFormat6Packet()
    for I = 0 to UInt(COUNT) + 2
        handle_atom(E);
    if (A) then
        handle_atom(N);
    else
        handle_atom(E);
```

6.4.14 Q instruction trace packet

Q packets indicate that Q elements have occurred. See *Q element on page 5-195*.

A Q packet consists of a header byte that begins `0b1010`, followed by a 4-bit TYPE field, and then a pair of optional payload sections. The TYPE field indicates whether or not each of the payload sections is present. The payload sections are:

Address This section might be present, and takes the format indicated by the TYPE field.

COUNT This section might be present.

Figure 6-44 shows the format of a Q packet in the instruction trace stream.

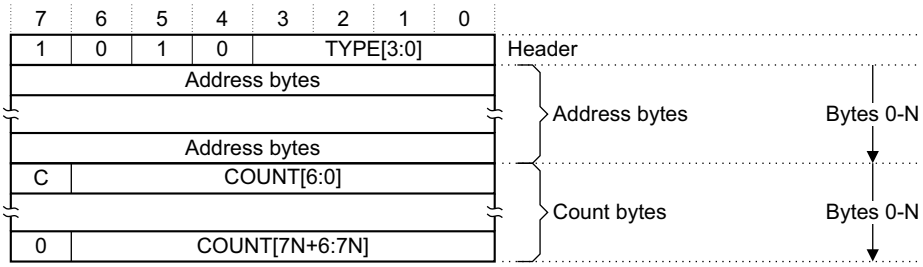


Figure 6-44 Q instruction trace packet

The fields in the Q packet are as follows:

TYPE The TYPE field indicates what form the rest of the packet takes. The valid values for this field are:

`0b0000` The Address section is not present. The COUNT section is present. A packet with this TYPE value also implies an Exact Match Address packet with QE set to `0b00`.

`0b0001` The Address section is not present. The COUNT section is present. A packet with this TYPE value also implies an Exact Match Address packet with QE set to `0b01`.

`0b0010` The Address section is not present. The COUNT section is present. A packet with this TYPE value also implies an Exact Match Address packet with QE set to `0b10`.

`0b0011` Reserved.

`0b0100` Reserved.

`0b0101` The Address section is present. The COUNT section is present. The Address section uses the same payload as Short IS0 Address packets.

`0b0110` The Address section is present. The COUNT section is present. The Address section uses the same payload as Short IS1 Address packets.

`0b0111` Reserved.

`0b1000` Reserved.

`0b1001` Reserved.

`0b1010` The Address section is present. The COUNT section is present. The Address section uses the same payload as 32-bit Long IS0 Address packets.

`0b1011` The Address section is present. The COUNT section is present. The Address section uses the same payload as 32-bit Long IS1 Address packets.

`0b1100` The Address section is not present. The COUNT section is present. No Address element is implied.

`0b1101` Reserved.

`0b1110` Reserved.

`0b1111` The Address section is not present. The COUNT section is not present. The COUNT is UNKNOWN. No Address element is implied.

Address If this packet implies a Short or Long Address element then its contents are placed in the Address section.

COUNT The COUNT section provides a count of instructions that are implied by the Q element. This section must not contain a value of 0.

Each byte of the COUNT section includes a C field in bit[7]. The C field indicates if there is another byte, as follows:

0 This is the last byte of the section.

1 There is at least one more byte.

The Q packet does not support an Address section with a 64-bit address. To output a Q packet with an Address that is 64 bits wide, two packets are required. The first packet is a Q packet without an Address section. The second packet is a 64-bit long Address packet.

The QPacket() function is:

```
//QPacket()
//=========

QPacket()
    case TYPE of
        when '0000'
            handle_q_element(COUNT);
            emit(address_element(address_regs[0].address,address_regs[0].IS));
            update_address_regs(address_regs[0].address,address_regs[0].IS);
        when '0001'
            handle_q_element(COUNT);
            emit(address_element(address_regs[1].address,address_regs[1].IS));
            update_address_regs(address_regs[1].address,address_regs[1].IS);
        when '0010'
            handle_q_element(COUNT);
            emit(address_element(address_regs[2].address,address_regs[2].IS));
            update_address_regs(address_regs[2].address,address_regs[2].IS);
        when '0101'
            handle_q_element(COUNT);
            bits(64) address = address_regs[0];
            address = address & ~0x1FF;
            address = address | (A<8:2> << 2);
            if (C field in Address bytes) then
                address = address & ~0x1FE00;
                address = address | (A<16:9> << 9);
            emit(address_element(address,0));
            update_address_regs(address,0);
        when '0110'
            handle_q_element(COUNT);
            bits(64) address = address_regs[0];
            address = address & ~0xFF;
            address = address | (A<7:1> << 1);
            if (C field in Address bytes) then
                address = address & ~0xFF00;
                address = address | (A<15:8> << 8);
            emit(address_element(address,1));
            update_address_regs(address,1);
        when '1010'
            handle_q_element(COUNT);
            bits(64) address = address_regs[0];
            address = address & ~0xFFFFFFFF;
            address = address | (A<8:2> << 2) | (A<15:9> << 9) |
                                (A<23:16> << 16) | (A<31:24> << 24);
            emit(address_element(address,0));
```

```
        update_address_regs(address,0);
    when '1011'
        handle_q_element(COUNT);
        bits(64) address = address_regs[0];
        address = address & ~0xFFFFFFFF;
        address = address | (A<7:1> << 1) | (A<15:8> << 8) |
                            (A<23:16> << 16) | (A<31:24> << 24);
        emit(address_element(address,1));
        update_address_regs(address,1);
    when '1100'
        handle_q_element(COUNT);
    when '1111'
        handle_q_element(UNKNOWN);
```

Handling Q elements

The handle_q_element(UInt count) function is:

```
//handle_q_element()
//==================

handle_q_element()
    emit(q_element(count,p0_key));
    p0_key = (p0_key + 1) MOD p0_key_max;
    curr_spec_depth = curr_spec_depth + 1;
    if(curr_spec_depth > max_spec_depth) then
        emit(commit_element(1));
        curr_spec_depth = curr_spec_depth - 1;
```

6.4.15 Branch Future Flush packet

The Branch Future Flush packet indicates a *Branch Future Flush Element*. See *Branch Future Flush element on page 5-213*.

A Branch Future Flush packet consists of a header byte plus one payload byte, as shown in Figure 6-45.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Extension header
0	0	0	0	0	1	1	1	Identifies the packet type as a Branch Future Flush packet

Figure 6-45 Branch Future Flush packet

The `BranchFutureFlush()` function for the instruction trace stream is:

```
// BranchFutureFlushPacket()
// =========================

BranchFutureFlushPacket()
    emit(branch_future_flush_element());
```

6.5 Descriptions of data trace packets

The following sections describe the packets that comprise the data trace stream:

6.5.1 Extension packets in the data trace stream

In Table 6-9 on page 6-249, four of the packets that are in the synchronization category and one of the Data Synchronization Marker packets are extension packets. In general, a packet type can be identified from its unique header byte. However, in the case of an extension packet, the header byte defines the packet as an extension packet, and it is the first payload byte that identifies the packet type. The header byte of an extension packet, regardless of what type of packet it is, always has the value `0b00000000`, as shown in Table 6-22.

Table 6-22 Extension packets

Packet name	Header byte	First payload byte	Purpose
-	`0b00000000`	`0bxxxxxxx0`	Reserved, except `0b00000000`
A-Sync		`0b00000000`	Identifies a packet boundary
Trace Info		`0b00000001`	Provides a point in the data trace stream where analysis of the trace stream can begin
Discard		`0b00000011`	Indicates that tracing has become inactive
Overflow		`0b00000101`	Indicates a trace unit buffer overflow
-		`0b00000111`	Reserved
-		`0b00001xx1`	Reserved
Numbered Data Sync Mark		`0b0001xxx1`	Enables approximate correlation of the data trace stream with the instruction trace stream
-		`0b001xxxx1`	Reserved
-		`0b01xxxxx1`	Reserved
-		`0b1xxxxxx1`	Reserved

A trace analyzer therefore requires the first two bytes of an extension packet to identify the packet type, whereas for all other packet types, a trace analyzer requires only the header byte to identify the packet type.

6.5.2 Packets associated with synchronization between the trace unit and a trace analyzer

The packet types that comprise this category are as follows:

- *Alignment Synchronization (A-Sync) data trace packet on page 6-309*.
- *Trace Info data trace packet on page 6-309*.
- *Discard data trace packet on page 6-310*.
- *Overflow data trace packet on page 6-311*.
- *Suppression data trace packet on page 6-311*.

Alignment Synchronization (A-Sync) data trace packet

Note

An A-Sync packet in the data trace stream:

- Is an extension packet. See *Extension packets in the data trace stream on page 6-308*.
- Does not indicate any trace elements. A trace analyzer uses it to synchronize with the data trace stream.

An A-Sync packet type also exists in the instruction trace stream.

Like its counterpart in the instruction trace stream, an A-Sync packet in the data trace stream is a unique sequence of bits that is used to identify the boundary of another packet. When a trace unit is first enabled, the first packet output is an A-Sync packet, and therefore this packet must be the first packet that a trace analysis tool searches for so that it can determine the start of a Trace Info packet.

The unique sequence of bits for the A-Sync packet in the data trace stream is exactly the same as it is in the instruction trace stream. That is, it is a header byte, `0b00000000`, followed by ten payload bytes of `0b00000000`, and one final payload byte of `0b10000000`. Any byte that immediately follows this sequence is the header byte of a new packet.

Figure 6-46 shows the A-Sync packet.

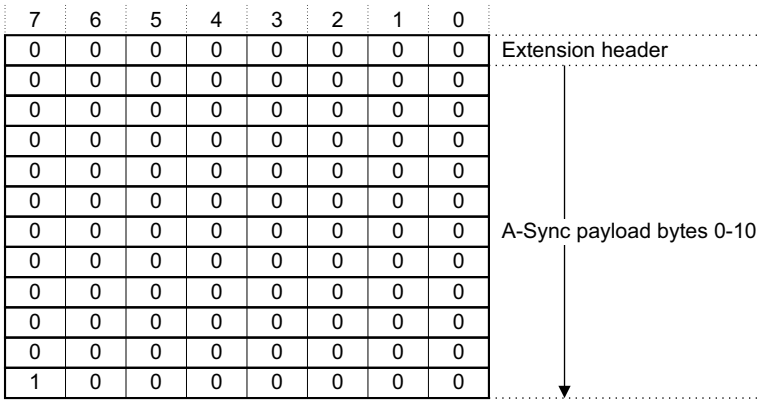


Figure 6-46 A-Sync data trace packet

In addition to being the first packet output whenever the trace unit is enabled, an A-Sync packet is also output:

- Periodically, based on trace synchronization requests. You can program the trace unit to automatically generate trace synchronization requests on a periodic basis. In addition, you can set the number of bytes of trace that are output between the trace synchronization requests. See *TRCSYNCPR, Synchronization Period Register on page 7-407*.
- After a trace unit buffer overflow, because if this happens, a trace synchronization request automatically occurs.

Trace Info data trace packet

Note

A Trace Info packet in the data trace stream is an Extension packet. See *Extension packets in the data trace stream on page 6-308*.

A Trace Info packet type also exists in the instruction trace stream.

For a description of the Trace Info data trace element, see *Trace Info data trace element on page 5-226*.

The purpose of a Trace Info packet is normally to:

- Signify a point in the trace stream from where analysis can begin.

- Provide a trace analyzer with information about the setup of the trace, for the particular trace run.

Indeed, the Trace Info packet in the instruction trace stream performs both of these functions.

However, in the data trace stream, the Trace Info packet does not contain any trace setup information. The reason for this is that for each trace run, the setup information is contained in the Trace Info instruction trace packet. See *Trace Info instruction trace packet on page 6-254*. Therefore, a Trace Info data trace packet only provides a point in the data trace stream where analysis of the trace stream can begin.

The Trace Info data trace packet is shown in Figure 6-47.

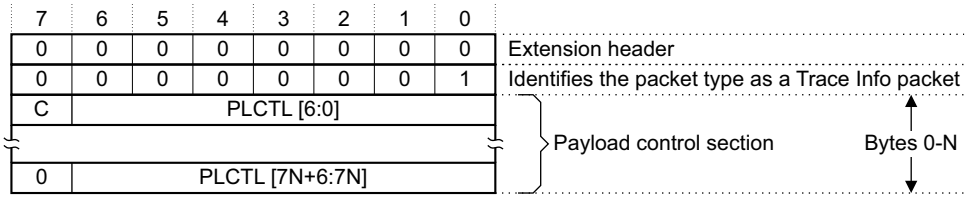


Figure 6-47 Trace Info data trace packet

In the data trace stream, a Trace Info packet is output after an A-Sync packet, when the trace unit is first enabled.

The fields in the payload sections of the Trace Info packet are:

PLCTL This is the payload control field. The bits in this field indicate which other payload sections are present in the packet, as described for the *Trace Info instruction trace packet on page 6-255*. For the Trace Info packet in the data trace stream, all bits are reserved.

A trace unit must not output more than one PLCTL byte in a Trace Info packet.

C The continuation bit. If a byte in the PLCTL section has this bit set to 1, then another byte follows. If a byte in PLCTL has this bit set to 0, then it is the last byte in the section, and therefore the last byte in the packet.

The `TraceInfoPacket()` function for the data trace stream is:

```
// TraceInfoPacket()
//=================

TraceInfoPacket()
    timestamp = 0;
    address_regs[0] = 0;
    address_regs[1] = 0;
    address_regs[2] = 0;
    endianness = LITTLE;
    p1_left_key = 0;
    p1_right_key = 0;
    p1_index = 0;
    p2_left_key = 0;
    emit(trace_info_element());
```

Discard data trace packet

Note

A Discard packet in the data trace stream is an extension packet. See *Extension packets in the data trace stream on page 6-308*.

A Discard packet type also exists in the instruction trace stream.

For a description of the Discard data trace element, see *Discard data trace element on page 5-226*.

In the data trace stream, a Discard packet indicates that tracing has become inactive, and that as a result, some P1 elements that are already output might not have P2 elements generated for them.

The format of a Discard packet in the data trace stream is identical to that of the Discard packet in the instruction trace stream. That is, it consists of a header byte that identifies it as an extension packet, and one payload byte, as shown in Figure 6-48.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Extension header
0	0	0	0	0	0	1	1	Identifies the packet type as a Discard packet

Figure 6-48 Discard data trace packet

For descriptions of the Discard instruction trace element, and Discard data trace element, see *Discard instruction trace element on page 5-191* and *Discard data trace element on page 5-226* respectively.

The `DiscardPacket()` function for the data trace stream is:

```
//DiscardPacket()
//==============

DiscardPacket()
    emit(discard_element());
```

Overflow data trace packet

Note

An Overflow packet in the data trace stream is an extension packet. See *Extension packets in the data trace stream on page 6-308*.

See *Overflow data trace element on page 5-226* for more information.

An Overflow packet type also exists in the instruction trace stream. See *Overflow instruction trace packet on page 6-258*.

An Overflow packet is output in the data trace stream whenever the data trace buffer in the trace unit overflows. This means that part of the data trace stream might be lost, and tracing is inactive until the overflow condition clears.

After an Overflow packet is output, the trace unit must output an A-Sync packet and a Trace Info packet to enable a trace analyzer to re-synchronize with the data trace stream. The trace unit is permitted to output Event, Discard, or Overflow packets before it outputs the A-Sync packet.

An Overflow packet in the data trace stream is identical in format to the Overflow packet in the instruction trace stream. That is, it consists of a header byte plus one payload byte, as shown in Figure 6-49.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Extension header
0	0	0	0	0	1	0	1	Identifies the packet type as an Overflow packet

Figure 6-49 Overflow data trace packet

The `OverflowPacket()` function for the data trace stream is:

```
//OverflowPacket()
//===============

OverflowPacket()
    emit(overflow_element());
    emit(discard_element());
```

Suppression data trace packet

Note

This packet type is only present in the data trace stream.

For a description of the Suppression trace element, see *Suppression data trace element on page 5-227*.

Some implementations of the ETMv4 enable the trace unit to discard some of the data trace elements it generates if there is a risk that the data trace buffer in the trace unit might overflow. For an implementation to have this capability, it is required that the *TRCSTALLCTLR, Stall Control Register on page 7-405*, is implemented. TRCSTALLCTLR contains a field, DATADISCARD, that you can use to choose what types of data trace elements you would prefer to discard. For example, you can discard only P1 elements that are associated with data load transfers, or only P1 elements that are associated with data store transfers, or both.

The process of discarding elements in this way, so that the risk of a data trace buffer overflow is reduced, is called suppression, and a Suppression packet is output in the data trace stream when the first P1 element is discarded. When the trace unit stops discarding P1 elements, tracing of these elements resumes.

It is ID Register 3, TRCIDR3, that tells you whether or not TRCSTALLCTLR is implemented. See *TRCIDR3, ID Register 3 on page 7-376*, and *TRCSTALLCTLR, Stall Control Register on page 7-405*.

A Suppression packet consists of only a header byte, as shown in Figure 6-50.

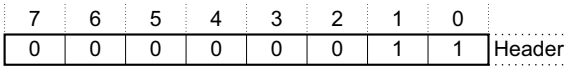


Figure 6-50 Suppression data trace packet

Note

- Whenever suppression is activated, if the trace unit has already generated P1 and P2 elements, but those elements have not been encoded into packets, then they must be encoded and output before the Suppression packet is output.
- Suppression is activated whenever the space left in the data trace buffer is less than the level set by the LEVEL field in the TRCSTALLCTLR.

The `SuppressionPacket()` function is:

```
//SuppressionPacket()
//==================

SuppressionPacket()
    emit(suppression_element());
```

6.5.3 Data Synchronization Marker (Data Sync Mark) data trace packets

Data Synchronization Marker packets are only output if the trace unit is programmed to output a data trace stream in addition to the instruction trace stream, because the purpose of data synchronization markers is to enable a trace analyzer to synchronize the two streams. When Data Synchronization Marker packets are output, a packet is output in both trace streams. For each Data Synchronization Marker packet output in the instruction trace stream, a matching Data Synchronization Marker packet is output in the data trace stream.

As described in *Data Synchronization Marker (Data Sync Mark) instruction trace packets on page 6-270*, in the instruction trace stream, there are two types of Data Synchronization Marker packet:

- Numbered Data Synchronization Marker packets.
- Unnumbered Data Synchronization Marker packets.

This is the same for the data trace stream, that is, the same two packet types exist in the data trace stream. However, the format of the packets differs from those in the instruction trace stream. The following two sections describe each packet type for the data trace stream.

Numbered Data Synchronization Marker data trace packet

Note

A Numbered Data Synchronization Marker packet in the data trace stream is an extension packet. See *Extension packets in the data trace stream on page 6-308*.

A Numbered Data Synchronization Marker packet also exists in the instruction trace stream.

For a description of the Data Synchronization Marker data trace element, see *Data Synchronization Marker (Data Sync Mark) data trace element on page 5-231*.

A Numbered Data Synchronization Marker provides an approximate correlation of the data trace stream with the instruction trace stream, because a Numbered Data Sync Marker packet in the data trace stream corresponds to the Numbered Data Sync Marker packet with the same number in the instruction trace stream. The format of a Numbered Data Sync Marker packet in the data trace stream is as shown in Figure 6-51.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	Extension header
0	0	0	1	NUM			1	Identifies the packet as a Numbered Data Synchronization Marker packet

Figure 6-51 Numbered Data Synchronization Marker data trace packet

The NUM field contains the number of the Data Synchronization Marker element.

The `NumberedDataSynchronizationMarkerPacket()` function for the data trace stream is:

```
//NumberedDataSynchronizationMarkerPacket()
//=========================================

NumberedDataSynchronizationMarkerPacket()
    emit(numbered_sync_marker_element(UInt(NUM)));
```

Unnumbered Data Synchronization Marker data trace packet

An Unnumbered Data Synchronization Marker enables more accurate correlation between the instruction and data trace streams. If an Unnumbered Data Synchronization Marker packet is output, it is located in the data trace stream somewhere between two Numbered Data Synchronization Marker packets.

An Unnumbered Data Synchronization Marker packet in the data trace stream consists of only a header byte, as shown in Figure 6-52.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	Header

Figure 6-52 Unnumbered Data Synchronization Marker data trace packet

The `UnnumberedDataSynchronizationMarkerPacket()` function for the data trace stream is:

```
//UnnumberedDataSynchronizationMarkerPacket()
//===========================================

UnnumberedDataSynchronizationMarkerPacket()
    emit(sync_marker_element());
```

6.5.4 Global timestamping

If you have enabled global timestamping, the trace unit periodically outputs Timestamp packets in each trace stream. For a description of what scenarios cause the trace unit to output a Timestamp packet, see *Timestamp instruction trace element on page 5-215*. For instructions on how to enable timestamping, see *Global timestamping on page 2-82*.

Timestamp data trace packet

Note

- A Timestamp packet also exists in the instruction trace stream, see *Timestamp instruction trace packet on page 6-259*.
- For a description of the Timestamp data trace element, see *Timestamp data trace element on page 5-230*.

A Timestamp packet in the data trace stream consists of a header byte, plus 1-9 bytes that contain the timestamp value, as shown in Figure 6-53.

Timestamp values can have a maximum size of either 48 or 64 bits, depending on the implementation. The TRCIDR0 shows which maximum size is implemented. See *TRCIDR0, ID Register 0 on page 7-370*.

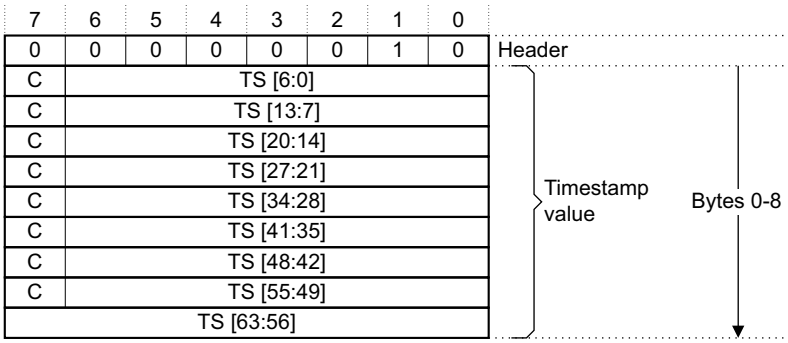


Figure 6-53 64-bit Timestamp packet in the data trace stream

The fields in the Timestamp packet are:

TS The Timestamp packet header is always followed by at least one byte of timestamp. The timestamp value is compressed, so that the trace unit generates only enough bytes of timestamp to output the least significant bits that have changed since the value given in the previous Timestamp packet.

If any bits in this field are not output, they are either:

- The same value as they were in the previous Timestamp packet.
- Zero, if they have not been output since the most recent Trace Info packet.

C The continuation bit indicates if there is another byte of timestamp information in the packet. If C is set to 1, then another TS byte follows. Otherwise, if C is set to 0, no more TS bytes follow.

Note

The maximum number of payload bytes that this packet can have is nine bytes. Therefore, if nine bytes are output, then the last byte does not contain a C bit because there can be no more bytes.

The `TimestampPacket()` function for the data trace stream is:

```
//TimestampPacket()
//================

TimestampPacket()
    timestamp = replace timestamp with new bits from TS, leaving other bits unchanged;
    emit(timestamp_element(timestamp));
```

6.5.5 P1 packet types

Note

For a description of P1 Data Address elements, see *P1 Data Address (P1) data trace elements on page 5-227*.

If the trace unit outputs a P1 packet type, it means that the trace unit has generated one or more P1 elements. A P1 element contains the address of a data transfer that the PE has performed as the result of executing a P0 instruction. P1 elements have a left-hand key, so that they can be associated with the correct P0 instruction element, and also a right-hand key, so that they can be associated with the correct P2 data value elements. In addition, P1 elements contain a transfer index that tells you where the address that the P1 element contains is, as an offset from the base address accessed by the P0 instruction.

For a description of how P0, P1, and P2 elements are related, see *Relationships between P0, P1, and P2 elements on page 2-38*.

The trace unit generates a new P1 element for each data transfer. This means that if one instruction results in multiple data transfers, a new P1 element is generated for each of those data transfers. P1 packets can indicate one or a number of P1 elements, depending on the packet type.

For more information about P0, P1 and P2 elements, see:

- *About instruction trace P0 elements on page 2-35*.

See also:

- *P1 Data Address (P1) data trace elements on page 5-227*.
- *P2 Data Value (P2) data trace elements on page 5-230*.

Compression techniques used when generating P1 packet types

When generating P1 packets, the trace unit uses compression techniques to minimize the amount of trace it generates. One of these techniques involves comparing the address contained in any new P1 element with the three most recent addresses that have been output, so that an optimal packet type can be generated. This compression technique is described as follows:

- The trace unit stores up to three of the most recent address values in a queue. The queue entries are `address_regs[0]`, `address_regs[1]`, and `address_regs[2]`. Each time a new P1 packet is output, the address indicated by that packet is stored at the top of the queue, in entry `address_regs[0]`, and the contents of the other entries are pushed downwards, so that the values that were stored in `address_regs[0]` are moved into `address_regs[1]`, the values that were stored in `address_regs[1]` are moved into `address_regs[2]`, and the values that were stored in `address_regs[2]` are discarded.

  Note

  — This queue is different to that used for Address packets in the instruction trace stream.
  — There is no requirement for an implementation to implement all three address stores.
  — The three stored addresses are cleared to zero whenever a Trace Info packet is output.

- Before generating a P1 Address packet, the trace unit compares the new data address value with each of the three stored addresses. It then chooses an optimum packet type depending on how many bits are different from the most similar stored address. For example, it might be the case that only bits[4:2] of a new data address are different from the address stored in `address_regs[2]`. In this case, the trace unit might output a P1 Format 2 packet type that is one byte long, rather than a P1 Format 3 packet type that is two bytes long or a P1 Format 1 packet type that is much longer.

When encoding a P1 Address element into a packet, the trace unit also compresses the values of the right-hand and left-hand keys that are associated with the P1 element. The key values are compressed relative to key values that have recently been output. This enables the trace unit to trace key values as small offsets from recently traced values, rather than explicitly tracing full key values for every P1 packet type.

When TRCCONFIGR.DA is `0b0`, the address and endianness included in a P1 element are UNKNOWN. However, the stored addresses are correctly updated by the actual address and endianness values output in a P1 packet even when TRCCONFIGR.DA is `0b0`.

Handling P1 packet types

The description of each packet type in this section includes pseudocode to explain how the packet describes the corresponding source element, and so how trace packets can be decoded. The P1 and P2 packet type pseudocode includes calls to a set of functions that manage the left- and right-hand keys and the address value queue. These functions are:

update_p1_left_key(integer value)

```
//update_p1_left_key(integer value)
//==================================

update_p1_left_key(integer value)
    p1_left_key = (p1_left_key + value) MOD p1_left_key_max;
```

update_p1_right_key(integer value)

```
//update_p1_right_key(integer value)
//==================================

update_p1_right_key(integer value)
    p1_right_key = (p1_right_key + value) MOD p1_right_key_max;
```

update_p2_left_key(integer value)

```
//update_p2_left_key(integer value)
//==================================

update_p2_left_key(integer value)
    p2_left_key = (p2_left_key + value) MOD p1_right_key_max;
```

update_address_regs(bits(64) address, integer reg)

```
//update_address_regs(bits(64) address, integer reg)
//=================================================

update_address_regs(bits(64) address, integer reg)
    case reg of
        when 2
            address_regs[2] = address_regs[1];
            address_regs[1] = address_regs[0];
        when 1
            address_regs[1] = address_regs[0];
        when 0
            if(address<63:6> != address_regs[0]<63:6>) then
                address_regs[2] = address_regs[1];
                address_regs[1] = address_regs[0];
    address_regs[0] = address;
```

is_key_special(integer key)

```
//is_key_special(integer key)
//===========================

boolean is_key_special(integer key)
    if(key >= p1_right_key_max) then
        return true;
    else
        return false;
```

Types of P1 packet

There are seven types of P1 packet:

- *P1 Format 1 data trace packet on page 6-317*. This packet type can contain a full address, plus the full values of both keys and the full value of the index, for one data transfer.
- *P1 Format 2 data trace packet on page 6-321*. This packet type contains only bits[5:2] of the address, for one word-aligned data transfer.

- *P1 Format 3 data trace packet on page 6-322*. This packet type contains only bits[9:2] of the address, for one word-aligned data transfer.
- *P1 Format 4 data trace packet on page 6-323*. This packet type contains only bits[3:2] of the address, for one halfword-aligned data transfer.
- *P1 Format 5 data trace packet on page 6-324*. This packet type does not contain an address. It is used when there is no requirement to trace the data address. Instead, it indicates that 1-4 P1 elements have been generated, and it provides the value of the left-hand key and the value of the transfer index for each of those elements.
- *P1 Format 6 data trace packet on page 6-325*. This packet type is used to indicate one P1 element when there is no requirement to trace the data address. The value of the left-hand key for the element is traced relative to a recently output key but the value of the right-hand key can be traced explicitly. This packet type also shows the value of the transfer index of the P1 element.
- *P1 Format 7 data trace packet on page 6-326*. This packet type updates the address given in the most recently traced P1 element. It does not indicate any new P1 elements, or provide values for right-hand keys, or an index value.

Table 6-23 shows the characteristics of each type of P1 packet.

Table 6-23 A summary of the characteristics of each P1 packet type

P1 packet type	Number of P1 elements	Can contain a nonzero index value?	Address alignment	Can contain a special P1 key?	Can indicate a change in endianness?
P1 Format 1	1	Yes	Any	Yes	Yes
P1 Format 2	1	No	Word-aligned	No	No
P1 Format 3	1	No	Word-aligned	No	No
P1 Format 4	1	No	Halfword-aligned	No	No
P1 Format 5	1-4	Yes	Any[a]	No	No
P1 Format 6	1	Yes	Any[a]	Yes	No
P1 Format 7	None	This packet type modifies the address of the most recently traced P1 element			

a. P1 Format 5 and Format 6 packets can only indicate P1 elements when those P1 elements contain address and endianness information that is not required by the trace analyzer. For an example of when the data address might not be required, see *P1 Format 5 data trace packet on page 6-324*.

For accesses to the PPB space on an Armv6-M, Armv7-M and Armv8-M PE, the endianness might be traced as little endian or big endian.

P1 Format 1 data trace packet

A P1 Format 1 packet can contain the full address of one data transfer, plus the full values of:

- The left-hand key, that enables association of the P1 element with the correct P0 element.
- The right-hand key, that enables association of the P1 element with the correct P2 element.
- The data transfer index, that tells you where the address that the P1 element contains is, as an offset from the base address accessed by the P0 instruction.

A P1 Format 1 packet consists of at least a header byte, plus an address payload section that contains at least one address byte. Other payload sections might also be present.

In summary, a P1 Format 1 packet contains:

1. An address payload section, ADDR. This section is always present.

2. A left-hand key payload section, LHKEY. This section might be present.
3. A right-hand key payload section, RHKEY. This section might be present.
4. An index payload section, INDEX. This section might be present.

The format of a P1 Format 1 packet is as shown in Figure 6-54.

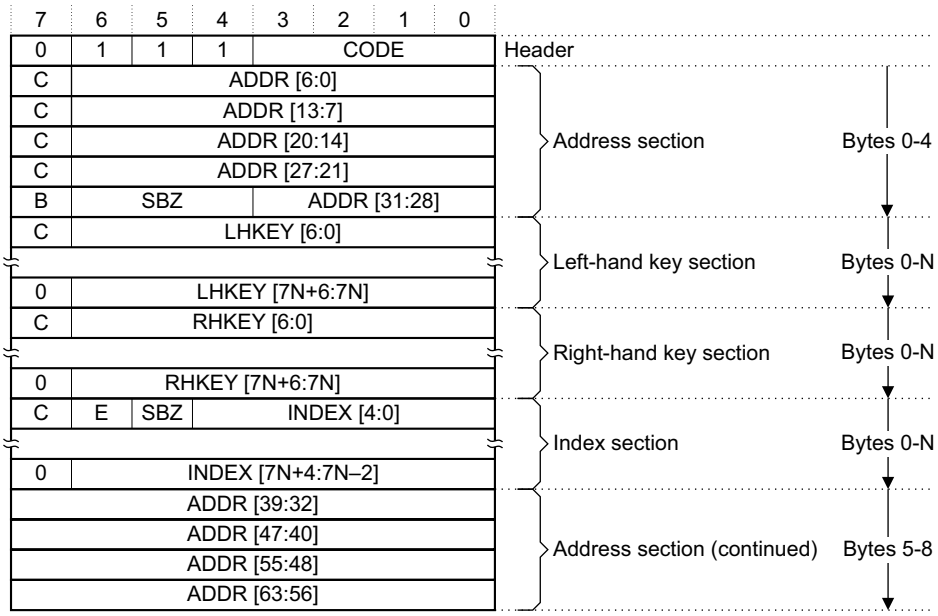


Figure 6-54 P1 Format 1 data trace packet

The fields in a P1 Format 1 packet are:

CODE The bits in this field indicate which other payload sections are present. In some cases, when other sections are not present, CODE indicates a value for the missing sections. In addition, the CODE field indicates which `address_regs[n]` the address has been compressed relative to. Table 6-24 on page 6-319 shows what the values of CODE mean.

ADDR The address of the data transfer that the PE has performed. This field is always present. The address is compressed relative to one of the values contained in either `address_regs[0]`, `address_regs[1]`, or `address_regs[2]`. For a description of how the trace unit compresses the value of the address, see *Compression techniques used when generating P1 packet types on page 6-315*. Any bits of ADDR that are not output are the same as in the `address_regs[n]` register that the CODE field indicates.

B Indicates whether address bytes 5-8 are present at the end of the packet:

0 ADDR bytes 5-8 are not present.

1 ADDR bytes 5-8 are present.

Address bytes 5-8 are only present if address bytes 0-4 are present, because address bytes 5-8 are only used if the address is too long to fit into address bytes 0-4.

LHKEY The value contained in this field is the value of the left-hand key that belongs to the P1 element.

If any bits of the LHKEY field are not output, their value is 0.

A trace unit must not output more LHKEY bytes than are required to indicate `p1_left_key_max` –1. For example, if `p1_left_key_max` is 32, no more than one LHKEY byte must be output.

RHKEY The value contained in this field is the value of the right-hand key that belongs to the P1 element.

If any bits of the RHKEY field are not output, their value is 0.

A trace unit must not output more RHKEY bytes than are required to indicate TRCIDR10.NUMP1KEY. For example, if TRCIDR10.NUMP1KEY is 32, no more than one RHKEY byte must be output.

INDEX The value shown in this field is the index value of the P1 element.

If any bits of the INDEX field are not output, their value is 0.

A trace unit must not output more than one INDEX byte in a P1 Format 1 packet.

E The endianness of the P1 Data Address element, that is, the endianness of the address value that is contained in the ADDR field:

0 LITTLE.

1 BIG.

If this bit is not output, then the `endianness` value that is stored in the trace analyzer between receiving packets is used. See *Trace analyzer state between receiving data trace packets on page 6-241*.

The E bit is present if the index bytes are present.

C The continuation bit. If a byte in a section has this bit set to 1, then another byte follows in the same section. If a byte in a section has this bit set to 0, then it is the last byte in the section.

Table 6-24 shows what the values of CODE indicate.

Table 6-24 Meaning of the CODE field in a P1 Format 1 packet

CODE value	`address_regs[N]` that ADDR is compressed relative to	LHKEY field[a]	Value of left-hand key when LHKEY field is not present	RHKEY field[b]	INDEX field[c]
`0b0000`	0	Not present	`p1_left_key` – 2	Not present	Not present
`0b0001`	0	Not present	`p1_left_key` – 1	Not present	Not present
`0b0010`	0	Not present	`p1_left_key` + 1	Not present	Not present
`0b0011`	0	Not present	`p1_left_key` + 2	Not present	Not present
`0b0100`	0	Not present	`p1_left_key` + 3	Not present	Not present
`0b0101`	1	Not present	`p1_left_key` + 1	Not present	Not present
`0b0110`	1	Not present	`p1_left_key` + 2	Not present	Not present
`0b0111`	1	Not present	`p1_left_key` + 3	Not present	Not present
`0b1000`	2	Not present	`p1_left_key` + 1	Not present	Not present
`0b1001`	2	Not present	`p1_left_key` + 2	Not present	Not present
`0b1010`	0	Present	-	Not present	Not present
`0b1011`	1	Present	-	Not present	Not present
`0b1100`	2	Present	-	Not present	Not present
`0b1101`	0	Present	-	Present	Not present
`0b1110`	0	Not present	`p1_left_key` + 1	Not present	Present
`0b1111`	0	Present	-	Present	Present

a. When the LHKEY section is not present, the left-hand key for the P1 element is relative to `p1_left_key` as indicated in the next column.

b. When the RHKEY section is not present, the right-hand key for the P1 element is always (`p1_right_key`+1) MOD `p1_right_key_max`.

c. When the index section is not present, the index value of the P1 element is zero and the endianness is the value of `endianness`. See *Trace analyzer state between receiving data trace packets on page 6-241*.

The `P1Format1Packet()` function is:

```
//P1Format1Packet()
//================

P1Format1Packet()
    case CODE of
        when ‘0000’ reg = 0;
        when ‘0001’ reg = 0;
        when ‘0010’ reg = 0;
        when ‘0011’ reg = 0;
        when ‘0100’ reg = 0;
        when ‘0101’ reg = 1;
        when ‘0110’ reg = 1;
        when ‘0111’ reg = 1;
        when ‘1000’ reg = 2;
        when ‘1001’ reg = 2;
        when ‘1010’ reg = 0;
        when ‘1011’ reg = 1;
        when ‘1100’ reg = 2;
        when ‘1101’ reg = 0;
        when ‘1110’ reg = 0;
        when ‘1111’ reg = 0;

    address = Construct address from ADDR and address_regs[reg];

    update_address_regs(address,reg);

    // Default right-hand key is +1, but this might change based on CODE.
    // Default index is 0, but this might change based on CODE.
    update_p1_right_key(+1);
    p1_index = 0;
    integer this_right_key = p1_right_key;

    case CODE of
        when ‘0000’ update_p1_left_key(-2);
        when ‘0001’ update_p1_left_key(-1);
        when ‘0010’ update_p1_left_key(+1);
        when ‘0011’ update_p1_left_key(+2);
        when ‘0100’ update_p1_left_key(+3);
        when ‘0101’ update_p1_left_key(+1);
        when ‘0110’ update_p1_left_key(+2);
        when ‘0111’ update_p1_left_key(+3);
        when ‘1000’ update_p1_left_key(+1);
        when ‘1001’ update_p1_left_key(+2);
        when ‘1010’ p1_left_key = LHKEY;
        when ‘1011’ p1_left_key = LHKEY;
        when ‘1100’ p1_left_key = LHKEY;
        when '1101'
            p1_left_key = LHKEY;
            this_right_key = RHKEY;
            if (!is_key_special(RHKEY)) then
                p1_right_key = RHKEY;
        when ‘1110’
            update_p1_left_key(+1);
            p1_index = INDEX;
            endianness = if E then BIG else LITTLE;
        when ‘1111’
            p1_left_key = LHKEY;
            this_right_key = RHKEY;
            if (!is_key_special(RHKEY)) then
                p1_right_key = RHKEY;
            p1_index = INDEX;
            endianness = if E then BIG else LITTLE;

    emit(p1_data_address_element(address,
                                 endianness
                                 p1_left_key
                                 this_right_key
                                 p1_index));
```

P1 Format 2 data trace packet

A P1 Format 2 packet is output when the address of a new data transfer differs only slightly from one of the address values contained in either `address_regs[0]`, `address_regs[1]`, or `address_regs[2]`. It does this by containing only bits[5:2] of the new address value. It does not contain bits[1:0] because this packet type indicates a word-aligned address, therefore bits[1:0] always have the value `0b00`.

In addition, this packet type indicates:

- The register, `address_regs[n]`. The address is compressed relative to the content of the register.
- The left-hand key value, that enables association of the P1 element with the correct P0 element.

Also, when this packet type appears in the trace stream, it implies that:

- The right-hand key value for the P1 element is an increment of one on the present value, `p1_right_key`.
- The transfer index of the P1 element is zero.

A P1 Format 2 packet consists of only a header byte, as shown in Figure 6-55.

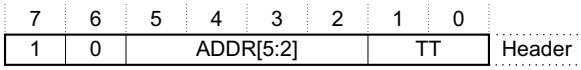


Figure 6-55 P1 Format 2 data trace packet

The fields in a P1 Format 2 packet are:

ADDR This field contains four bits, [5:2], of the address of the data transfer. Bits[1:0] of the address are not included in the packet because the address indicated by this packet type is word-aligned, therefore bits[1:0] always have the value `0b00`. All other bits of ADDR are the same as given in the relevant `address_regs[n]`, as indicated by the TT field.

TT This field indicates whether the address value is compressed relative to `address_regs[0]`, `address_regs[1]`, or `address_regs[2]`. It also indicates the value of the left-hand key. The possible values for TT are:

`0b00` `address_regs[0]`. The value of the left-hand key is `p1_left_hand_key`+1.

`0b01` `address_regs[1]`. The value of the left-hand key is `p1_left_hand_key`+1.

`0b10` `address_regs[2]`. The value of the left-hand key is `p1_left_hand_key`+1.

`0b11` `address_regs[0]`. The value of the left-hand key is `p1_left_hand_key`+2.

The `P1Format2Packet()` function is:

```
//P1Format2Packet()
//================

P1Format2Packet()
    integer reg;
    case TT of
        when ‘00’ reg = 0;
        when ‘01’ reg = 1;
        when ‘10’ reg = 2;
        when ‘11’ reg = 0;

    address<63:6> = address_regs[reg]<63:6>;
    address<5:2> = ADDR;
    address<1:0> = ‘00’;

    update_address_regs(address,reg);

    update_p1_right_key(+1);
    p1_index = 0;
    case TT of
        when ‘00’ update_p1_left_key(+1);
        when ‘01’ update_p1_left_key(+1);
        when ‘10’ update_p1_left_key(+1);
        when ‘11’ update_p1_left_key(+2);
```

```
    emit(p1_data_address_element(address,
                                 endianness
                                 p1_left_key
                                 p1_right_key
                                 p1_index));
```

P1 Format 3 data trace packet

A P1 Format 3 packet contains more bits of the data address than a P1 Format 2 packet. A P1 Format 3 packet contains bits[9:2] of the address, whereas a P1 Format 2 packet contains only bits[5:2] of the address. However, like a P1 Format 2 packet, this packet indicates a word-aligned address, and therefore the bottom two bits of the address, [1:0], always have the value `0b00`.

Like a P1 Format 2 packet, this packet type also indicates:

- The register, `address_regs[n]`. The address is compressed relative to the content of this register.
- The left-hand key value, that enables association of the P1 element with the correct P0 element.

In addition, and also like a P1 Format 2 packet, the presence of this packet implies that:

- The right-hand key value for the P1 element is an increment of one on the present value, `p1_right_key`.
- The transfer index of the P1 element is zero.

A P1 Format 3 packet consists of a header byte plus one address payload byte, as shown in Figure 6-56.

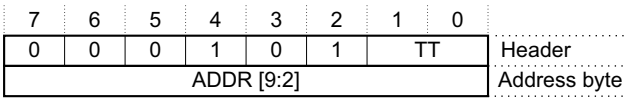


Figure 6-56 P1 Format 3 data trace packet

The fields in a P1 Format 3 packet are:

ADDR This field contains eight bits, [9:2], of the address of the data transfer. Bits[1:0] of the address are not included in the packet because this packet type contains word-aligned addresses, therefore bits[1:0] always have the value `0b00`. All other bits of ADDR are the same as given in the relevant `address_regs[n]`, as indicated by the TT field.

TT This field indicates whether the address value is compressed relative to `address_regs[0]`, `address_reg[1]`, or `address_regs[2]`. It also indicates the value of the left-hand key. The possible values for TT are:

- `0b00` `address_regs[0]`. The value of the left-hand key is `p1_left_hand_key`+1.
- `0b01` `address_regs[1]`. The value of the left-hand key is `p1_left_hand_key`+1.
- `0b10` `address_regs[2]`. The value of the left-hand key is `p1_left_hand_key`+1.
- `0b11` `address_regs[0]`. The value of the left-hand key is `p1_left_hand_key`+2.

The `P1Format3Packet()` function is:

```
//P1Format3Packet()
//================

P1Format3Packet()
    integer reg;
    case TT of
        when '00' reg = 0;
        when '01' reg = 1;
        when '10' reg = 2;
        when '11' reg = 0;

    address<63:10> = address_regs[reg]<63:10>;
    address<9:2> = ADDR;
    address<1:0> = '00';

    update_address_regs(address,reg);
```

```
    update_p1_right_key(+1);
    p1_index = 0;
    case TT of
        when '00' update_p1_left_key(+1);
        when '01' update_p1_left_key(+1);
        when '10' update_p1_left_key(+1);
        when '11' update_p1_left_key(+2);

    emit(p1_data_address_element(address,
                                 endianness
                                 p1_left_key
                                 p1_right_key
                                 p1_index));
```

P1 Format 4 data trace packet

A P1 Format 4 packet is different from P1 Format 2 and P1 Format 3 packets in that the address it contains is halfword-aligned, rather than word-aligned. Only bits[3:2] of the new address are contained in this packet, therefore it can only show addresses that differ only slightly from one of the values stored in either `address_regs[0]`, `address_regs[1]`, or `address_regs[2]`.

Like P1 Format 2 and Format 3 packets, this packet type also indicates:

- The register, `address_regs[n]`. The address is compressed relative to the content of this register.
- The left-hand key value, that enables association of the P1 element with the correct P0 element.

Also like P1 Format 2 and Format 3 packets, this packet implies that:

- The right-hand key value for the P1 element is an increment of one on the present value, `p1_right_key`.
- The transfer index of the P1 element is zero.

A P1 Format 4 packet consists of only a header byte, as shown in Figure 6-57.

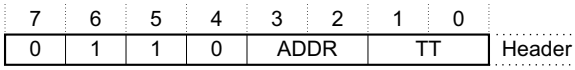


Figure 6-57 P1 Format 4 data trace packet

The fields in a P1 Format 4 packet are:

ADDR This field contains two bits, [3:2], of the address of the data transfer. Bits[1:0] of the address are not included. In this case, the value of bits[1:0] are always `0b10` but the packet can indicate halfword-aligned addresses. All other bits of ADDR are the same as given in the relevant `address_regs[n]`, as indicated by the TT field.

TT This field indicates whether the address value is compressed relative to `address_regs[0]`, `address_regs[1]`, or `address_regs[2]`. It also indicates the value of the left-hand key. The possible values for TT are:

- `0b00` `address_regs[0]`. The value of the left-hand key is `p1_left_hand_key`+1.
- `0b01` `address_regs[1]`. The value of the left-hand key is `p1_left_hand_key`+1.
- `0b10` `address_regs[2]`. The value of the left-hand key is `p1_left_hand_key`+1.
- `0b11` `address_regs[0]`. The value of the left-hand key is `p1_left_hand_key`+2.

The `P1Format4Packet()` function is:

```
//P1Format4Packet()
//================

P1Format4Packet()
    integer reg;
    case TT of
        when '00' reg = 0;
        when '01' reg = 1;
        when '10' reg = 2;
```

```
    when '11' reg = 0;
address<63:4> = address_regs[reg]<63:4>;
address<3:2> = ADDR;
address<1:0> = '10';

update_address_regs(address,reg);

update_p1_right_key(+1);
p1_index = 0;
case TT of
    when '00' update_p1_left_key(+1);
    when '01' update_p1_left_key(+1);
    when '10' update_p1_left_key(+1);
    when '11' update_p1_left_key(+2);

emit(p1_data_address_element(address,
                             endianness
                             p1_left_key
                             p1_right_key
                             p1_index));
```

P1 Format 5 data trace packet

A P1 Format 5 packet type can indicate 1-4 P1 elements, when there is no requirement to trace the addresses contained in those elements. This situation might occur, for example, when a single instruction that results in multiple data transfers to contiguous addresses is executed. In this case, a new P1 element is generated for each data transfer but a trace analyzer only requires the address contained in one of those P1 elements. This is because the addresses for the other P1 elements can be inferred from that one address. If the one address that is required is traced as part of a past or future P1 packet, then a P1 Format 5 packet might be output to indicate some of the other P1 elements that resulted from the instruction. Figure 5-6 on page 5-230 shows this.

This packet type might also be output if you have disabled the tracing of data addresses by setting the DA bit in the TRCCONFIGR to zero. See *TRCCONFIGR, Trace Configuration Register on page 7-361* for more information.

A P1 Format 5 packet therefore only indicates:

- The left-hand key values, that enable association of each of the P1 elements with their correct P0 element.
- The value of the transfer index for each P1 element.

Also like P1 Format 2, Format 3, and Format 4 packet types, this packet implies that:

- The right-hand key value for the P1 element is an increment of one on the present value, `p1_right_key`.

A P1 Format 5 packet consists of only a header byte, as shown in Figure 6-58.

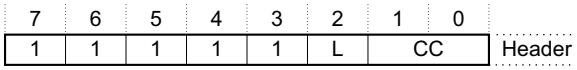


Figure 6-58 P1 Format 5 data trace packet

The fields in a P1 Format 5 packet are:

L This bit indicates the value of the left-hand key and the value of the index for each P1 element, as follows:

- **0** The value of `p1_left_key` is unchanged but the value of `p1_index` is `p1_index`+1. This might occur if the packet signifies up to four data transfers that belong to the same P0 instruction.
- **1** The value of `p1_left_key` is `p1_left_key`+1. The value of `p1_index` is zero. This might occur if the packet signifies up to four data transfers that each belong to separate P0 instructions.

CC The value given in this field is the number of P1 elements that the packet signifies. The count can range from 1-4.

Note

The values held in `address_regs[0]`, `address_regs[1]`, and `address_regs[2]` are not updated when a P1 Format 5 packet is output, because this packet does not contain or indicate an address.

The `P1Format5Packet()` function is:

```
//P1Format5Packet()
//================

P1Format5Packet
    for I = 0 to UInt(CC)
        if (L) then
            p1_index = 0;
            update_p1_left_key(+1);
        else
        p1_index = p1_index + 1;

        update_p1_right_key(+1);

        emit(p1_data_address_element(NOT_PROVIDED,
                                     NOT_PROVIDED,
                                     p1_left_key,
                                     p1_right_key,
                                     p1_index));
```

P1 Format 6 data trace packet

A P1 Format 6 packet type indicates one P1 element when there is no requirement to trace the address of that element. The description given in the P1 Format 5 data trace packet section describes a situation when this might occur. See *P1 Format 5 data trace packet on page 6-324*.

A P1 Format 6 packet indicates:

- The left-hand key value, that enables association of the P1 element with the correct P0 element.
- The value of the transfer index.

In addition, and unlike P1 Formats 2-5 packets, a P1 Format 6 packet can contain the full value of:

- The right-hand key, that enables association of the P1 element with the correct P2 elements.

A P1 Format 6 packet consists of a header byte, plus a payload section that consists of at least one byte of right-hand key value. The format is shown in Figure 6-59.

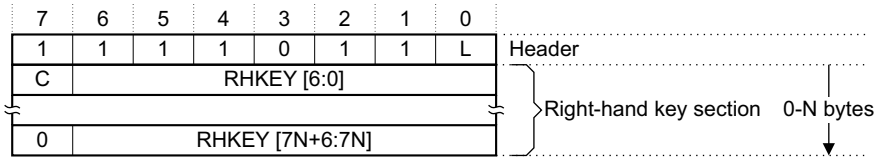


Figure 6-59 P1 Format 6 data trace packet

The fields in a P1 Format 6 packet are:

L This bit indicates the value of the left-hand key and the value of the index for the P1 element, as follows:

- **0** The value of `p1_left_key` is unchanged but the value of `p1_index` is `p1_index`+1.
- **1** The value of `p1_left_key` is `p1_left_key`+1. The value of `p1_index` is zero.

RHKEY The value contained in this field is the value of the right-hand key for the P1 element that the packet signifies. Any bits that are not output have the value zero.

A trace unit must not output more RHKEY bytes than are required to indicate TRCIDR10.NUMP1KEY. For example, if TRCIDR10.NUMP1KEY is 32, no more than one RHKEY byte must be output.

C The continuation bit indicates if there is another right-hand key byte in the packet. If C is set to 1, then another right-hand key byte follows. Otherwise, if C is set to 0, no more right-hand key bytes follow.

Note

The values held in `address_regs[0]`, `address_regs[1]`, and `address_regs[2]` are not updated when a P1 Format 6 packet is output, because this packet does not contain or indicate an address.

The `P1Format6Packet()` function is:

```
//P1Format6Packet()
//================

P1Format6Packet()
    if (L) then
        p1_index = 0;
        update_p1_left_key(+1);
    else
        p1_index = p1_index + 1;

    // The default operation is to increment the right-hand key.
    update_p1_right_key(+1);
    // Get the right-hand key for this element, but do not update the stored key
    // if this is a special key.
    integer this_right_key = RHKEY;
    if (!is_key_special(this_right_key)) then
        p1_right_key = this_right_key;

    emit(p1_data_address_element(NOT_PROVIDED,
                                 NOT_PROVIDED,
                                 p1_left_key,
                                 this_right_key,
                                 p1_index));
```

P1 Format 7 data trace packet

A P1 Format 7 packet updates the address given in the most recently traced P1 element. It does not indicate any new P1 elements, or provide values for right-hand or left-hand keys or an index value.

Bits of the address that can be updated are [63:56].

Note

Because this packet updates the address of the most recent P1 element to be traced, the value in `address_regs[0]` is updated.

A P1 Format 7 packet consists of a header byte plus one address update byte, as shown in Figure 6-60.

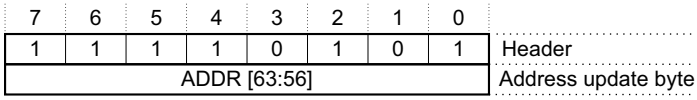


Figure 6-60 P1 Format 7 packet

The `P1Format7Packet()` function is:

```
//P1Format7Packet()
//================
```

```
P1Format7Packet()
    address<63:56> = ADDR;
    address<55:0> = address_regs[0];

    update last p1 data address element with the new address;

    address_regs[0] = address;
```

6.5.6 P2 packet types

Note

For a description of P2 Data Value elements, see *P2 Data Value (P2) data trace elements on page 5-230*.

If the trace unit outputs a P2 packet type, it means that the trace unit has generated one or more P2 elements. A P2 element contains the value that was transferred in a data transfer that has been traced as a P1 element. P2 elements have a left-hand key, so that they can be associated with the correct P1 data address element, and also a data value that might be up to 64 bits in length.

For a description of how P0, P1, and P2 elements are related, see *Relationships between P0, P1, and P2 elements on page 2-38*.

The following techniques are used to minimize the trace generated by tracing data values:

- If a small value is being traced, in the range 0-4, a small packet format is used.
- Any leading zeros in a data value held in a P2 packet type are suppressed.
- Left-hand keys are compressed relative to recent values, so that the full key does not need to be traced in each packet.

There are six types of P2 packet:

- *P2 Format 1 data trace packet*.
- *P2 Format 2 data trace packet on page 6-329*.
- *P2 Format 3 data trace packet on page 6-330*.
- *P2 Format 4 data trace packet on page 6-331*.
- *P2 Format 5 data trace packet on page 6-331*.
- *P2 Format 6 data trace packet on page 6-332*.

P2 Format 1 data trace packet

A P2 Format 1 packet type can indicate one data value that is a maximum of 64 bits in length. In addition, this packet can contain a value for the left-hand key of the P2 element. The left-hand key enables association of the P2 element with the correct P1 parent element.

A P2 Format 1 packet consists of a header byte, plus payload sections as follows:

- A left-hand key payload section. This section might be present.
- A data value payload section. This section is always present.

The format of a P2 Format 1 packet is shown in Figure 6-61 on page 6-328.

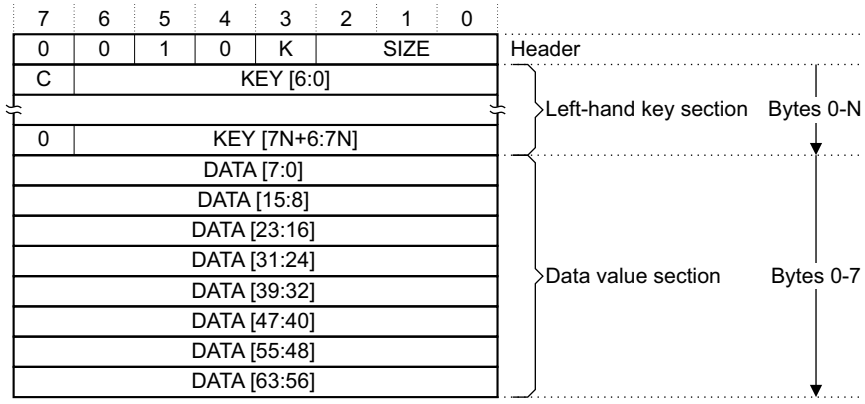


Figure 6-61 P2 Format 1 data trace packet

The fields in a P2 Format 1 packet are:

SIZE The SIZE field indicates the number of DATA bytes present in the data value section. The possible values are:

`0b000` 0 DATA bytes.

`0b001` 1 DATA byte.

`0b010` 2 DATA bytes.

`0b011` 3 DATA bytes.

`0b100` 4 DATA bytes.

`0b101` 6 DATA bytes.

`0b110` 8 DATA bytes.

`0b111` Reserved.

Note

The SIZE field does not indicate the size of the data value. It only indicates how many DATA bytes are present in the data value payload section.

K Indicates if the left-hand key section is present in the packet:

0 The left-hand section is not present in the packet. In this case, the value of the left-hand key is incremented by one from the previous value of `p2_left_key`.

1 The left-hand key section is present in the packet.

KEY The value contained in this field is the value of the left-hand key for the P2 element. If any bits of the KEY field are not output, their value is zero.

A trace unit must not output more KEY bytes than are required to indicate TRCIDR10.NUMP1KEY. For example, if TRCIDR10.NUMP1KEY is 32, no more than one KEY byte must be output.

DATA The value contained in this field is the data value of the data transfer. The SIZE field indicates the number of bytes that are present in the data value payload section. Any bits of the DATA field which are not output are zero.

C The continuation bit. In this packet, continuation bits are only present in the left-hand key payload section, if that section is present in the packet. The C bit indicates if there is another byte in the section. If a left-hand key byte has C set to 1, then another left-hand key byte follows. Otherwise, if C is set to 0, no more left-hand key bytes follow.

The `P2Format1Packet()` function is:

```
//P2Format1Packet()
```

```
//================

P2Format1Packet()
    integer this_left_key;
    if (K) then
        this_left_key = KEY;
        if (is_key_special(this_left_key)) then
            update_p2_left_key(+1);
        else
            p2_left_key = this_left_key;

    else
        update_p2_left_key(+1);
        this_left_key = p2_left_key;
    emit(p2_data_value_element(DATA,this_left_key));
```

P2 Format 2 data trace packet

A P2 Format 2 packet indicates one P2 element, when:

- The data value contained in the P2 element is sixteen bits or less in length.
- The value of the left-hand key for the P2 element is offset by a small amount from the previous value of `p2_left_key`.

A P2 Format 2 packet consists of a header byte, plus a data value payload section, as shown in Figure 6-62.

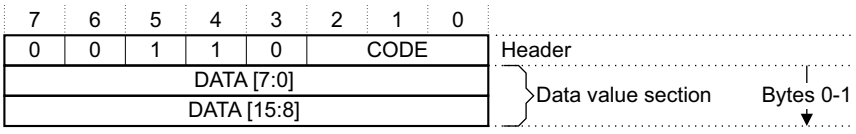


Figure 6-62 P2 Format 2 data trace packet

The fields in a P2 Format 2 packet are:

CODE Indicates the value of the left-hand key for the P2 element, as an offset from the previous value of `p2_left_key`. This field also indicates the number of bytes present in the data value payload section. Table 6-25 lists the possible values.

Table 6-25 Possible values for the CODE field of a P2 Format 2 packet

CODE value	Left-hand key value	Number of DATA bytes
`0b000`	Previous `p2_left_key` value +2	0
`0b001`	Previous `p2_left_key` value +2	1
`0b010`	Previous `p2_left_key` value +2	2
`0b011`	Previous `p2_left_key` value +3	0
`0b100`	Previous `p2_left_key` value +3	1
`0b101`	Previous `p2_left_key` value +3	2
`0b110`	Previous `p2_left_key` value –1	1
`0b111`	Previous `p2_left_key` value –1	2

DATA The value contained in this field is the data value of the data transfer. The CODE field indicates the number of DATA bytes that are present in the data value payload section.

If any bits in this field are not output, their value is zero.

Bits[63:16] of the data value are zero.

Note

The number of DATA bytes that are output does not indicate the size of the data transfer performed. For example, even though only zero to two bytes can be output, the data transfer might actually be 64 bits in length, where the majority of the leading bits are zero. Therefore, the trace unit employs leading zero compression and only outputs the number of least significant bytes that are required to indicate the data value.

The `P2Format2Packet()` function is:

```
//P2Format2Packet()
//================

P2Format2Packet()
    case CODE of
        when '000' update_p2_left_key(+2);
        when '001' update_p2_left_key(+2);
        when '010' update_p2_left_key(+2);
        when '011' update_p2_left_key(+3);
        when '100' update_p2_left_key(+3);
        when '101' update_p2_left_key(+3);
        when '110' update_p2_left_key(-1);
        when '111' update_p2_left_key(-1);
    emit(p2_data_value_element(DATA,p2_left_key));
```

P2 Format 3 data trace packet

A P2 Format 3 packet indicates one P2 element, when:

- That element has either 32 or 64 bits of data.
- The value of the left-hand key for the P2 element is offset by a medium amount from the previous value of `p2_left_key`.

A P2 Format 3 packet consists of a header byte plus a data value payload section, as shown in Figure 6-63.

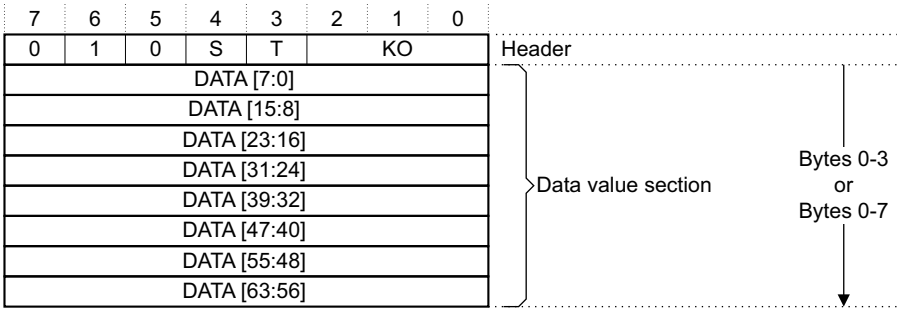


Figure 6-63 P2 Format 3 data trace packet

The fields in a P2 Format 3 packet are:

KO The value contained in this field forms part of the P2 left-hand key offset. The value of the T bit is required to interpret the meaning of this field.

T Use this field, and the value given in the KO field, to calculate the left-hand key offset:

- **0** The left-hand key value is offset from the previous value of `p2_left_key` by KO + 2.
- **1** The left-hand key value is offset from the previous value of `p2_left_key` by KO – 8.

S Indicates the number of bytes present in the data value payload section, as follows:

- **0** Four DATA bytes are present.

1 Eight DATA bytes are present.

DATA The value contained in this field is the data value of the data transfer. The S field indicates how many DATA bytes are present. If any bits of the DATA field are not output, their value is zero.

Note

The number of DATA bytes that are output does not indicate the size of the data transfer performed.

The `P2Format3Packet()` function is:

```
//P2Format3Packet()
//================

P2Format3Packet()
    integer offset;
    if (T) then
        offset = UInt(K0) - 8;
    else
        offset = UInt(K0) + 2;
    update_p2_left_key(offset);
    emit(p2_data_value_element(DATA,p2_left_key));
```

P2 Format 4 data trace packet

This packet type indicates one P2 element, when that element contains:

- A data value that is in the range 1-4.
- A left-hand key value that is incremented by one from the previous value of `p2_left_key`.

A P2 Format 4 packet consists of a header byte only, as shown in Figure 6-64.

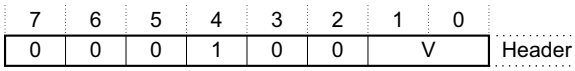


Figure 6-64 P2 Format 4 data trace packet

The fields in a P2 Format 4 packet are:

V Indicates the data value of the data transfer. The data value is V+1.

The `P2Format4Packet()` function is:

```
//P2Format4Packet()
//================

P2Format4Packet()
    update_p2_left_key(+1);
    emit(p2_data_value_element(V+1,p2_left_key));
```

P2 Format 5 data trace packet

A P2 Format 5 packet signifies one P2 element and up to four P1 elements when:

- The P2 element has either 32 or 64 bits of data value.
- There is no requirement to trace the addresses of any of the P1 elements. For an example of when the data address of a data transfer might not be required, see *P1 Format 5 data trace packet on page 6-324*.

A P2 Format 5 packet consists of a header byte plus a data value payload section, as shown in Figure 6-65 on page 6-332.

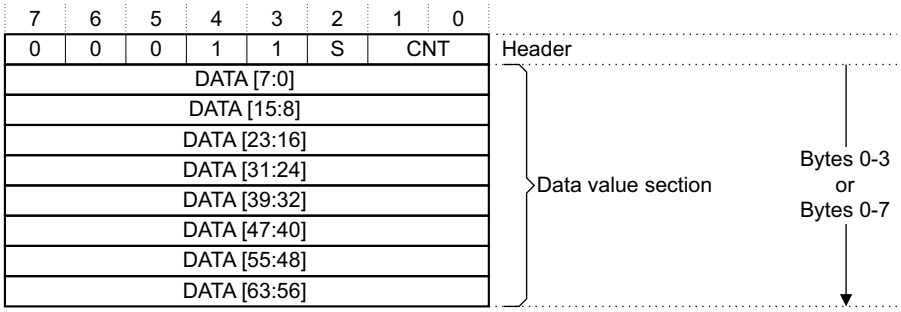


Figure 6-65 P2 Format 5 data trace packet

The fields in a P2 Format 5 packet are:

CNT Indicates the number of P1 elements that the packet signifies. 1-4 elements can be indicated. The number of P1 elements is CNT+1.

S Indicates the number of bytes present in the data value payload section, as follows:

0 Four DATA bytes are present.

1 Eight DATA bytes are present.

DATA The value contained in this field is the data value of the data transfer. The S field indicates how many DATA bytes are present. When only four bytes are present, bits[63:32] of the data value are zero.

Note

The number of DATA bytes that are output does not indicate the size of the data transfer performed.

The `P2Format5Packet()` function is:

```
//P2Format5Packet()
//================

P2Format5Packet()
    for I = 0 to UInt(CNT)
        p1_index = p1_index + 1;
        update_p1_right_key(+1);
        emit(p1_data_address_element(NOT_PROVIDED,
                                     NOT_PROVIDED,
                                     p1_left_key,
                                     p1_right_key,
                                     p1_index));
    update_p2_left_key(+1);
    emit(p2_data_value_element(DATA,p2_left_key));
```

P2 Format 6 data trace packet

A P2 Format 6 packet indicates two P2 elements and up to four P1 elements when:

- The P2 elements each have a 32-bit data value.
- There is no requirement to trace the addresses of any of the P1 elements. For an example of when the data address of a data transfer might not be required, see *P1 Format 5 data trace packet on page 6-324*.

A P2 Format 6 packet consists of a header byte and two data value payload sections, as shown in Figure 6-66 on page 6-333.

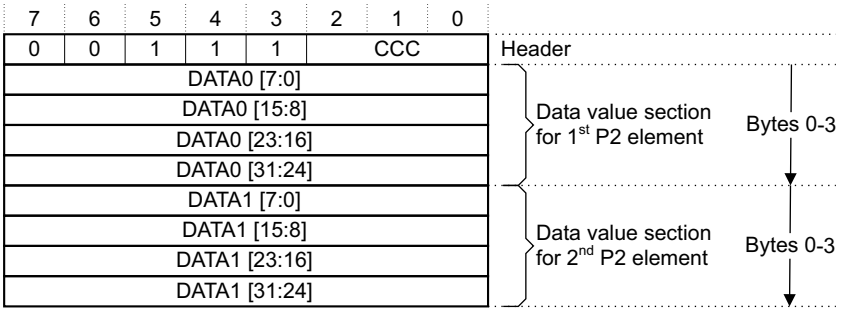


Figure 6-66 P2 Format 6 data trace packet

The fields in a P2 Format 6 packet are:

CCC Indicates the number of P1 elements, and the value of the left-hand key for the first P2 element, as shown in Table 6-26.

Table 6-26 Possible values for the CCC field of a P2 Format 6 packet

CCC value	Left-hand key value for first P2 element	Number of P1 elements
0b000	Previous p2_left_key value +1	0
0b001	Previous p2_left_key value +1	1
0b010	Previous p2_left_key value +1	2
0b011	Previous p2_left_key value +1	3
0b100	Previous p2_left_key value +1	4
0b101	Previous p2_left_key value +2	0
0b110	Previous p2_left_key value +3	0
0b111	Reserved	-

DATA0 The value contained in this field is the data value of the first data transfer, that is, the data value given in the first P2 element. Bits[63:32] of the data value are zero.

DATA1 Contains the data value of the second data transfer. Bits[63:32] are zero.

Note

- The left-hand key value for the second P2 element is always an increment of 1 on the left-hand key value for the first P2 element.
- For DATA0 and DATA1, the number of data value payload bytes that are output does not indicate the size of the data transfer performed.

The `P2Format6Packet()` function is:

```
//P2Format6Packet()
//================

P2Format6Packet()
    integer count = UInt(CCC);
    integer offset;
    if (count <= 4 && count > 0) then
        for I = 0 to count - 1
            p1_index = p1_index + 1;
            update_p1_right_key(+1);
```

```
            emit(p1_data_address_element(NOT_PROVIDED,
                                         NOT_PROVIDED,
                                         p1_left_key,
                                         p1_right_key,
                                         p1_index));

    case CCC of
        when '101' offset = 2;
        when '110' offset = 3;
        otherwise offset = 1;

    update_p2_left_key(offset);
    emit(p2_data_value_element(DATA0,p2_left_key));

    update_p2_left_key(+1);
    emit(p2_data_value_element(DATA1,p2_left_key));
```

6.5.7 Ignore packet

An Ignore packet is a single-byte packet that is ignored by a trace analyzer. The header value `0x05` is allocated for the Ignore packet in the data trace protocol, as shown in Figure 6-67. This header value is Reserved in ETMv4.2 or earlier.

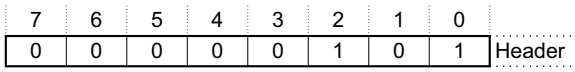


Figure 6-67 Ignore data trace packet

Ignore packets might be used to pad the trace stream to a convenient boundary. To avoid inefficient use of trace capture bandwidth or storage, Arm recommends that Ignore packets are not used frequently.

6.5.8 Event tracing data trace packet

Note

- An Event packet also exists in the instruction trace stream.
- For a description of the Event data trace element, see *Event data trace element on page 5-231*.

An Event packet in the data trace stream indicates that an event has occurred. This is a single-byte packet, as shown in Figure 6-68.

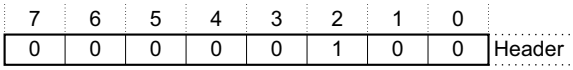


Figure 6-68 Event data trace packet

The `EventTracingPacket()` function for the data trace stream is:

```
//EventTracingPacket()
//====================

EventTracingPacket()
    emit(event_element());
```

Chapter 7
Register Descriptions

This chapter describes the registers in the ETMv4 architecture. It contains the following sections:

7.1 Register summary

Table 7-1 shows a list of the trace unit registers, in order of their offset from the base address.

Table 7-1 Register summary

Number	Offset	Name	Access	Width	Type	Description
0	0x000	-	-	-	-	Reserved
1	0x004	TRCPRGCTLR	RW	32	Trace	Programming Control Register
2	0x008	TRCPROCSELR	RW	32	Trace	PE Select Control Register
3	0x00C	TRCSTATR	RO	32	Trace	Trace Status Register
4	0x010	TRCCONFIGR	RW	32	Trace	Trace Configuration Register
5	0x014	-	-	-	-	Reserved
6	0x018	TRCAUXCTLR	RW	32	Trace	Auxiliary Control Register
7	0x01C	-	-	-	Trace	Reserved
8	0x020	TRCEVENTCTL0R	RW	32	Trace	Event Control 0 Register
9	0x024	TRCEVENTCTL1R	RW	32	Trace	Event Control 1 Register
10	0x028	-	-	-	-	Reserved
11	0x02C	TRCSTALLCTLR	RW	32	Trace	Stall Control Register
12	0x030	TRCTSCTLR	RW	32	Trace	Global Timestamp Control Register
13	0x034	TRCSYNCPR	RW[a]	32	Trace	Synchronization Period Register
14	0x038	TRCCCCTLR	RW	32	Trace	Cycle Count Control Register
15	0x03C	TRCBBCTLR	RW	32	Trace	Branch Broadcast Control Register
16	0x040	TRCTRACEIDR	RW	32	Trace	Trace ID Register
17	0x044	TRCQCTLR	RW	32	Trace	Q Element Control Register
18-31	0x048-0x07C	-	-	-	-	Reserved
32	0x080	TRCVICTLR	RW	32	Trace	ViewInst Main Control Register
33	0x084	TRCVIIECTLR	RW	32	Trace	ViewInst Include/Exclude Control Register
34	0x088	TRCVISSCTLR	RW	32	Trace	ViewInst Start/Stop Control Register
35	0x08C	TRCVIPCSSCTLR	RW	32	Trace	ViewInst Start/Stop PE Comparator Control Register
36-39	0x090-0x09C	-	-	-	-	Reserved
40	0x0A0	TRCVDCTLR	RW	32	Trace	ViewData Main Control Register
41	0x0A4	TRCVDSACCTLR	RW	32	Trace	ViewData Include/Exclude Single Address Comparator Control Register
42	0x0A8	TRCVDARCCTLR	RW	32	Trace	ViewData Include/Exclude Address Range Comparator Control Register
43-63	0x0AC-0x0FC	-	-	-	-	Reserved

Table 7-1 Register summary (continued)

Number	Offset	Name	Access	Width	Type	Description
64-66	0x100+*nn*[b]	TRCSEQEVRn	RW	32	Trace	Sequencer State Transition Control Register [n=0-2]
67-69	0x10C-0x114	-	-	-	-	Reserved
70	0x118	TRCSEQRSTEVR	RW	32	Trace	Sequencer Reset Control Register
71	0x11C	TRCSEQSTR	RW	32	Trace	Sequencer State Register
72	0x120	TRCEXTINSELR	RW	32	Trace	External Input Select Register
73-79	0x124-0x13C	-	-	-	-	Reserved
80-83	0x140+*nn*[b]	TRCCNTRLDVRn	RW	32	Trace	Counter Reload Value Register [n=0-3]
84-87	0x150+*nn*[b]	TRCCNTCTLRn	RW	32	Trace	Counter Control Register [n=0-3]
88-91	0x160+*nn*[b]	TRCCNTVRn	RW	32	Trace	Counter Value Register [n=0-3]
92-95	0x170-0x17C	-	-	-	-	Reserved
96	0x180	TRCIDR8	RO	32	Trace	ID Register 8
97	0x184	TRCIDR9	RO	32	Trace	ID Register 9
98	0x188	TRCIDR10	RO	32	Trace	ID Register 10
99	0x18C	TRCIDR11	RO	32	Trace	ID Register 11
100	0x190	TRCIDR12	RO	32	Trace	ID Register 12
101	0x194	TRCIDR13	RO	32	Trace	ID Register 13
102-111	0x198-0x1BC	-	-	-	-	Reserved
112	0x1C0	TRCIMSPEC0	RW	32	Trace	IMPLEMENTATION DEFINED register 0
113-119	0x1C0+*nn*[b]	TRCIMSPECn	-	32	Trace	IMPLEMENTATION DEFINED register [n=1-7]
120	0x1E0	TRCIDR0	RO	32	Trace	ID Register 0
121	0x1E4	TRCIDR1	RO	32	Trace	ID Register 1
122	0x1E8	TRCIDR2	RO	32	Trace	ID Register 2
123	0x1EC	TRCIDR3	RO	32	Trace	ID Register 3
124	0x1F0	TRCIDR4	RO	32	Trace	ID Register 4
125	0x1F4	TRCIDR5	RO	32	Trace	ID Register 5
126	0x1F8	TRCIDR6	RO	32	Trace	ID Register 6
127	0x1FC	TRCIDR7	RO	32	Trace	ID Register 7
128-129	0x200-0x204	-	-	-	-	Reserved
130-159	0x200+*nn*[b]	TRCRSCTLRn	RW	32	Trace	Resource Selection Control Register [n=2-31]
160-167	0x280+*nn*[b]	TRCSSCCRn	RW	32	Trace	Single-shot Comparator Control Register [n=0-7]

Table 7-1 Register summary (continued)

Number	Offset	Name	Access	Width	Type	Description
168-175	0x2A0+*nn*[b]	TRCSSCSRn	RW	32	Trace	Single-shot Comparator Status Register [n=0-7]
176-183	0x2C0+*nn*[b]	TRCSSPCICRn	RW	32	Trace	Single-shot PE Comparator Input Control Register [n=0-7]
184-191	0x2E0-0x2FC	-	-	-	-	Reserved
192	0x300	TRCOSLAR	WO	32	Management	OS Lock Access Register
193	0x304	TRCOSLSR	RO	32	Management	OS Lock Status Register
194-195	0x308-0x30C	-	-	-	-	Reserved
196	0x310	TRCPDCR	RW	32	Management	PowerDown Control Register
197	0x314	TRCPDSR	RO	32	Management	PowerDown Status Register
198-223	0x318-0x37C	-	-	-	-	Reserved
224-255	0x380-0x3FC	-	-	-	-	Reserved, block number 7, see Table 4-12 on page 4-164
256-287	0x400+*nn*[c]	TRCACVRn	RW	64	Trace	Address Comparator Value Register [n=0-15]
288-319	0x480+*nn*[c]	TRCACATRn	RW	64	Trace	Address Comparator Access Type Register [n=0-15]
320-351	0x500+*nn*[d]	TRCDVCVRn	RW	64	Trace	Data Value Comparator Value Register [n=0-7]
352-383	0x580+*nn*[d]	TRCDVCMRn	RW	64	Trace	Data Value Comparator Mask Register [n=0-7]
384-399	0x600+*nn*[c]	TRCCIDCVRn	RW	64	Trace	Context ID Comparator Value Register [n=0-7]
400-415	0x640+*nn*[c]	TRCVMIDCVRn	RW	64	Trace	Virtual context identifier Comparator Value Register [n=0-7]
416	0x680	TRCCIDCCTLR0	RW	32	Trace	Context ID Comparator Control Register 0
417	0x684	TRCCIDCCTLR1	RW	32	Trace	Context ID Comparator Control Register 1
418	0x688	TRCVMIDCCTLR0	RW	32	Trace	Virtual context identifier Comparator Control Register 0
419	0x68C	TRCVMIDCCTLR1	RW	32	Trace	Virtual context identifier Comparator Control Register 1
420-447	0x690-0x6FC	-	-	-	-	Reserved, block number 13, see Table 4-12 on page 4-164
448-927	0x700-0xE7C	-	-	-	-	Reserved, block numbers 14-28, see Table 4-12 on page 4-164

Table 7-1 Register summary (continued)

Number	Offset	Name	Access	Width	Type	Description
928-959	0xE80-0xEFC	-	-	-	-	Reserved for IMPLEMENTATION DEFINED integration and topology detection registers.
960	0xF00	TRCITCTRL	RW	32	Management	Integration Mode Control register
961-991	0xF04-0xF7C	-	-	-	-	Reserved, block number 30, see Table 4-12 on page 4-164
992-999	0xF80-0xF9C	-	-	-	-	Reserved, block number 31, see Table 4-12 on page 4-164
1000	0xFA0	TRCCLAIMSET	RW	32	Trace	Claim Tag Set register
1001	0xFA4	TRCCLAIMCLR	RW	32	Trace	Claim Tag Clear register
1002	0xFA8	TRCDEVAFF0	RO	32	Management	Device Affinity register 0
1003	0xFAC	TRCDEVAFF1	RO	32	Management	Device Affinity register 1
1004	0xFB0	TRCLAR	WO	32	Management	Software Lock Access Register
1005	0xFB4	TRCLSR	RO	32	Management	Software Lock Status Register
1006	0xFB8	TRCAUTHSTATUS	RO	32	Management	Authentication Status register
1007	0xFBC	TRCDEVARCH	RO	32	Management	Device Architecture register
1008-1009	0xFC0-0xFC4	-	-	-	-	Reserved, block number 31, see Table 4-12 on page 4-164
1010	0xFC8	TRCDEVID	RO	32	Management	Device ID register
1011	0xFCC	TRCDEVTYPE	RO	32	Management	Device Type register
1012	0xFD0	TRCPIDR4	RO	32	Management	Peripheral ID4 Register
1013-1015	0xFD4-0xFDC	TRCPIDR[5,6,7]	RO	32	Management	Peripheral ID5 to Peripheral ID7 Registers
1016	0xFE0	TRCPIDR0	RO	32	Management	Peripheral ID0 Register
1017	0xFE4	TRCPIDR1	RO	32	Management	Peripheral ID1 Register
1018	0xFE8	TRCPIDR2	RO	32	Management	Peripheral ID2 Register
1019	0xFEC	TRCPIDR3	RO	32	Management	Peripheral ID3 Register
1020	0xFF0	TRCCIDR0	RO	32	Management	Component ID0 Register
1021	0xFF4	TRCCIDR1	RO	32	Management	Component ID1 Register
1022	0xFF8	TRCCIDR2	RO	32	Management	Component ID2 Register
1023	0xFFC	TRCCIDR3	RO	32	Management	Component ID3 Register

a. Some implementations might restrict access to RO.
b. *nn* =0x4×*n*, where n is the register suffix number.
c. *nn* =0x8×*n*, where n is the register suffix number.
d. *nn* =0x10×*n*, where n is the register suffix number.

7.2 Access permissions

Table 7-3 on page 7-341, Table 7-4 on page 7-342, and Table 7-5 on page 7-343 show behaviors on register accesses for different trace unit states. Each table gives behavior information for a particular access mechanism. The possible trace unit states are defined in Table 7-2.

Table 7-2 Possible trace unit states for the different access methods

Access method	Trace unit state[a]			
	No debug power	No core power[b]	OS Lock locked	Non-privileged
External debugger. See Table 7-3 on page 7-341.	The tables show behaviors on accesses when the trace unit debug domain is powered down.	The tables show behaviors on accesses when both: • The trace unit core power domain is powered down or is in a retention state. • The trace unit is not in the *no debug power* state.	The table shows behaviors on accesses when all the following apply: • The trace unit is not in the *no debug power* state. • The trace unit is not in the *no core power* state. • The OS Lock is locked.	-
Memory map. See Table 7-4 on page 7-342.			-	-
System instructions. See Table 7-5 on page 7-343.	-	The table shows behaviors on accesses when the trace unit core power domain is powered down or is in a retention state.	-	The table shows behaviors on accesses when both: • The trace unit is not in the *no core power* state. • The PE is operating in a non-privileged mode, or accesses to the trace unit registers are disabled using the CPACR, NSACR, or HCPTR in the PE.

a. If the trace unit is in a state that is not covered by one of these definitions, then the behaviors on register accesses, for a particular access method, are as shown in the *otherwise* column in the appropriate table.

b. For information on the effect of the OS Double Lock, see *Effect of the OS Double Lock on page 3-95*.

The behaviors that are shown in Table 7-3 on page 7-341, Table 7-4 on page 7-342, and Table 7-5 on page 7-343 are:

Error Slave-generated error response. Writes are ignored. Reads return an UNKNOWN value. For all memory-mapped accesses, an error is returned through the memory system. For all accesses by an external debugger, an error is returned to the external debugger. For all accesses by system instructions, an Undefined Instruction exception is taken.

OK The read or write access is successful. Writes to RO locations are ignored. Reads from RES0H or WO locations return zero.

WI Writes are ignored. Reads return the register value.

IMPDEF The behavior is IMPLEMENTATION DEFINED.

SW Lock Software Lock. The TRCLSR shows the status of the Software Lock.

Table 7-3 Behaviors when using an external debugger

Register	Trace unit state			
	No debug power[a]	No core power[a]	OS Lock locked	Otherwise
All trace registers except: • TRCPRGCTLR. • TRCCLAIMCLR. • TRCCLAIMSET.	Error	Error	Error	OK[b]
TRCPRGCTLR, TRCCLAIMCLR, and TRCCLAIMSET	Error	Error	Error	OK
TRCLSR	Error	RES0H	RES0H	RES0H
TRCLAR	Error	RES0H	RES0H	RES0H
TRCPDSR and TRCPDCR	Error	OK	OK	OK
TRCOSLSR	Error	Error	OK	OK
TRCOSLAR	Error	Error	OK	OK
TRCDEVID and TRCAUTHSTATUS	Error	OK	OK	OK
TRCITCTRL	Error	IMPDEF	IMPDEF	OK
Reserved for IMPLEMENTATION DEFINED integration and topology detection registers	Error	IMPDEF	IMPDEF	OK
Other management	Error	OK	OK	OK
Unimplemented trace	Error	Error	Error or RES0H	RES0H
Reserved trace	Error	Error	Error or RES0H	RES0H
Reserved management	Error	RES0H	RES0H	RES0H

a. In a single-power trace unit implementation, the no debug power and no core power states are not possible. This is because the trace unit core and debug domains are either both powered or both unpowered. In this case, if the trace unit is in an unpowered state, all accesses must result in the behaviors shown in the no debug power state.

b. When the trace unit is enabled or not idle, as defined in *Trace unit behavior when the trace unit is enabled on page 3-100*, these registers might be WI.

Table 7-4 Behaviors when using memory-mapped access

Register	Trace unit state		
	No debug power[a]	No core power	Otherwise
All trace registers except: • TRCPRGCTLR. • TRCCLAIMCLR. • TRCCLAIMSET.	Error	Error	OK[bc]
TRCPRGCTLR, TRCCLAIMCLR, and TRCCLAIMSET	Error	Error	OK[c]
TRCLSR	Error	OK	OK
TRCLAR	Error	OK	OK
TRCPDCR	Error	OK[c]	OK[c]
TRCPDSR	Error	OK[d]	OK[d]
TRCOSLSR	Error	Error	OK
TRCOSLAR	Error	Error	OK[c]
TRCDEVID and TRCAUTHSTATUS	Error	OK	OK
TRCITCTRL	Error	IMPDEF[c]	OK[c]
Reserved for IMPLEMENTATION DEFINED integration and topology detection registers	Error	IMPDEF[c]	OK[c]
Other management	Error	OK[c]	OK[c]
Reserved trace and unimplemented trace	Error	Error	RES0H
Reserved management and unimplemented management	Error	RES0H	RES0H

a. In a single-power trace unit implementation, the no debug power and no core power states are not possible. This is because the trace unit core and debug domains are either both powered or both unpowered. In this case, if the trace unit is in an unpowered state, all accesses must result in the behaviors shown in the no debug power state.

b. When the trace unit is enabled or not idle, as defined in *Trace unit behavior when the trace unit is enabled on page 3-100*, these registers might be WI.

c. These registers are WI when the Software Lock is implemented and locked. This is to prevent on-chip software, when it is being debugged, from accidentally disabling or reprogramming the trace unit. The TRCLSR shows the status of the Software Lock.

d. Normally, on reading the TRCPDSR, if the TRCPDSR.STICKYPD bit is set to 1 it is cleared to 0 if the trace unit core power domain is powered. However, when the Software Lock is implemented and locked, reads from the TRCPDSR do not clear TRCPDSR.STICKYPD to 0.

Note

When using memory-mapped access, whenever the Software Lock is implemented and locked:

- The TRCLAR is the only register that can be written to. Write accesses to all other registers are ignored.
- Read accesses to all registers are unaffected, except reads from the TRCPDSR which do not clear TRCPDSR.STICKYPD.

Table 7-5 Behaviors when using system instructions

Register table	Trace unit state		
	No core power[a]	Non-privileged	Otherwise[b]
All trace registers except: • TRCPRGCTLR. • TRCCLAIMCLR. • TRCCLAIMSET.	Error	Error	OK[cd]
TRCPRGCTLR, TRCCLAIMCLR, and TRCCLAIMSET	Error	Error	OK
TRCLSR	Error	Error	Error
TRCLAR	Error	Error	Error
TRCPDSR and TRCPDCR	Error	Error	Error
TRCOSLSR	Error	Error	OK[d]
TRCOSLAR	Error	Error	OK[e]
TRCDEVID, TRCAUTHSTATUS, and TRCDEVARCH	Error	Error	OK[d]
TRCITCTRL	Error	Error	Error
Reserved for IMPLEMENTATION DEFINED integration and topology detection registers	Error	Error	Error
Other management	Error	Error	Error
Reserved trace and unimplemented trace	Error	Error	Error
Reserved management and unimplemented management	Error	Error	Error

a. In a single-power trace unit implementation, the no core power state is the same as the trace unit being unpowered.

b. System instructions cannot access any registers that are in the trace unit debug power domain. That is, system instructions cannot access any trace unit management registers. See *Trace unit power domains on page 3-91*.

c. When the trace unit is enabled or not idle, as defined in *Trace unit behavior when the trace unit is enabled on page 3-100*, these registers might be WI.

d. Writes to read-only registers are considered to be accesses to Reserved registers and result in an Error. The following read-only registers are accessible by system instructions TRCIDRn, TRCAUTHSTATUS, TRCDEVID, TRCDEVARCH, TRCSTATR, and TRCOSLSR.

e. Reads of the write-only register TRCOSLAR are considered to be accesses to Reserved registers and result in an Error.

7.2.1 Trace unit behavior on accesses to reserved trace unit registers and fields

This section defines trace unit behavior on accesses to reserved trace unit registers, or to reserved fields within trace unit registers.

Table 7-6 Behavior on accesses to reserved trace unit registers and fields

Access to:	Access method	
	Memory-mapped or external debugger	System instructions
Reserved registers	RES0H[a]	UNDEFINED[b]
Unimplemented registers	RES0H[c]	UNDEFINED[b]
Reserved fields in registers	RES0 or RES1	RES0 or RES1
Unimplemented fields in registers	RES0 or RES1	RES0 or RES1
Unimplemented bits in implemented fields	RES0	RES0

a. External debugger accesses to Reserved Trace registers when the OS Lock is set might behave as Error.
b. If the behavior is UNDEFINED for a system instruction, the instruction takes an Undefined Instruction exception.
c. External debugger accesses to Unimplemented Trace registers when the OS Lock is set might behave as Error.

Reads of write-only registers are considered accesses to Reserved registers. Writes to read-only registers are considered accesses to Reserved registers.

7.3 ETMv4 registers descriptions, in register name order

This section describes all the ETMv4 registers. Registers are shown in alphabetical order by register name.

7.3.1 TRCACATRn, Address Comparator Access Type Registers, n=0-15

The TRCACATRn characteristics are:

Purpose Defines the type of access for the corresponding TRCACVRn Register. This register selects the context type, Exception levels, alignment, and masking that is applied by the address comparator, and defines how the address comparator behaves when it is one half of an address range comparator.

Usage constraints

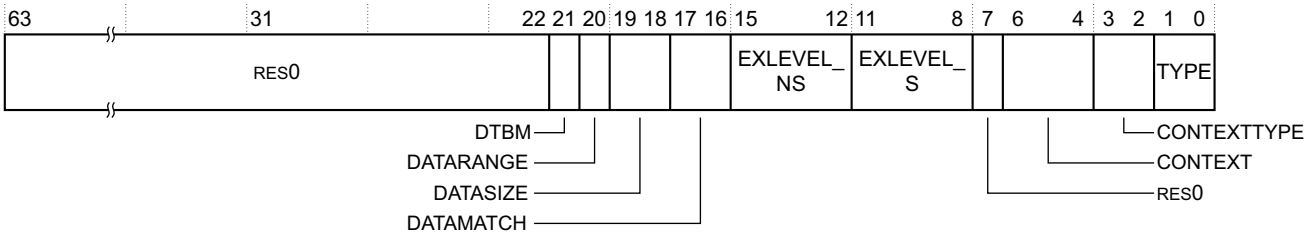
- Might ignore writes when the trace unit is enabled or not idle.
- CONSTRAINED UNPREDICTABLE behavior of a comparator resource occurs if:
  - TYPE==0 and DATAMATCH==`0b01`, `0b10`, or `0b11`.
  - DATAMATCH==`0b01`, `0b10`, or `0b11` and software programs an address comparator to control ViewData.

  In these scenarios, the comparator might match unexpectedly or might not match.
- If software uses two single address comparators as an address range comparator it must program the corresponding TRCACATRn with identical values in the following fields:
  - TYPE.
  - CONTEXTTYPE.
  - CONTEXT.
  - EXLEVEL_S.
  - EXLEVEL_NS.
  - DTBM.

Configurations The number, n, of TRCACATRn is IMPLEMENTATION DEFINED and is set by 2×TRCIDR4.NUMACPAIRS.

Attributes A 64-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCACATRn bit assignments are:



Bits[63:22] RES0.

DTBM, bit[21] Controls whether data address comparisons use the data address [63:56] bits:

0 The trace unit ignores the data address [63:56] bits for data address comparisons.

1 The trace unit uses the data address [63:56] bits for data address comparisons.

Supported only if TRCIDR2.DASIZE indicates that the data address size is 64 bits, and TRCIDR2.SUPPDAC indicates that data address comparisons are implemented. Otherwise this bit is RES0.

DATARANGE, bit[20]

Controls whether a data value comparison uses the single address comparator or the address range comparator:

0 The trace unit uses the single address comparator for data value comparisons. The behavior of the:

- Address range comparator is CONSTRAINED UNPREDICTABLE, as *Rules when a data value comparator is programmed for use with a single address comparator on page 4-155* describes.
- Other single address comparator in the pair is not affected.

1 The trace unit uses the address range comparator for data value comparisons. The behavior of the single address comparators in this pair is CONSTRAINED UNPREDICTABLE, as *Rules when a data value comparator is programmed for use with a single address comparator on page 4-155* describes.

The trace unit ignores this field when DATAMATCH==`0b00`.

Supported only if the corresponding data value comparator is supported, otherwise this bit is RES0.

DATASIZE, bits[19:18]

Controls the width of the data value comparison:

`0b00` Byte.

`0b01` Halfword.

`0b10` Word.

`0b11` Doubleword.

Supported only if the corresponding data value comparator is supported, otherwise this field is RES0.

The doubleword width is supported only if TRCIDR2.DVSIZE indicates that 64-bit values are supported. If 64-bit values are not supported, `0b11` is reserved.

DATAMATCH, bits[17:16]

Controls how the trace unit performs a data value comparison:

`0b00` The trace unit does not perform a data value comparison.

`0b01` The trace unit performs a data value comparison. If the data value comparator matches and the address comparator matches, the trace unit signals a match.

`0b10` Reserved

`0b11` The trace unit performs a data value comparison. If the data value comparator does not match and the address comparator matches, the trace unit signals a match.

Supported only if the corresponding data value comparator is supported, otherwise this field is RES0.

EXLEVEL_NS, bits[15:12]

In Non-secure state, each bit controls whether a comparison can occur for the corresponding Exception level:

0 The trace unit can perform a comparison, in Non-secure state, for Exception level *n*.

For Armv8-M PEs, the trace unit does not distinguish between the Secure and Non-secure states, and treats both states as Secure state. This field always reads as zero, indicating no Non-secure states are supported.

1 The trace unit does not perform a comparison, in Non-secure state, for Exception level *n*.

The Exception levels are:

Bit[12] Exception level 0.

Bit[13] Exception level 1.

Bit[14] Exception level 2.

Bit[15] RES0. EXLEVEL_NS[3] is never implemented.

The content of **EXLEVEL_NS** is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR3.EXLEVEL_NS. Unimplemented bits are RES0.

EXLEVEL_S, bits[11:8]

In Secure state, each bit controls whether a comparison can occur for the corresponding Exception level:

0 The trace unit can perform a comparison, in Secure state, for Exception level *n*.

1 The trace unit does not perform a comparison, in Secure state, for Exception level *n*.

The Exception levels are:

Bit[8] Exception level 0.

Bit[9] Exception level 1.

Bit[10] Exception level 2.

Bit[11] Exception level 3.

The content of **EXLEVEL_S** is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR3.EXLEVEL_S. Unimplemented bits are RES0.

Bit[7] RES0.

CONTEXT, bits[6:4]

When TRCIDR4.NUMCIDC > `0b0001` or TRCIDR4.NUMVMIDC > `0b0001` these bits selects a Context ID comparator or Virtual context identifier comparator:

`0b000` Comparator 0.

`0b001` Comparator 1.

`0b010` Comparator 2.

. .

. .

. .

`0b111` Comparator 7.

The implemented width of this field is determined by the number of Context ID comparators and Virtual context identifier comparators, as defined by TRCIDR4.NUMCIDC and TRCIDR4.NUMVMIDC.

If NUMCIDC ≤ 1 and NUMVMIDC ≤ 1, then bits[6:4] are RES0.

If NUMCIDC ≤ 2 and NUMVMIDC ≤ 2, then bits[6:5] are RES0.

If NUMCIDC ≤ 4 and NUMVMIDC ≤ 4, then bit[6] is RES0.

When TRCIDR4.NUMCIDC ≤ `0b0001` and TRCIDR4.NUMVMIDC ≤ `0b0001` these bits are RES0.

Note

A Context ID comparator compares with the current Context ID value regardless of the current Exception level at which the PE is executing. Similarly, a Virtual context identifier comparator compares the current Virtual context identifier value regardless of the current Exception level at which the PE is executing. This behavior differs from the PE debug logic where, for example, Virtual context identifier comparisons do not occur when the PE is in EL2 or in Secure state. The trace unit can be programmed to prevent comparisons in some Exception levels using the EXLEVEL_S and EXLEVEL_NS fields.

CONTEXTTYPE, bits[3:2]

When TRCIDR4.NUMCIDC≥0b0001 or TRCIDR4.NUMVMIDC≥0b0001 this field controls whether the trace unit performs a Context ID comparison, a *Virtual context identifier* comparison, or both comparisons:

- 0b00 The trace unit does not perform a Context ID comparison.
- 0b01 The trace unit performs a Context ID comparison using the Context ID comparator that the CONTEXT field specifies. If both the Context ID comparator and the address comparator match, the trace unit signals a match.
- 0b10 The trace unit performs a Virtual context identifier comparison using the Virtual context identifier comparator that the CONTEXT field specifies. If both the Virtual context identifier comparator and the address comparator match, the trace unit signals a match.
- 0b11 The trace unit performs a Context ID comparison and a Virtual context identifier comparison using the comparators that the CONTEXT field specifies. If the Context ID comparator, the Virtual context identifier comparator and address comparator matches, the trace unit signals a match.

If TRCIDR4.NUMVMIDC==0 then bit[3] is RES0 and bit[2] controls whether the trace unit performs a Context ID comparison:

- **0** The trace unit does not perform a Context ID comparison.
- **1** The trace unit performs a Context ID comparison using the Context ID comparator that the CONTEXT field specifies. If both the Context ID comparator and the address comparator match, the trace unit signals a match.

If TRCIDR4.NUMCIDC==0 and TRCIDR4.NUMVMIDC==0 these bits are RES0.

TYPE, bits[1:0] Controls what type of comparison the trace unit performs:

- 0b00 Instruction address.
- 0b01 Data load address.
- 0b10 Data store address.
- 0b11 Data load address or data store address.

If TRCIDR4.SUPPDAC does not indicate that data address comparisons are implemented, then this field is RES0. This means that any comparison performed by this address comparator is an instruction address comparison.

7.3.2 TRCACVRn, Address Comparator Value Registers, n=0-15

The TRCACVRn characteristics are:

Purpose Contains an address value.

Usage constraints Might ignore writes when the trace unit is disabled.

Configurations

- The number, n, of TRCACVRs is IMPLEMENTATION DEFINED and is set by 2×TRCIDR4.NUMACPAIRS.
- The register width is IMPLEMENTATION DEFINED and is the larger of TRCIDR2.IASIZE and TRCIDR2.DASIZE.

Attributes A 64-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCACVRn bit assignments are:

ADDRESS, bits[63:0] Address value.

The address comparators can support implementations that use multiple address widths. When the trace unit compares the ADDRESS field with an address that has a width less than this field, then the address must be zero-extended to the ADDRESS field width. The trace unit then compares all implemented bits. For example, in a system that supports both 32-bit and 64-bit addresses, when the PE is in AArch32 state the comparator must zero-extend the 32-bit address and compare against the full 64-bits that are stored in the TRCACVRn. This requires that the trace analyzer always programs all implemented bits of the TRCACVRn.

In an Armv8-A PE, if TRCIDR2.IASIZE indicates an instruction address size of 64 bits and the trace unit only supports instruction address comparisons:

- The result of writing a value other than all zeros or all ones to ADDRESS at bits[63:P] is an UNKNOWN value, where P is defined as the virtual address size supported by the PE.
- The result of writing a value of all zeros or all ones to ADDRESS at bits[63:P] is the written value, and a read of the register returns the written value.

Note

ETMv4.2 introduces support for virtual addresses larger than 48 bits.

7.3.3 TRCAUTHSTATUS, Authentication Status register

The TRCAUTHSTATUS characteristics are:

Purpose

Returns the level of tracing that the trace unit can support.

The CoreSight authentication signals control the level of tracing that the PE and trace unit can support.

See the *CoreSight Architecture Specification* for information about the authentication signals, apart from the following architectural changes which are introduced in ETMv4.2 and replicate the new EL2 behavior and functionality to control debug in EL2 introduced by Armv8-R.

For a trace unit for an Armv8-R PE that implements EL2, the authentication interface provides two signals:

- HIDEN, which enables invasive debug in EL2.
- HNIDEN, which enables non-invasive debug in EL2.

When HNIDEN and HIDEN are both LOW, tracing is prohibited in EL2. These signals are optional, and if not implemented then debug at EL2 is controlled by the Non-secure portion of the authentication interface, DBGEN and NIDEN.

Invasive debug in EL2 is only permitted if invasive Non-secure debug is also permitted.

Non-invasive debug in EL2 is only permitted if non-invasive Non-secure debug is also permitted.

Note

- Arm considers the trace unit a non-invasive debug component.

Usage constraints

There are no usage constraints.

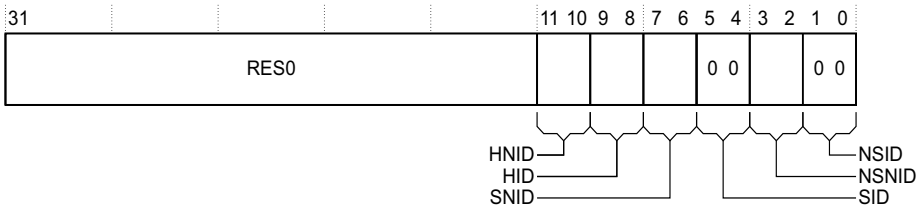
Configurations

- Available in all implementations.
- It is IMPLEMENTATION DEFINED whether the register contains the NSNID field.

Attributes

A 32-bit RO management register. The register is set to an IMPLEMENTATION DEFINED value on an external trace reset. See also *Register summary on page 7-336*.

The TRCAUTHSTATUS bit assignments are:



Bits[31:12] RES0

HNID, bits[11:10] These bits indicate whether a non-invasive debug enable for EL2 is implemented, and whether debug at EL2 is permitted:

- `0b00` Separate enable for EL2 is not implemented or EL2 is not implemented. See TRCAUTHSTATUS.NSNID.
- `0b10` Separate enable for EL2 is implemented and is disabled.
- `0b11` Separate enable for EL2 is implemented and is enabled.

All other values are reserved.

HID, bits[9:8] These bits indicate whether an invasive debug enable for EL2 is implemented, and whether debug at EL2 is permitted:

- `0b00` The trace unit does not support invasive debug for EL2.

All other values are reserved.

SNID, bits[7:6] Indicates whether the system enables the trace unit to support Secure non-invasive debug:

- `0b00` The trace unit does not implement support for Secure non-invasive debug.
- `0b01` Reserved.
- `0b10` Secure non-invasive debug is disabled.
- `0b11` Secure non-invasive debug is enabled.

For Armv7-A PEs that implement the Security Extensions, TRCAUTHSTATUS.SNID indicates the permitted level of debug in Secure state.

For Armv7-A PEs that do not implement the Security Extensions, TRCAUTHSTATUS.SNID indicates the permitted level of debug.

For Armv8-A PEs that implement EL3, TRCAUTHSTATUS.SNID indicates the permitted level of debug in Secure state.

For Armv8-A PEs that do not implement EL3, if the PE is executing in Secure state, TRCAUTHSTATUS.SNID indicates the permitted debug level, otherwise TRCAUTHSTATUS.SNID is `0b00`.

For Armv8.4-A and above PEs that implement Armv8.4-Trace, TRCAUTHSTATUS.SNID holds the same value as DBGAUTHSTATUS_EL1.SNID.

For Armv8-R PEs, TRCAUTHSTATUS.SNID has the value `0b00`.

For Armv8-M PEs, if Secure non-invasive debug is disabled, then tracing is prohibited when the PE is executing in Secure state. Secure state offers the ability to override SPIDEN and SPNIDEN, using DAUTHCTRL, and this overridden value must be used by the trace unit.

SID, bits[5:4] Indicates whether the trace unit supports Secure invasive debug:

- `0b00` The trace unit does not support Secure invasive debug.

All other values are reserved.

NSNID, bits[3:2] These bits indicate whether the system enables the trace unit to support Non-secure non-invasive debug:

`0b00` The trace unit does not implement support for Non-secure non-invasive debug.

`0b01` Reserved.

`0b10` Non-secure non-invasive debug is disabled.

`0b11` Non-secure non-invasive debug is enabled.

For Armv7-A PEs that implement the Security Extensions, TRCAUTHSTATUS.NSNID indicates the permitted level of debug in Non-secure state.

For Armv7-A PEs that do not implement the Security Extensions, TRCAUTHSTATUS.NSNID is always `0b00` and the permitted level of debug is indicated in TRCAUTHSTATUS.SNID.

For Armv8-A PEs that implement EL3, TRCAUTHSTATUS.NSNID indicates the permitted level of debug in Non-secure state.

For Armv8-A PEs that do not implement EL3, if the PE is executing in Non-secure state then TRCAUTHSTATUS.NSNID indicates the permitted level of debug, otherwise TRCAUTHSTATUS.NSNID is `0b00`.

For Armv8.4-A and above PEs that implement Armv8.4-Trace, TRCAUTHSTATUS.NSNID holds the same value as DBGAUTHSTATUS_EL1.NSNID.

For Armv8-R PEs, TRCAUTHSTATUS.NSNID indicates the permitted level of debug.

When the PE implements Armv8.4-Trace, non-invasive debug is always permitted.

NSID, bits[1:0] Indicates whether the trace unit supports Non-secure invasive debug:

`0b00` The trace unit does not support Non-secure invasive debug.

All other values are reserved.

For implementations that support multiple access mechanisms, different access mechanisms can return different values for reads of TRCAUTHSTATUS if the authentication signals have changed and that change has not yet been synchronized by a Context synchronization event. This scenario can happen if, for example, the external debugger view is implemented separately from the system instruction view to allow for separate power domains, and so observes changes on the signals differently.

7.3.4 TRCAUXCTLR, Auxiliary Control Register

The TRCAUXCTLR characteristics are:

Purpose The function of this register is IMPLEMENTATION DEFINED.

Usage constraints There are no usage constraints.

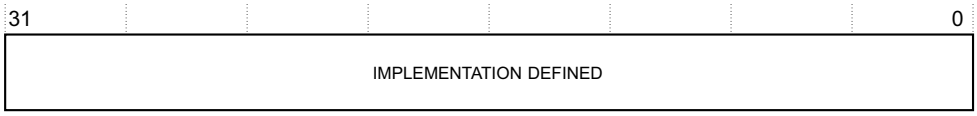
Note

If trace debug tools set the value of this register to nonzero then it might cause the behavior of a trace unit to contradict this architecture specification. See the documentation of the specific implementation for information about the IMPLEMENTATION DEFINED support for this register.

Configurations This register is always implemented.

Attributes A 32-bit RW trace register. This register is set to zero on a trace unit reset. Resetting this register to zero ensures that none of the IMPLEMENTATION DEFINED features are enabled by default, and that the trace unit resets to a known state (an ETMv4 trace unit with no IMPLEMENTATION DEFINED features enabled). See also *Register summary on page 7-336*.

The TRCAUXCTLR bit assignments are:



Bits[31:0] IMPLEMENTATION DEFINED.

7.3.5 TRCBBCTLR, Branch Broadcast Control Register

The TRCBBCTLR characteristics are:

Purpose Controls which regions in the memory map are enabled to use branch broadcasting.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- Must be programmed if TRCCONFIGR.BB == 1.
- CONSTRAINED UNPREDICTABLE tracing occurs if software writes to this register and selects an address range comparator pair that is not programmed to be an instruction address comparator.

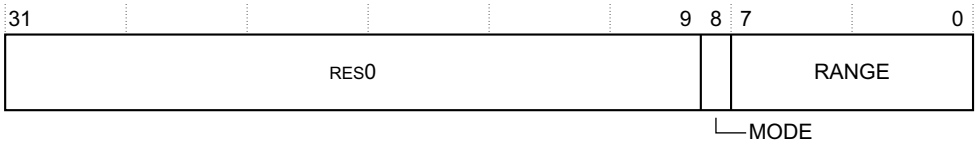
Configurations Implemented when a trace unit implements both branch broadcasting and address comparators, that is, when TRCIDR0.TRCBB == 1 and TRCIDR4.NUMACPAIRS > 0.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

In a region where branch broadcasting is active:

- A trace unit must trace the branch target for each branch instruction that is taken, even if the branch is mispredicted.
- If the return stack is enabled, that is TRCCONFIGR.RS==1, then the branch broadcast mode has a higher priority.

The TRCBBCTLR bit assignments are:



Bits[31:9] RES0.

MODE, bit[8] Mode bit:

0 Exclude mode. Branch broadcasting is not enabled for branch instructions in the address ranges that RANGE defines.
If RANGE==0 then branch broadcasting is enabled for the entire memory map.

1 Include mode. Branch broadcasting is enabled for branch instructions in the address ranges that RANGE defines.
If RANGE==0 then the behavior of the trace unit is CONSTRAINED UNPREDICTABLE. That is, the trace unit might or might not consider any instructions to be in a branch broadcast region.

RANGE, bits[7:0] Address range field. Selects which address range comparator pairs are in use with branch broadcasting. Each bit represents an address range comparator pair, so bit[*n*] controls the selection of address range comparator pair *n*. If bit[*n*] is:

0 The address range that address range comparator pair *n* defines, is not selected.

1 The address range that address range comparator pair *n* defines, is selected.

The width of the field is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR4.NUMACPAIRS. If TRCIDR4.NUMACPAIRS is < 8 then bits[7:TRCIDR4.NUMACPAIRS] is RES0.

7.3.6 TRCCCCTLR, Cycle Count Control Register

The TRCCCCTLR characteristics are:

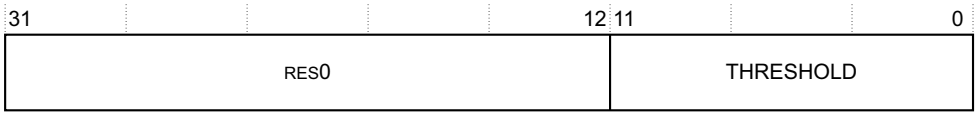
Purpose Sets the threshold value for cycle counting.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- Must be programmed if TRCCONFIGR.CCI==1.

Configurations Implemented when a trace unit implements cycle counting, that is, when TRCIDR0.TRCCCI==1.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCCCTLR bit assignments are:



Bits[31:12] RES0.

THRESHOLD, bits[11:0]

Sets the threshold value for instruction trace cycle counting.

The minimum threshold value that can be programmed into THRESHOLD is given in TRCIDR3.CCITMIN. If the THRESHOLD value is smaller than the value in TRCIDR3.CCITMIN then the behavior is CONSTRAINED UNPREDICTABLE. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

Writing a value of zero when TRCCONFIGR.CCI is set to enable instruction trace cycle counting, results in CONSTRAINED UNPREDICTABLE behavior. That is, cycle counts might or might not be included in the trace and the cycle count threshold is not known.

7.3.7 TRCCIDCCTLR0, Context ID Comparator Control Register 0

The TRCCIDCCTLR0 characteristics are:

Purpose Contains Context ID mask values for the TRCCIDCVRn registers, where n=0-3.

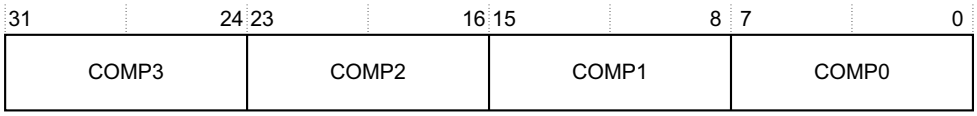
Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- If software uses the TRCCIDCVRn registers, where n=0-3, then it must program this register.
- If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVRn to `0x00`.
- If any bit is `0b1` and the relevant byte in TRCCIDCVRn is not `0x00`, the behavior of the Context ID comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Configurations
- Only implemented when TRCIDR4.NUMCIDC > 0, indicating that at least one Context ID comparator is implemented, and TRCIDR2.CIDSIZE > 0, indicating that the Context ID is greater than 0 bits in length.
- The number of COMP<n> fields that the register contains is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMCIDC.

- The width of a COMP<n> field is IMPLEMENTATION DEFINED and is set by TRCIDR2.CIDSIZE. Unimplemented bits are RES0.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCIDCCTLR0 bit assignments are:



COMP3, bits[31:24] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==3. Each bit in this field corresponds to a byte in TRCCIDCVR3. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR3 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR3 when it performs the Context ID comparison.

For example, if bit[30]==1 then the trace unit ignores TRCCIDCVR3.VALUE[55:48].

Supported only if TRCIDR4.NUMCIDC ≥ `0b0100`, otherwise bits[31:24] are RES0.

COMP2, bits[23:16] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==2. Each bit in this field corresponds to a byte in TRCCIDCVR2. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR2 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR2 when it performs the Context ID comparison.

For example, if bit[21]==1 then the trace unit ignores TRCCIDCVR2.VALUE[47:40].

Supported only if TRCIDR4.NUMCIDC ≥ `0b0011`, otherwise bits[23:16] are RES0.

COMP1, bits[15:8] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==1. Each bit in this field corresponds to a byte in TRCCIDCVR1. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR1 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR1 when it performs the Context ID comparison.

For example, if bit[12]==1 then the trace unit ignores TRCCIDCVR1.VALUE[39:32].

Supported only if TRCIDR4.NUMCIDC ≥ `0b0010`, otherwise bits[15:8] are RES0.

COMP0, bits[7:0] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==0. Each bit in this field corresponds to a byte in TRCCIDCVR0. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR0 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR0 when it performs the Context ID comparison.

For example, if bit[3]==1 then the trace unit ignores TRCCIDCVR0.VALUE[31:24].

Supported only if TRCIDR4.NUMCIDC ≥ `0b0001`, otherwise bits[7:0] are RES0.

7.3.8 TRCCIDCCTLR1, Context ID Comparator Control Register 1

The TRCCIDCCTLR1 characteristics are:

Purpose Contains Context ID mask values for the TRCCIDCVRn registers, where n=4-7.

Usage constraints

- Might ignore writes when the trace unit is enabled or not idle.

- If software uses the TRCCIDCVRn registers, where n=4-7, then it must program this register.
- If software sets a mask bit to 1 then it must program the relevant byte in TRCCIDCVRn to `0x00`.
- If any bit is `0b1` and the relevant byte in TRCCIDCVRn is not `0x00`, the behavior of the Context ID comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

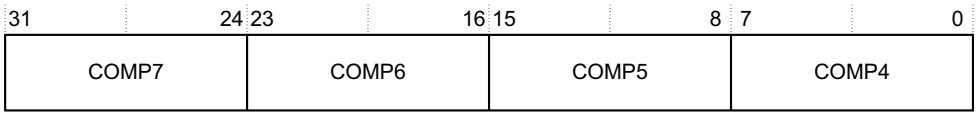
Configurations

- Only implemented when TRCIDR4.NUMCIDC > 4, indicating that more than 4 Context ID comparators are implemented, and TRCIDR2.CIDSIZE > 0, indicating that the Context ID is greater than 0 bits in length.
- The number of COMP<n> fields that the register contains is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMCIDC–4.
- The width of a COMP<n> field is IMPLEMENTATION DEFINED and is set by TRCIDR2.CIDSIZE. Unimplemented bits are RES0.

Attributes

A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCIDCCTLR1 bit assignments are:



COMP7, bits[31:24] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==7. Each bit in this field corresponds to a byte in TRCCIDCVR7. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR7 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR7 when it performs the Context ID comparison.

For example, if bit[30]==1 then the trace unit ignores TRCCIDCVR7.VALUE[55:48].

Supported only if TRCIDR4.NUMCIDC==`0b1000`, otherwise bits[31:24] are RES0.

COMP6, bits[23:16] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==6. Each bit in this field corresponds to a byte in TRCCIDCVR6. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR6 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR6 when it performs the Context ID comparison.

For example, if bit[21]==1 then the trace unit ignores TRCCIDCVR6.VALUE[47:40].

Supported only if TRCIDR4.NUMCIDC≥`0b0111`, otherwise bits[23:16] are RES0.

COMP5, bits[15:8] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==5. Each bit in this field corresponds to a byte in TRCCIDCVR5. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR5 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR5 when it performs the Context ID comparison.

For example, if bit[12]==1 then the trace unit ignores TRCCIDCVR5.VALUE[39:32].

Supported only if TRCIDR4.NUMCIDC≥`0b0110`, otherwise bits[15:8] are RES0.

COMP4, bits[7:0] Controls the mask value that the trace unit applies to TRCCIDCVRn, where n==4. Each bit in this field corresponds to a byte in TRCCIDCVR4. When a bit is:

0 The trace unit includes the relevant byte in TRCCIDCVR4 when it performs the Context ID comparison.

1 The trace unit ignores the relevant byte in TRCCIDCVR4 when it performs the Context ID comparison.

For example, if bit[3]==1 then the trace unit ignores TRCCIDCVR4.VALUE[31:24].

Supported only if TRCIDR4.NUMCIDC≥0b0101, otherwise bits[7:0] are RES0.

7.3.9 TRCCIDCVRn, Context ID Comparator Value Registers, n=0-7

The TRCCIDCVRn characteristics are:

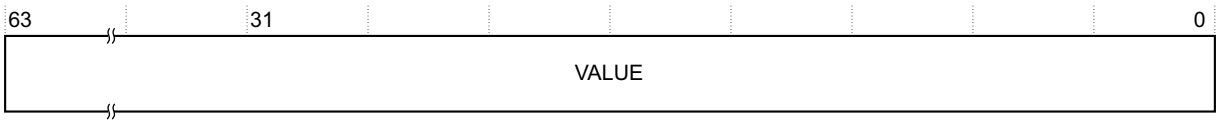
Purpose Contains a Context ID value.

Usage constraints Might ignore writes when the trace unit is enabled or not idle.

Configurations The number, n, of TRCCIDCVRs is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMCIDC.

Attributes A 64-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCIDCVRn bit assignments are:



VALUE, bits[63:0] Context ID value. The implemented width of this field is IMPLEMENTATION DEFINED and is set by TRCIDR2.CIDSIZE. Unimplemented bits are RES0.

After a PE reset, the trace unit assumes that the Context ID is zero until the PE updates the Context ID.

7.3.10 TRCCIDR0, Component ID0 Register

The TRCCIDR0 characteristics are:

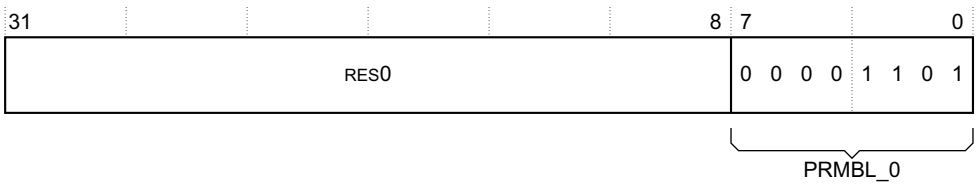
Purpose Returns byte 0 of the CoreSight preamble information.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register that returns a value of 0x0D. See also *Register summary on page 7-336*.

The TRCCIDR0 bit assignments are:



Bits[31:8] RES0.

PRMBL_0, bits[7:0] Returns `0x0D`. The other TRCCIDR registers provide the remaining bits of the preamble.

7.3.11 TRCCIDR1, Component ID1 Register

The TRCCIDR1 characteristics are:

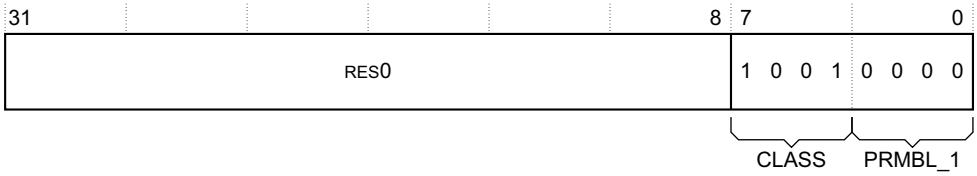
Purpose Returns byte 1 of the CoreSight preamble information.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register that returns a value of `0x90`. See also *Register summary on page 7-336*.

The TRCCIDR1 bit assignments are:



Bits[31:8] RES0.

CLASS, bits[7:4] Returns `0x9`, to indicate that the trace unit is a debug component, with CoreSight architecture compliant management registers.

PRMBL_1, bits[3:0] Returns `0x0`. The other TRCCIDR registers provide the remaining bits of the preamble.

7.3.12 TRCCIDR2, Component ID2 Register

The TRCCIDR2 characteristics are:

Purpose Returns byte 2 of the CoreSight preamble information.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register that returns a value of `0x05`. See also *Register summary on page 7-336*.

The TRCCIDR2 bit assignments are:

Bits[31:8] RES0.

PRMBL_2, bits[7:0] Returns `0x05`. The other TRCCIDR registers provide the remaining bits of the preamble.

7.3.13 TRCCIDR3, Component ID3 Register

The TRCCIDR3 characteristics are:

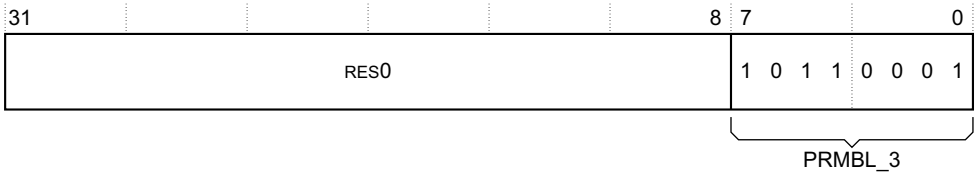
Purpose Returns byte 3 of the CoreSight preamble information.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register that returns a value of `0xB1`. See also *Register summary on page 7-336*.

The TRCCIDR3 bit assignments are:



Bits[31:8] RES0.

PRMBL_3, bits[7:0] Returns `0xB1`. The other TRCCIDR registers provide the remaining bits of the preamble.

7.3.14 TRCCLAIMCLR, Claim Tag Clear register

The TRCCLAIMCLR characteristics are:

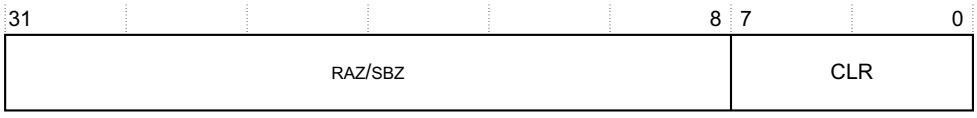
Purpose Software can use this to:
- Clear bits in the claim tag to 0.
- Read the value of the claim tag.

Usage constraints The ETMv4 architecture does not define any functionality for the CLR bits. The CLR bits do not affect the functionality of the trace unit. Software can use the claim tag to control whether the application software or the external debug agent is given access to the trace unit.

Configurations
- Available in all implementations.
- The implemented width of the claim tag, or CLR field, is IMPLEMENTATION DEFINED. The ETMv4 architecture supports a maximum claim tag width of 8 bits. Arm recommends that implementations support a minimum of four claim tag bits, that is, CLR[3:0]. Unimplemented bits within the field are RAZ/WI.

Attributes A 32-bit RW trace register. This register is set to zero on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCLAIMCLR bit assignments are:



Bits[31:8] RAZ/SBZ.

CLR, bits[7:0] When a write to one of these bits occurs, with the value:

0 The register ignores the write.

1 If the bit is supported then it is set to 0. A single write operation can set multiple bits to 0.

A read returns the value of the claim tag.

7.3.15 TRCCLAIMSET, Claim Tag Set register

The TRCCLAIMSET characteristics are:

Purpose Software can use this to:

- Set bits in the claim tag to 1.
- Discover the number of bits that the claim tag supports.

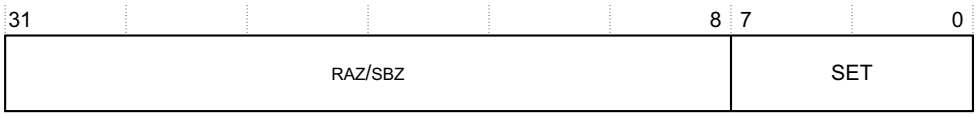
Usage constraints The ETMv4 architecture does not define any functionality for the SET bits. The SET bits do not affect the functionality of the trace unit. Software can use the claim tag to control whether the application software or the external debug agent is given access to the trace unit.

Configurations

- Available in all implementations.
- The width of the claim tag, or SET field, is IMPLEMENTATION DEFINED. The ETMv4 architecture supports a maximum claim tag width of 8 bits. This register can be read to determine the number of claim tag bits that are supported by this implementation. Arm recommends that implementations support a minimum of four claim tag bits, that is, SET[3:0].

Attributes A 32-bit RW trace register. This register is set to an IMPLEMENTATION DEFINED value on trace unit reset. See also *Register summary on page 7-336*.

The TRCCLAIMSET bit assignments are:



Bits[31:8] RAZ/SBZ.

SET, bits[7:0] When a write to one of these bits occurs, with the value:

0 The register ignores the write.

1 If the bit is supported then it is set to 1. A single write operation can set multiple bits to 1.

When a read occurs, the implemented bits in the SET field are RAO and therefore the value the register returns indicates how many SET bits are supported. Any unimplemented bits in the SET field are RAZ. A debug agent can read this register to discover the width of the claim tag.

Software must use the TRCCLAIMCLR register to:

- Read the values of the claim tag.
- Clear a claim tag bit to 0.

7.3.16 TRCCNTCTLRn, Counter Control Registers, n=0-3

The TRCCNTCTLRn characteristics are:

Purpose Controls the operation of counter <n>.

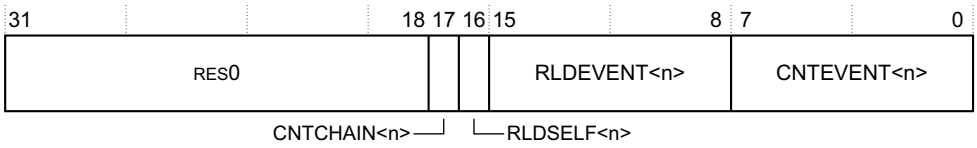
Usage constraints Might ignore writes when the trace unit is enabled or not idle.

Configurations The TRCIDR5.NUMCNTR field sets the value of n and therefore controls how many TRCCNTCTLRs are implemented.

TRCCNTCTLR0 is not implemented, if the reduced function counter is implemented. The TRCIDR5.REDFUNCNTR bit defines whether an implementation supports a reduced function counter.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCNTCTLRn bit assignments are:



Bits[31:18] RES0.

CNTCHAIN<n>, bit[17]

For TRCCNTCTLR3 and TRCCNTCTLR1, this bit controls whether counter <n> decrements when a reload event occurs for counter <n–1>:

0 Counter <n> does not decrement when a reload event for counter <n–1> occurs.

1 Counter <n> decrements when a reload event for counter <n–1> occurs. This concatenates counter <n> and counter <n–1>, to provide a larger count value.

For full details on counter chaining see *Counters on page 4-139*.

For TRCCNTCTLR2 and TRCCNTCTLR0 this bit is RES0.

RLDSELF<n>, bit[16]

Controls whether a reload event occurs for counter <n>, when counter <n> reaches zero:

0 The counter is in Normal mode.

1 The counter is in Self-reload mode.

For full details on these modes see *Counters on page 4-139*.

RLDEVENT<n>, bits[15:8]

An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes. Selects an event, that when it occurs causes a reload event for counter <n>.

For ETMv4.3 or later, this field is not implemented if TRCIDR4.NUMRSPAIR has the value `0b0000`, which indicates that no counters are implemented.

CNTEVENT<n>, bits[7:0]

An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes. Selects an event, that when it occurs causes counter <n> to decrement.

For ETMv4.3 or later, this field is not implemented if TRCIDR4.NUMRSPAIR has the value `0b0000`, which indicates that no counters are implemented.

7.3.17 TRCCNTRLDVRn, Counter Reload Value Registers, n=0-3

The TRCCNTRLDVRn characteristics are:

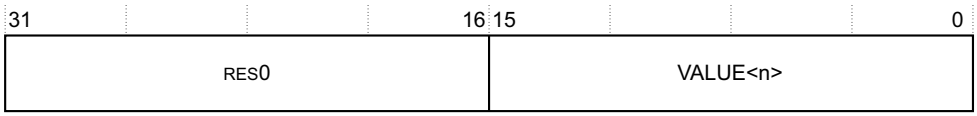
Purpose This sets or returns the reload count value for counter <n>.

Usage constraints Might ignore writes when the trace unit is enabled or not idle.

Configurations The TRCIDR5.NUMCNTR field sets the value of n and therefore controls how many TRCCNTRLDVRs are implemented.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCNTRLDVRn bit assignments are:



Bits[31:16] RES0.

VALUE<n>, bits[15:0]

Contains the reload value for counter <n>. When a reload event occurs for counter <n> then the trace unit copies the VALUE<n> field into counter <n>.

7.3.18 TRCCNTVRn, Counter Value Registers, n=0-3

The TRCCNTVRn characteristics are:

Purpose This sets or returns the value of counter <n>.

Usage constraints

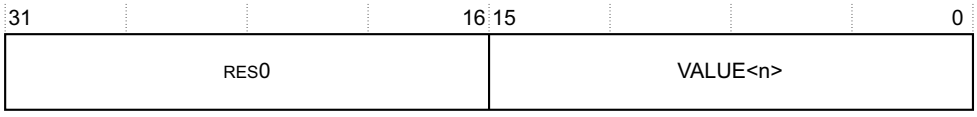
- Might ignore writes when the trace unit is enabled or not idle.
- The count value is only stable when TRCSTATR.PMSTABLE==1.
- If software uses counter <n> then it must write to this register to set the initial counter value.

Configurations The TRCIDR5.NUMCNTR field sets the value of n and therefore controls how many TRCCNTVRs are implemented.

TRCCNTVR0 is not implemented if the reduced function counter is implemented, as defined by TRCIDR5.REDFUNCNTR.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCNTVRn bit assignments are:



Bits[31:16] RES0.

VALUE<n>, bits[15:0]

Contains the count value of counter <n>.

7.3.19 TRCCONFIGR, Trace Configuration Register

The TRCCONFIGR characteristics are:

Purpose Controls the tracing options.

Usage constraints

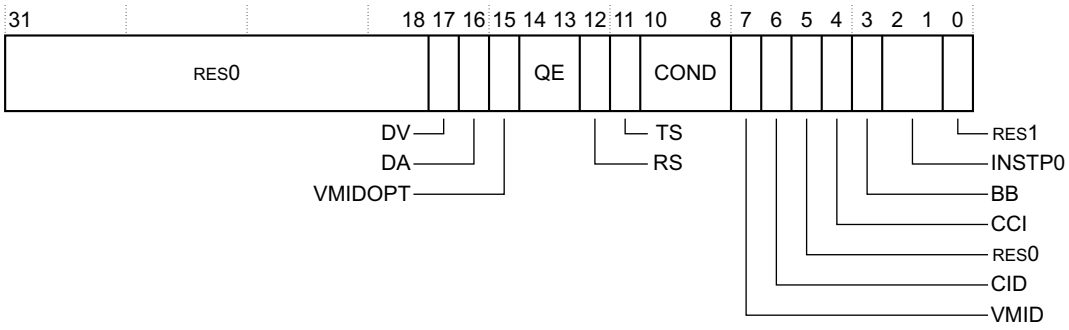
- This register must always be programmed as part of trace unit initialization.
- Might ignore writes when the trace unit is enabled or not idle.

Configurations

- Available in all implementations.
- It is IMPLEMENTATION DEFINED which fields are supported. See the field descriptions for more information.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCCONFIGR bit assignments are:



Bits[31:18] RES0.

DV, bit[17] Data value tracing bit:

0 Data value tracing is disabled.

1 Data value tracing is enabled when INSTP0 is not `0b00`.

TRCIDR0.TRCDATA indicates whether this bit is supported. If it is not supported then this bit is RES0.

DA, bit[16] Data address tracing bit:

0 Data address tracing is disabled.

1 Data address tracing is enabled when INSTP0 is not `0b00`.

TRCIDR0.TRCDATA indicates whether this bit is supported. If it is not supported then this bit is RES0.

VMIDOPT, bit[15] Control bit to select the Virtual context identifier value used by the trace unit, both for trace generation and in the Virtual context identifier comparators. The permitted values are:

`0b0` VTTBR_EL2.VMID is used. If the trace unit supports a Virtual context identifier larger than the VTTBR_EL2.VMID, the upper unused bits are always zero. If the trace unit supports a Virtual context identifier larger than 8 bits and if the VTCR_EL2.VS bit forces use of an 8-bit Virtual context identifier, bits [15:8] of the trace unit Virtual context identifier are always zero.

`0b1` CONTEXTIDR_EL2 is used.

TRCIDR2.VMIDOPT indicates whether this field is implemented.

This field is only present if both the following are true:

- At least ETMv4.1 is implemented.
- TRCIDR2.VMIDOPT indicates this field is implemented.

QE, bits[14:13] Q element enable field:

`0b00` Q elements are disabled.

`0b01` Q elements with instruction counts are enabled. Q elements without instruction counts are disabled.

`0b10` Reserved.

`0b11` Q elements with and without instruction counts are enabled.

TRCIDR0.QSUPP indicates which values of this field are implemented.

TRCCONFIGR.QE must be set to `0b00` if any of the following are true:

- TRCCONFIGR.INSTP0 is not `0b00`
- TRCCONFIGR.COND is not `0b000`
- TRCCONFIGR.BB is not 0.

RS, bit[12] Return stack enable bit:

0 Return stack is disabled.

1 Return stack is enabled.

TRCIDR0.RETSTACK indicates whether this bit is supported. If it is not supported then this bit is RES0.

TS, bit[11] Global timestamp tracing bit:

0 Global timestamp tracing is disabled.

1 Global timestamp tracing is enabled. TRCTSCTLR controls the insertion of timestamps in the trace.

TRCIDR0.TSSIZE indicates whether this bit is supported. If it is not supported then this bit is RES0.

COND, bits[10:8] Conditional instruction tracing bit. The permitted values are:

`0b000` Conditional instruction tracing is disabled.

`0b001` Conditional load instructions are traced.

`0b010` Conditional store instructions are traced.

`0b011` Conditional load and store instructions are traced.

`0b111` All conditional instructions are traced.

All other values are reserved.

TRCIDR0.TRCCOND indicates whether this field is supported. If it is not supported then this field is RES0.

VMID, bit[7] Virtual context identifier tracing bit:

0 Virtual context identifier tracing is disabled.

1 Virtual context identifier tracing is enabled.

TRCIDR2.VMIDSIZE indicates whether this bit is supported. If it is not supported then this bit is RES0.

CID, bit[6] Context ID tracing bit:

0 Context ID tracing is disabled.

1 Context ID tracing is enabled.

TRCIDR2.CIDSIZE indicates whether this bit is supported. If it is not supported then this bit is RES0.

Bit[5] RES0.

CCI, bit[4] Cycle counting instruction trace bit:

0 Cycle counting in the instruction trace is disabled.

1 Cycle counting in the instruction trace is enabled. TRCCCCTLR controls the threshold value for cycle counting.

TRCIDR0.TRCCCI indicates whether this bit is supported. If it is not supported then this bit is RES0.

BB, bit[3] Branch broadcast mode bit:

0 Branch broadcast mode is disabled.

1 Branch broadcast mode is enabled. TRCBBCTLR controls which regions of memory are enabled to use branch broadcasting.

TRCIDR0.TRCBB indicates whether this bit is supported. If it is not supported then this bit is RES0.

INSTP0, bits[2:1] Instruction P0 field. This field controls whether load and store instructions are traced as P0 instructions:

`0b00` Do not trace load and store instructions as P0 instructions.

`0b01` Trace load instructions as P0 instructions.

`0b10` Trace store instructions as P0 instructions.

`0b11` Trace load and store instructions as P0 instructions.

TRCIDR0.INSTP0 indicates whether this field is supported. If it is not supported then this field is RES0.

If TRCCONFIGR.INSTP0 is `0b00`, then the behavior of the trace unit is CONSTRAINED UNPREDICTABLE if either:

- TRCCONFIG. DA is not `0b0`.
- TRCCONFIGR.DV is not `0b0`.

In this case, any of the following might occur:

- Data trace might or might not be generated.
- Event tracing in the data trace stream might or might not occur.
- ATB triggers in the data trace stream might or might not occur.

Bit[0] RES1.

7.3.20 TRCDEVAFF0, Device Affinity register 0

The TRCDEVAFF0 characteristics are:

Purpose TRCDEVAFF0 returns the lower 32 bits of the PE MPIDR, that is, MPIDR[31:0]. This enables a debugger to determine which PE in a system with multiple PEs the trace unit relates to.

The value given is the value seen in the PE in the highest implemented Exception level, and is not affected by the VMPIDR or VMPIDR_EL2:

- For Armv7 PEs without the Virtualization Extensions, it is the value of the MPIDR.
- For Armv7 PEs with the Virtualization Extensions, it is the value of the MPIDR as seen from Hyp mode or Secure state, unaffected by VMPIDR.
- For Armv6-M, Armv7-M, and Armv8-M PEs, TRCDEVAFF0 is RES0H.
- For Armv8-A and Armv8-R PEs where EL1 is the highest implemented Exception level, it is the value of the lower 32 bits of the MPIDR_EL1.
- For Armv8-A and Armv8-R PEs where EL2 or EL3 is the highest implemented Exception level, it is the value of the lower 32 bits of the MPIDR_EL1 as seen from EL2 or EL3, unaffected by VMPIDR_EL2.

If the trace unit is shared between multiple PEs, the value in this register indicates the MPIDR value of the PE that TRCPROCSELR selects.

Usage constraints Accessible only from the memory-mapped or external debugger interfaces.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register returns a value that depends on the value of the MPIDR in the PE being traced. See the *Armv8 Architecture Reference Manual*. See also *Register summary on page 7-336*.

7.3.21 TRCDEVAFF1, Device Affinity register 1

The TRCDEVAFF1 characteristics are:

Purpose TRCDEVAFF1 returns the upper 32 bits of the PE MPIDR, that is, MPIDR[63:32] if the PE has a 64-bit architecture. This enables a debugger to determine which PE in a system with multiple PEs the trace unit relates to.

The value given is the value seen in the PE in the highest implemented Exception level, and is not affected by the VMPIDR or VMPIDR_EL2:

- For Armv7 PEs without the Virtualization Extensions, it is the value of the MPIDR.

- For Armv7 PEs with the Virtualization Extensions, it is the value of the MPIDR as seen from Hyp mode or Secure state, unaffected by VMPIDR.
- For Armv6-M, Armv7-M, and Armv8-M PEs, TRCDEVAFF1 is RES0H.
- For Armv8-A and Armv8-R PEs where EL1 is the highest implemented Exception level, it is the value of the upper 32 bits of the MPIDR_EL1.
- For Armv8-A and Armv8-R PEs where EL2 or EL3 is the highest implemented Exception level, it is the value of the upper 32 bits of the MPIDR_EL1 as seen from EL2 or EL3, unaffected by VMPIDR_EL2.

If the trace unit is shared between multiple PEs, the value in this register indicates the MPIDR value of the PE that TRCPROCSELR selects.

Usage constraints Accessible only from the memory-mapped or external debugger interfaces.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register returns a value that depends on the value of the MPIDR in the PE being traced. See the *Armv8 Architecture Reference Manual*. See also *Register summary on page 7-336*.

7.3.22 TRCDEVARCH, Device Architecture register

The TRCDEVARCH characteristics are:

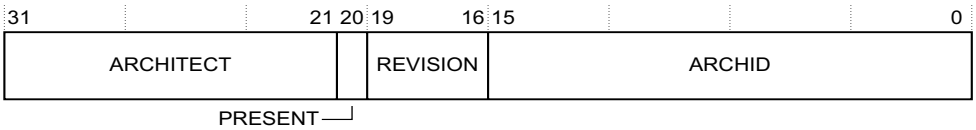
Purpose Identifies the trace unit as an ETMv4 component. This enables debuggers to make some use of a component that conforms to the architecture definition but whose part number is not necessarily recognized.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register that returns a value of `0x47704A13`. See also *Register summary on page 7-336*.

The TRCDEVARCH bit assignments are:



ARCHITECT, bits[31:21]

Defines the architect of the component. This is always Arm Limited, so:

Bits[31:28] Return `0x4`, the JEP106 continuation code for Arm.

Bits[27:21] Return `0b0111011`, the JEP106 code for Arm.

See the *Standard Manufacturers Identification Code* for information about JEP106.

PRESENT, bit[20] Indicates the presence of this register. RAO.

REVISION, bits[19:16]

Identifies the revision number of the trace unit architecture.

The permitted values are:

`0x0` Indicates ETMv4.0.

`0x1` Indicates ETMv4.1.

`0x2` Indicates ETMv4.2.

`0x3` Indicates ETMv4.3.

`0x4` Indicates ETMv4.4.

`0x5` Indicates ETMv4.5.

ARCHID, bits[15:0] Architecture ID. Defines the component to be an ETMv4 trace unit, so:

Bits[15:12] Return `0x4`, the architecture version for ETMv4.

Bits[11:0] Return `0xA13`, the architecture part number for an ETMv4 trace unit.

7.3.23 TRCDEVID, Device ID register

The TRCDEVID characteristics are:

Purpose Register is Reserved

Note

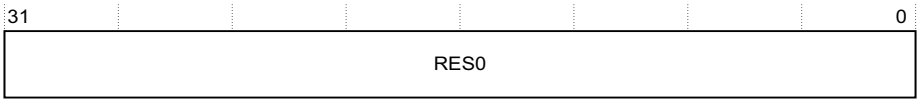
The *CoreSight Architecture Specification* requires every CoreSight component to implement a Device ID register.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. See also *Register summary on page 7-336*.

The TRCDEVID bit assignments are:



Bits[31:0] RES0.

7.3.24 TRCDEVTYPE, Device Type register

The TRCDEVTYPE characteristics are:

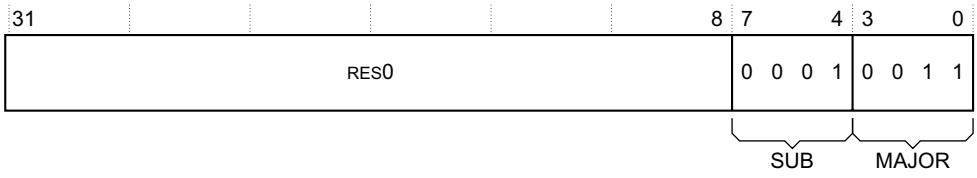
Purpose Indicates what type of CoreSight device the trace unit is. See the *CoreSight Architecture Specification* for more information about the CoreSight device types.

Usage constraints Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO management register that returns a value of `0x13`. See also *Register summary on page 7-336*.

The TRCDEVTYPE bit assignments are:



Bits[31:8] RES0.

SUB, bits[7:4] Returns `0x1`, to indicate that the trace unit generates PE trace.

All other values are reserved.

MAJOR, bits[3:0] Returns `0x3`, to indicate that the trace unit is a trace source.

All other values are reserved.

7.3.25 TRCDVCMRn, Data Value Comparator Mask Registers, n=0-7

The TRCDVCMRn characteristics are:

Purpose Contains a data mask value.

Usage constraints Might ignore writes when the trace unit is enabled or not idle.

Configurations The number, n, of TRCDVCMRs is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMDVC.

Attributes A 64-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCDVCMRn bit assignments are:

MASK, bits[63:0] Data mask value. The implemented width of this field is IMPLEMENTATION DEFINED and is set by TRCIDR2.DVSIZE. Unimplemented bits are RES0.

If a bit is set to 1 in the mask then the comparator ignores that bit number for a data value comparison. Software must ensure that the relevant bit in TRCDVCVRn is programmed to 0, otherwise the comparator might fail to match.

The data value comparators can support implementations that use multiple data widths. When the trace unit compares the TRCDVCVRn.VALUE field with a data value that has a width less than this field, then software must also write the data mask value to both the upper bits and the lower bits of the MASK field. For example, in a system that supports both 32-bit and 64-bit data widths, then software must set MASK[63:32]==MASK[31:0] if the trace unit is to compare a 32-bit data value. See *Data value comparators on page 4-153* for more information.

7.3.26 TRCDVCVRn, Data Value Comparator Value Registers, n=0-7

The TRCDVCVRn characteristics are:

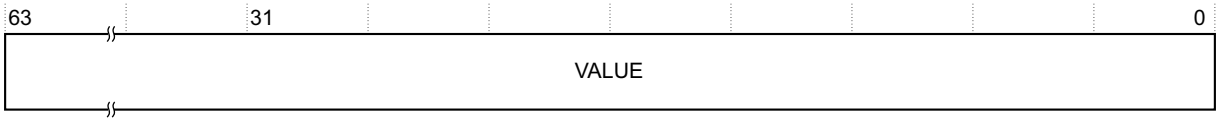
Purpose Contains a data value.

Usage constraints Might ignore writes when the trace unit is enabled or not idle.

Configurations The number, n, of TRCDVCVRs is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMDVC.

Attributes A 64-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCDVCVRn bit assignments are:



VALUE, bits[63:0] Data value. The implemented width of this field is IMPLEMENTATION DEFINED and is set by TRCIDR2.DVSIZE. Unimplemented bits are RES0.

The data value comparators can support implementations that use multiple data widths. When the trace unit compares the VALUE field with a data value that has a width less than this field, then software must also write the comparison data value to both the upper bits and the lower bits of the VALUE field. For example, in a system that supports both 32-bit and

64-bit data widths, then software must set VALUE[63:32]==VALUE[31:0] if the trace unit is to compare a 32-bit data value. See *Data value comparators on page 4-153* for more information.

7.3.27 TRCEVENTCTL0R, Event Control 0 Register

The TRCEVENTCTL0R characteristics are:

Purpose Controls the tracing of arbitrary events.

Each of the EVENT fields in this register is an event selector. If any of the selected events occurs and the corresponding bit in TRCEVENTCTL1R.INSTEN == 1, then an Event element is generated in the instruction trace stream. If any of the selected events occurs and the corresponding bit in TRCEVENTCTL1R.DATAEN == 1, then an Event element is generated in the data trace stream. See *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177*.

Usage constraints

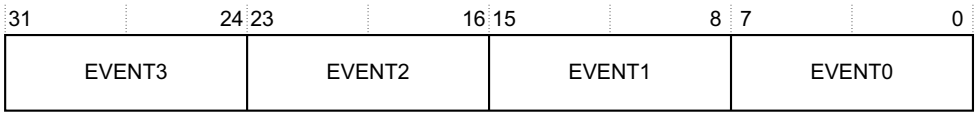
- This register must always be programmed as part of trace unit initialization.
- Might ignore writes when the trace unit is enabled or not idle.

Configurations

- For ETMv4.2 or earlier, TRCEVENTCTL0R is available in all implementations.
- For ETMv4.3 or later, TRCEVENTCTL0R is not implemented if TRCIDR4.NUMRSPAIR is `0b0000` because no events are implemented. For all other values of TRCIDR4.NUMRSPAIR, TRCEVENTCTL0R is implemented.
- It is IMPLEMENTATION DEFINED how many EVENT fields are supported. TRCIDR0.NUMEVENT indicates how many fields are supported.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCEVENTCTL0R bit assignments are:



EVENT3, bits[31:24] Only supported if TRCIDR0.NUMEVENT==`0b11`.

EVENT2, bits[23:16] Only supported if TRCIDR0.NUMEVENT==`0b11` or `0b10`.

EVENT1, bits[15:8] Only supported if TRCIDR0.NUMEVENT==`0b11`, `0b10`, or `0b01`.

EVENT0, bits[7:0] Support for this field depends on the ETM architecture version:

ETMv4.2 or earlier

EVENT0 is always supported.

ETMv4.3 or later

EVENT0 is only supported if TRCIDR4.NUMRSPAIR is not `0b0000`.

7.3.28 TRCEVENTCTL1R, Event Control 1 Register

The TRCEVENTCTL1R characteristics are:

Purpose Controls the behavior of the events that TRCEVENTCTL0R selects.

Usage constraints

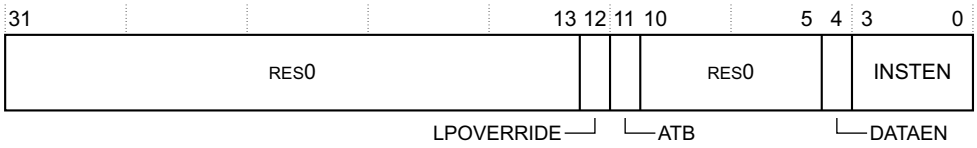
- This register must always be programmed as part of trace unit initialization.
- Might ignore writes when the trace unit is enabled or not idle.

Configurations

- Available in all implementations.
- It is IMPLEMENTATION DEFINED whether the LPOVERRIDE and ATB bits are supported.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCEVENTCTL1R bit assignments are:



Bits[31:13] RES0.

LPOVERRIDE, bit[12]

Low-power state behavior override bit. Controls how a trace unit behaves in low-power state:

0 Trace unit low-power state behavior is not affected. That is, the trace unit is enabled to enter low-power state.

1 Trace unit low-power state behavior is overridden. That is, entry to a low-power state does not affect the trace unit resources or trace generation. See *Trace unit behavior on a PE low-power state on page 3-105* for more information.

TRCIDR5.LPOVERRIDE indicates whether this bit is supported. If it is not supported then this bit is RES0.

ATB, bit[11] *AMBA Trace Bus* (ATB) trigger enable bit:

0 ATB trigger is disabled.

1 ATB trigger is enabled. If a CoreSight ATB interface is implemented then when event 0 occurs the trace unit sets:

- **ATID**==`0x7D`.
- **ATDATA** to the value of TRCTRACEIDR. If the width of **ATDATA** is greater than the width of TRCTRACEIDR.TRACEID then the trace unit zeros the upper **ATDATA** bits.

If event 0 is programmed to occur based on program execution, such as an address comparator, the ATB trigger might not be inserted into the ATB stream at the same time as any trace generated by that program execution is output by the trace unit. Typically, the generated trace might be buffered in a trace unit which means that the ATB trigger would be output before the associated trace is output.

If event 0 is asserted multiple times in close succession, the trace unit is required to generate an ATB trigger for the first assertion, but might ignore one or more of the subsequent assertions. Arm recommends that the window in which event 0 is ignored is limited only by the time taken to output an ATB trigger.

Note

If the trace unit is in an overflow state when event 0 occurs then it must generate an ATB trigger. If event 0 occurs multiple times when the trace unit is in overflow state then the trace unit must generate at least one ATB trigger.

TRCIDR5.ATBTRIG indicates whether this bit is supported. If it is not supported then this bit is RES0.

Bits[10:5] RES0.

DATAEN, bit[4] Data event enable bit. If event 0 occurs when DATAEN is:

0 The trace unit does not generate an Event element.

1 The trace unit generates an Event element in the data trace stream.

This bit is only implemented if the following conditions are met:

- Data tracing is implemented, as indicated by TRCIDR0.TRCDATA.

- For ETMv4.3 or later, TRCIDR4.NUMRSPAIR does not have the value 0b0000, which indicates that no events are implemented.

If this field is not implemented, it is RES0.

INSTEN, bits[3:0] Instruction event enable field. Each bit represents an event, so the number of events, *n*=0-3. If event *n* occurs when INSTEN[*n*] is:

0 The trace unit does not generate an Event element.

1 The trace unit generates an Event element for event *n*, in the instruction trace stream.

TRCIDR0.NUMEVENT indicates which bits of this field are implemented. Unimplemented bits are RES0.

For ETMv4.3 or later, this field is not implemented if TRCIDR4.NUMRSPAIR has the value 0b0000, which indicates that no events are implemented.

7.3.29 TRCEXTINSELR, External Input Select Register

The TRCEXTINSELR characteristics are:

Purpose Use this to set, or read, which external inputs are resources to the trace unit.

Usage constraints Might ignore writes when the trace unit is enabled or not idle.

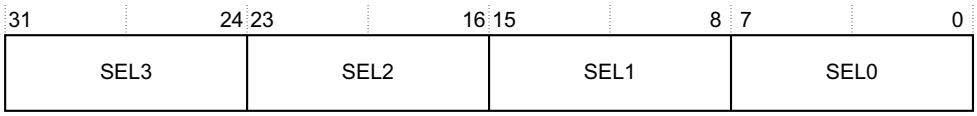
Configurations Only implemented if TRCIDR5.NUMEXTINSEL > 0.

The TRCIDR5.NUMEXTINSEL field controls how many input select resources are supported.

The TRCIDR5.NUMEXTIN field controls how many inputs, from a maximum of 256, are supported.

Attributes A 32-bit RW trace register. This register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCEXTINSELR bit assignments are:



SEL3[31:24] Supported only if TRCIDR5.NUMEXTINSEL==0b100. This field is a binary value, of up to 8 bits, that selects which external input is a resource for the trace unit.

SEL2[23:16] Supported only if TRCIDR5.NUMEXTINSEL==0b100 or 0b011. This field is a binary value, of up to 8 bits, that selects which external input is a resource for the trace unit.

SEL1[15:8] Supported only if TRCIDR5.NUMEXTINSEL==0b100, 0b011, or 0b010. This field is a binary value, of up to 8 bits, that selects which external input is a resource for the trace unit.

SEL0[7:0] Supported only if TRCIDR5.NUMEXTINSEL==0b100, 0b011, 0b010, or 0b001. This field is a binary value, of up to 8 bits, that selects which external input is a resource for the trace unit.

The TRCIDR5.NUMEXTIN field defines how many external inputs are supported and therefore each SEL<n> field might not support all eight bits. For example, if an implementation supports a maximum of 50 external inputs then it requires a 6-bit field so the upper two bits in each SEL<n> field might not be supported.

7.3.30 TRCIDR0, ID Register 0

The TRCIDR0 characteristics are:

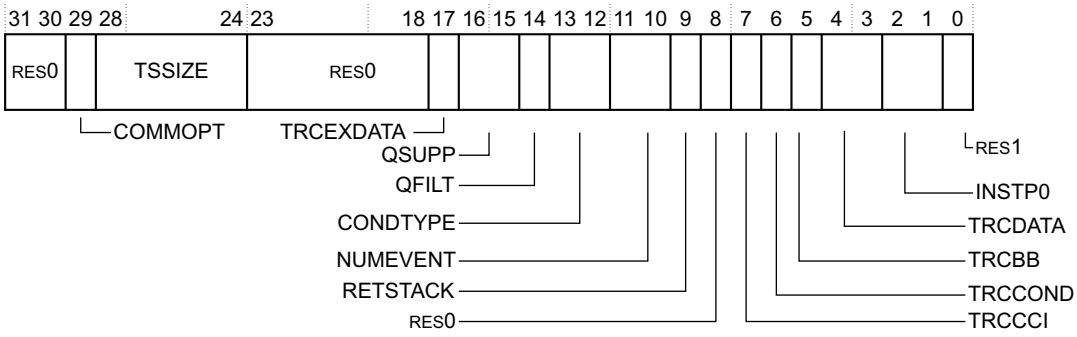
Purpose Returns the tracing capabilities of the trace unit.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR0 bit assignments are:



Bits[31:30] RES0.

COMMOPT, bit[29] Commit mode field. Unless TRCIDR0.TRCCCI is zero, and this field is then RES0, the permitted values are:

0 Commit mode 0.

1 Commit mode 1.

The commit mode that the trace unit is using affects the encoding of the *Cycle Count Format 1 instruction trace packet on page 6-267* and the *Cycle Count Format 3 instruction trace packet on page 6-269*.

TSSIZE, bits[28:24] Global timestamp size field. The permitted values are:

`0b00000` Global timestamping is not implemented.

`0b00110` Implementation supports a maximum global timestamp of 48 bits.

`0b01000` Implementation supports a maximum global timestamp of 64 bits.

All other values are reserved.

When global timestamping is implemented then:

- TRCCONFIGR.TS is supported
- TRCTSCTLR is supported.

If the PE implements Armv8.4-Trace, the timestamp size must be 64 bits.

Bits[23:20] RES0.

BF, bit[19:18]

Indicates support for branch future:

`0b00` Branch future support is not implemented.

`0b01` Branch future support is implemented. This value is only permitted from ETMv4.5.

All other values are reserved.

TRCEXDATA, bit[17]

Indicates support for the tracing of data transfers for exceptions and exception returns on Armv6-M, Armv7-M, and Armv8-M PEs:

0 TRCVDCTLR.TRCEXDATA is not implemented.

1 TRCVDCTLR.TRCEXDATA is implemented.

QSUPP, bits[16:15] Q element support field. The permitted values are:

- `0b00` Q element support is not implemented. TRCCONFIGR.QE is RES0.
- `0b01` Q element support is implemented, and only supports Q elements with instruction counts. TRCCONFIGR.QE can only take the values `0b00` or `0b01`.
- `0b10` Q element support is implemented, and only supports Q elements without instruction counts. TRCCONFIGR.QE can only take the values `0b00` or `0b11`.
- `0b11` Q element support is implemented, and supports both Q elements with instruction counts and Q elements without instruction counts. TRCCONFIGR.QE is fully implemented.

QFILT, bit[14]

When QSUPP > `0b00` this is the Q element filtering support field. The permitted values are:

- **0** Q element filtering is not implemented.
- **1** Q element filtering is implemented. TRCQCTLR is implemented.

When QSUPP==`0b00` this field is RES0.

CONDTYPE, bits[13:12]

Conditional tracing field. The permitted values are:

- `0b00` The trace unit indicates only if a conditional instruction passes or fails its condition code check.
- `0b01` The trace unit provides the value of the APSR condition flags, for a conditional instruction.

All other values are reserved.

NUMEVENT, bits[11:10]

Number of events field. Indicates how many events the trace unit supports:

- `0b00` The meaning of NUMEVENT depends on the ETM architecture version:
  - For ETM4.2 or earlier, this value of NUMEVENT indicates that the trace unit supports one event.
  - For ETMv4,3 or later, if TRCIDR4.NUMRSPAIR is `0b0000`, NUMEVENT takes this value, which indicates that no events are implemented. For all other values of TRCIDR4.NUMRSPAIR, this value of NUMEVENTS indicates that the trace unit supports one event.
- `0b01` The trace unit supports 2 events.
- `0b10` The trace unit supports 3 events.
- `0b11` The trace unit supports 4 events.

This field controls how many fields are supported in TRCEVENTCTL0R. This field indicates the size of TRCEVENTCTL1R.INSTEN.

RETSTACK, bit[9] Return stack bit. Indicates if the implementation supports a return stack:

- **0** Return stack is not implemented.
- **1** Return stack is implemented, so TRCCONFIGR.RS is supported.

Bit[8] RES0.

TRCCCI, bit[7] Cycle counting instruction bit. Indicates if the trace unit supports cycle counting for instructions:

- **0** Cycle counting in the instruction trace is not implemented.
- **1** Cycle counting in the instruction trace is implemented, so:
  - TRCCONFIGR.CCI is supported
  - TRCCCCTLR is supported.

TRCCOND, bit[6] Conditional instruction tracing support bit. Indicates if the trace unit supports conditional instruction tracing:

- **0** Conditional instruction tracing is not supported.

1 Conditional instruction tracing is supported, so TRCCONFIGR.COND is supported.

Arm recommends that conditional instruction trace is implemented for Armv8.1-M PEs.

TRCBB, bit[5] Branch broadcast tracing support bit. Indicates if the trace unit supports branch broadcast tracing:

0 Branch broadcast tracing is not supported.

1 Branch broadcast tracing is supported, so:

- TRCCONFIGR.BB is supported.
- TRCBBCTLR is supported if TRCIDR4.NUMACPAIRS is greater than zero.

TRCDATA, bits[4:3] Conditional tracing field. The permitted values are:

`0b00` Data tracing is not supported.

`0b11` Tracing of data addresses and data values is supported, so:

- TRCCONFIGR.DA is supported.
- TRCCONFIGR.DV is supported.
- TRCSTALLCTLR.DATADISCARD is supported.
- TRCSTALLCTLR.INSTPRIORITY is supported.
- TRCSTALLCTLR.DSTALL is supported.
- TRCEVENTCTL1R.DATAEN is implemented.

All other values are reserved.

Data tracing is not permitted for Armv8.1-M implementations.

INSTP0, bits[2:1] P0 tracing support field. The permitted values are:

`0b00` Tracing of load and store instructions as P0 elements is not supported.

`0b11` Tracing of load and store instructions as P0 elements is supported, so TRCCONFIGR.INSTP0 is supported.

When TRCIDR0.TRCDATA is `0b00`, INSTP0 must be `0b00`.

All other values are reserved.

Bit[0] RES1.

7.3.31 TRCIDR1, ID Register 1

The TRCIDR1 characteristics are:

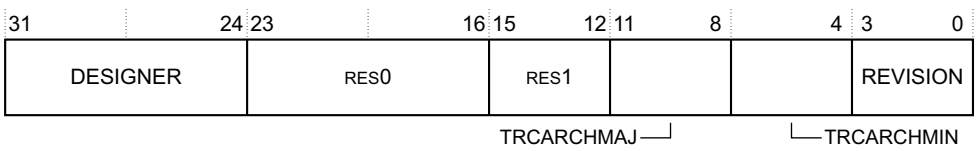
Purpose Returns the base architecture of the trace unit.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR1 bit assignments are:



DESIGNER, bits[31:24]

Indicates which company designed the trace unit. The permitted values are:

`0x41` Arm Limited.

0x42	Broadcom Corporation.
0x43	Cavium Inc.
0x44	Digital Equipment Corporation.
0x49	Infineon Technologies AG.
0x4D	Motorola or Freescale Semiconductor Inc.
0x4E	NVIDIA Corporation.
0x50	Applied Micro Circuits Corporation.
0x51	Qualcomm Inc.
0x56	Marvell International Inc.
0x69	Intel Corporation.

All other values are reserved.

Bits[23:16] RES0.

Bits[15:12] RES1.

TRCARCHMAJ, bits[11:8]

Indicates the major version number of the trace unit architecture. The permitted value is:

0x4	Indicates ETMv4.

All other values are reserved.

TRCARCHMIN, bits[7:4]

Identifies the minor version number of the trace unit architecture. The permitted value is:

0x0	Indicates ETMv4 minor version 0.
0x1	Indicates ETMv4 minor version 1.
0x2	Indicates ETMv4 minor version 2.
0x3	Indicates ETMv4 minor version 3.
0x4	Indicates ETMv4 minor version 4.
0x5	Indicates ETMv4 minor version 5.

All other values are reserved.

REVISION, bits[3:0] Returns an IMPLEMENTATION DEFINED value that identifies the revision of:

- The trace registers.
- The OS Lock registers.

Arm recommends:

- The initial implementation sets REVISION==0x0 and the field then increments for any subsequent implementations. However, it is acceptable to omit some values or use another scheme to identify the revision number.
- That TRCPIDR2.REVISION==TRCIDR1.REVISION. However, in situations where it is difficult to align these fields, such as with a metal layer fix then it is acceptable to change the REVISION fields independently.

7.3.32 TRCIDR2, ID Register 2

The TRCIDR2 characteristics are:

Purpose Returns the maximum size of the following parameters in the trace unit:

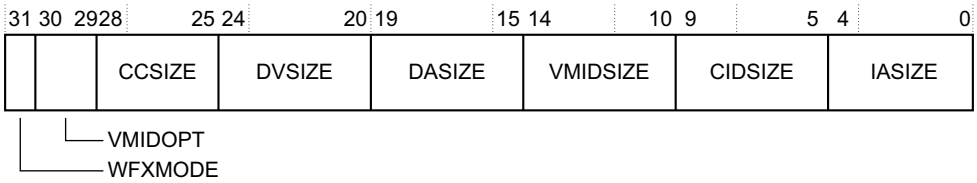
- Data value.
- Data address.
- Virtual context identifier.
- Context ID.
- Instruction address.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR2 bit assignments are:



WFXMODE, bit[31]

For ETMv4.2 and earlier, this field is RES0.

For ETM4.3 or later, this field indicates whether `WFI` and `WFE` instructions are classified as branch instructions:

`0b0` `WFI` and `WFE` instructions are not classified as branch instructions.

`0b1` `WFI` and `WFE` instructions are classified as direct branch instructions.

VMIDOPT, bits[30:29]

Indicates the options for observing the Virtual context identifier in the PE:

`0b00` TRCCONFIGR.VMIDOPT is not implemented and is RES0. VTTBR_EL2.VMID is the value of the Virtual context identifier in the trace unit.

`0b01` TRCCONFIGR.VMIDOPT is implemented.

For a trace unit for an Armv8-A PE, this field must be `0b01` for trace units where the PE supports the Virtualization Host Extensions, otherwise this field must be `0b00`.

Other values of this field are reserved.

Note

This field is only present from ETMv4.1, and not in earlier versions of the architecture.

CCSIZE, bits[28:25] Indicates the size of the cycle counter in bits minus 12.

`0b0000` The cycle counter is 12 bits in length.

`0b0001` The cycle counter is 13 bits in length.

. .

. .

. .

`0b1000` The cycle counter is 20 bits in length.

All other values are reserved.

This field is RES0 if cycle counting is not implemented, as indicated by TRCIDR0.TRCCCI.

DVSIZE, bits[24:20] Indicates the data value size in bytes. The permitted values are:

`0b00000` Data value tracing is not supported. Therefore, an implementation must set TRCIDR0.TRCDATA==`0b00`.

`0b00100` Maximum of 32-bit data value size.

`0b01000` Maximum of 64-bit data value size. This value is not permitted when tracing Armv6 and Armv7 PEs.

All other values are reserved.

DASIZE, bits[19:15] Indicates the data address size in bytes. The permitted values are:

- `0b00000` Data address tracing is not supported. Therefore, an implementation must also set TRCIDR0.TRCDATA==`0b00`.
- `0b00100` Maximum of 32-bit data address size.
- `0b01000` Maximum of 64-bit data address size. This value is not permitted when tracing Armv6 and Armv7 PEs.

All other values are reserved.

VMIDSIZE, bits[14:10]

Indicates the trace unit Virtual context identifier size. The permitted values are:

- `0b00000` VMID tracing is not supported.
- `0b00001` 8-bit Virtual context identifier size, TRCCONFIGR.VMID is supported.
- `0b00010` 16-bit Virtual context identifier size, TRCCONFIGR.VMID is supported.
- `0b00100` 32-bit Virtual context identifier size, TRCCONFIGR.VMID is supported.

All other values are reserved.

Note

The values that indicate a 16-bit and 32-bit Virtual context identifier size are only permitted from ETMv4.1.

For a trace unit for an Armv8-A PE, the trace unit Virtual context identifier size must be at least as large as the Virtual context identifier size of the PE being traced. If the PE supports the Virtualization Host Extensions, a 32-bit Virtual context identifier must be supported.

CIDSIZE, bits[9:5] Indicates the Context ID size. The permitted values are:

- `0b00000` Context ID tracing is not supported.
- `0b00100` Maximum of 32-bit Context ID size, so TRCCONFIGR.CID is supported.

All other values are reserved.

IASIZE, bits[4:0] Indicates the instruction address size. The permitted values are:

- `0b00100` Maximum of 32-bit address size.
- `0b01000` Maximum of 64-bit address size. This value is not permitted when tracing Armv6, Armv7, Armv8-M, and Armv8-R PEs.

All other values are reserved.

7.3.33 TRCIDR3, ID Register 3

The TRCIDR3 characteristics are:

Purpose Indicates:

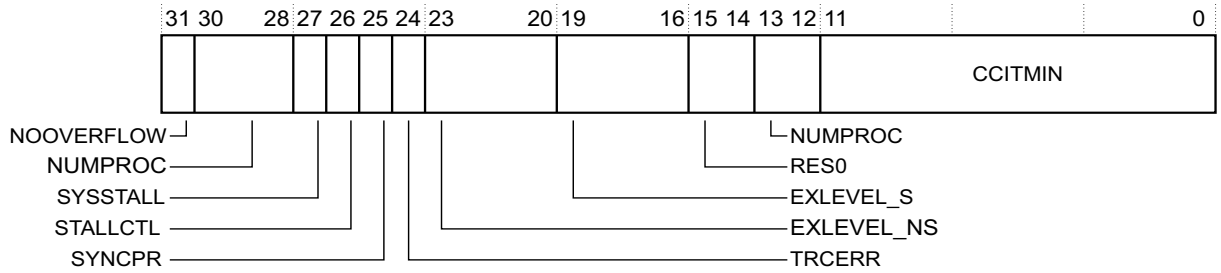
- If TRCVICTLR.TRCERR is supported.
- The number of PEs available for tracing.
- If an Exception level supports instruction tracing.
- The minimum threshold value for instruction trace cycle counting.
- If the synchronization period is fixed.
- If TRCSTALLCTLR is supported and if so, whether it supports trace overflow prevention and supports stall control of the PE.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR3 bit assignments are:



NOOVERFLOW, bit[31]

Indicates if TRCSTALLCTLR.NOOVERFLOW is supported:

0 TRCSTALLCTLR.NOOVERFLOW is not supported, or STALLCTL==0.

1 TRCSTALLCTLR.NOOVERFLOW is supported.

SYSSTALL, bit[27] Indicates if the implementation can support stall control:

0 The system does not support stall control of the PE.

1 The system can support stall control of the PE.

Only when SYSSTALL==1 and STALLCTL==1 does the system support stalling of the PE.

STALLCTL, bit[26] Indicates if TRCSTALLCTLR is supported:

0 TRCSTALLCTLR is not supported.

1 TRCSTALLCTLR is supported.

SYNCPR, bit[25] Indicates if an implementation has a fixed synchronization period:

0 TRCSYNCPR is read-write so software can change the synchronization period.

1 TRCSYNCPR is read-only so the synchronization period is fixed.

TRCERR, bit[24] Indicates if TRCVICTLR.TRCERR is supported:

0 TRCVICTLR.TRCERR is not supported

1 TRCVICTLR.TRCERR is supported.

EXLEVEL_NS, bits[23:20]

In Non-secure state, each bit indicates whether instruction tracing is supported for the corresponding Exception level:

0 In Non-secure state, Exception level *n* is not supported so the corresponding bit in:

- TRCACATRn.EXLEVEL_NS is not supported.
- TRCVICTLR.EXLEVEL_NS is not supported.

For Armv8-M PEs, the trace unit does not distinguish between the Secure and Non-secure states, and treats both states as Secure state. This field always reads as zero, indicating no Non-secure states are supported.

1 In Non-secure state, Exception level *n* is supported so the corresponding bit in:

- TRCACATRn.EXLEVEL_NS is supported.
- TRCVICTLR.EXLEVEL_NS is supported.

The Exception levels are:

Bit[20] Exception level 0.

Bit[21] Exception level 1.

Bit[22] Exception level 2.

Bit[23] SBZ. EXLEVEL_NS[3] is never implemented.

EXLEVEL_S, bits[19:16]

In Secure state, each bit indicates whether instruction tracing is supported for the corresponding Exception level:

0 In Secure state, Exception level *n* is not supported so the corresponding bit in:
- TRCACATRn.EXLEVEL_S is not supported.
- TRCVICTLR.EXLEVEL_S is not supported.

1 In Secure state, Exception level *n* is supported so the corresponding bit in:
- TRCACATRn.EXLEVEL_S is supported.
- TRCVICTLR.EXLEVEL_S is supported.

The Exception levels are:

Bit[16] Exception level 0.

Bit[17] Exception level 1.

Bit[18] Exception level 2.

Bit[19] Exception level 3.

RES0, bits[15:14] RES0.

NUMPROC, bits[13:12], [30:28]

Indicates the number of PEs available for tracing. The possible values are:

`0b00000` The trace unit can trace one PE.

`0b00001` The trace unit can trace two PEs.

`0b00010` The trace unit can trace three PEs.

. .

. .

. .

`0b11111` The trace unit can trace 32 PEs.

This field uses a combination of bits [30:28] and bits [13:12] to form a single 5-bit field. Bits [13:12] form the top bits of this field.

This field sets the maximum value of TRCPROCSELR.PROCSEL.

For Armv8.4-A or above PEs that implement Armv8.4-Trace, the trace unit must not be shared and TRCIDR3.NUMPROC must be zero.

CCITMIN, bits[11:0] Indicates the minimum value that can be programmed in TRCCCCTLR.THRESHOLD.

When cycle counting in the instruction trace is supported, that is TRCIDR0.TRCCCI==1, then the minimum value of this field is `0x001`, otherwise it is RES0.

7.3.34 TRCIDR4, ID Register 4

The TRCIDR4 characteristics are:

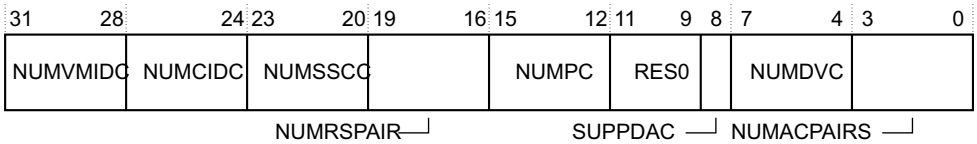
Purpose Returns how many resources the trace unit supports.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR4 bit assignments are:



NUMVMIDC, bits[31:28]

Indicates the number of Virtual context identifier comparators that are available for tracing. The permitted values are:

`0b0000` No Virtual context identifier comparators are available.

`0b0001` The implementation has one Virtual context identifier comparator.

`0b0010` The implementation has two Virtual context identifier comparators.

. .

. .

. .

`0b1000` The implementation has eight Virtual context identifier comparators.

All other values are reserved.

NUMCIDC, bits[27:24]

Indicates the number of Context ID comparators that are available for tracing. The permitted values are:

`0b0000` No Context ID comparators are available.

`0b0001` The implementation has one Context ID comparator.

`0b0010` The implementation has two Context ID comparators.

. .

. .

. .

`0b1000` The implementation has eight Context ID comparators.

All other values are reserved.

NUMSSCC, bits[23:20]

Indicates the number of single-shot comparator controls that are available for tracing. The permitted values are:

`0b0000` No single-shot comparator controls are available.

`0b0001` The implementation has one single-shot comparator control.

`0b0010` The implementation has two single-shot comparator controls.

. .

. .

. .

`0b1000` The implementation has eight single-shot comparator controls.

All other values are reserved.

NUMRSPAIR, bits[19:16]

Indicates the number of resource selection pairs that are available for tracing. The permitted values are:

`0b0000` For ETM4.2 or earlier, the implementation has one resource selection pair.

For ETM4.3 or later, this value imposes the following restrictions on the trace unit resources and resource selectors:

- No data value comparators are available, and TRCIDR4.NUMDVC takes the value `0b0000`.
- No external inputs are available, and TRCIDR5.NUMEXTIN takes the value `0b000000000`.

- No external input selectors are available, and TRCIDR5.NUMEXTINSEL takes the value 0b000.
- No counters are available, and TRCIDR5.NUMCNTR takes the value 0b000.
- No sequencer states are available, and TRCIDR5.NUMSEQSTATE takes the value 0b000.
- The trace unit might implement address comparators, Context ID comparators, Virtual context identifier comparators, or PE comparator inputs, but, when implemented, these comparators are only selectable for precise ViewInst filtering.
- The trace unit might implement Single Shot Comparator Controls, but, when implemented, the status of these controls is only visible in TRCSSCSRn.
- No resource selectors are implemented and the TRUE or FALSE resource selectors are not available.

For ETM4.3 or later, this value imposes the following restrictions on events and event selectors:

- No events are implemented, and TRCIDR0.NUMEVENT takes the value 0b00.
- Each event selector behaves in a specific way:
  - TRCEVENTCTL1R.INSTEN and TRCEVENTCTL1R.DATAEN are not implemented because no events are implemented.
  - If TRCTSCTLR is implemented, TRCTSCTLR.EVENT behaves as if the FALSE event is selected, and the field is RES0.
  - If TRCVDCTLR is implemented, TRCVDCTLR.EVENT behaves as if the TRUE event is selected. Bits[7:1] are RES0 and bit[0] is RES1.
  - TRCEVENTCTL0R is not implemented because no events are implemented.
  - TRCCNTCTLRn.CNTEVENT and TRCCNTCTLRn.RLDEVENT are not implemented because no counters are implemented.
  - The TRCSEQEVRn are not implemented because no sequencer states are implemented.

Note

TRCTSCTLR is implemented whenever TRCIDR0.TSSIZE indicates that global timestamping is implemented, regardless of the value of TRCIDR4.NUMRSPAIR.

0b0001	The implementation has two resource selection pairs.
0b0010	The implementation has three resource selection pairs.
.	.
.	.
.	.
0b1111	The implementation has 16 resource selection pairs.

Implementations always have at least one resource selection pair so that they can support the FALSE and TRUE resource selectors, that is, 0 and 1.

NUMPC, bits[15:12] Indicates the number of PE comparator inputs that are available for tracing. The permitted values are:

0b0000	No PE comparator inputs are available.
0b0001	The implementation has one PE comparator input.
0b0010	The implementation has two PE comparator inputs.

. .
. .
. .

`0b1000` The implementation has eight PE comparator inputs.

All other values are reserved.

Bits[11:9] RES0.

SUPPDAC, bit[8] Indicates if the implementation can support data address comparisons:

0 The implementation does not support data address comparisons.

1 The implementation can support data address comparisons.

NUMDVC, bits[7:4] Indicates the number of data value comparators that are available for tracing. The permitted values are:

`0b0000` No data value comparators are available.

`0b0001` The implementation has one data value comparator.

`0b0010` The implementation has two data value comparators.

. .
. .
. .

`0b1000` The implementation has eight data value comparators.

For ETMv4.3 or later, if NUMRSPAIR is `0b0000`, this field takes the value `0b0000`.

All other values are reserved.

NUMACPAIRS, bits[3:0]

Indicates the number of address comparator pairs that are available for tracing. The permitted values are:

`0b0000` No address comparator pairs are available.

`0b0001` The implementation has one address comparator pair.

`0b0010` The implementation has two address comparator pairs.

. .
. .
. .

`0b1000` The implementation has eight address comparator pairs.

All other values are reserved.

7.3.35 TRCIDR5, ID Register 5

The TRCIDR5 characteristics are:

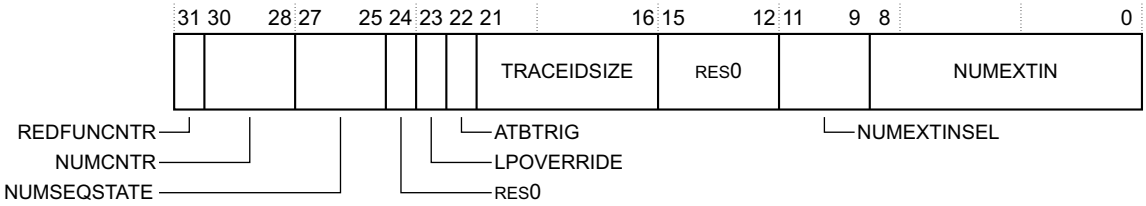
Purpose Returns how many resources the trace unit supports.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR5 bit assignments are:



REDFUNCNTR, bit[31]

Indicates if the reduced function counter is implemented:

0 The reduced function counter is not supported.

1 Counter 0 is implemented as a reduced function counter.

NUMCNTR, bits[30:28]

Indicates the number of counters that are available for tracing. The permitted values are:

`0b000` No counters are available.

`0b001` The implementation has one counter.

`0b010` The implementation has two counters.

`0b011` The implementation has three counters.

`0b100` The implementation has four counters.

All other values are Reserved.

For ETMv4.3 or later, if TRCIDR4.NUMRSPAIR is `0b0000`, this field takes the value `0b000`.

NUMSEQSTATE, bits[27:25]

Indicates the number of sequencer states that are implemented. The permitted values are:

`0b000` No sequencer states are implemented, and TRCSEQEVR0-TRCSEQEVR2 are not implemented.

`0b100` The implementation has four sequencer states, and TRCSEQEVR0-TRCSEQEVR2 are implemented.

All other values are Reserved.

For ETMv4.3 or later, if TRCIDR4.NUMRSPAIR is `0b0000`, this field takes the value `0b000`.

Bit[24] RES0.

LPOVERRIDE, bit[23]

Indicates if the implementation can support low-power state override:

0 The implementation does not support low-power state override.

1 The implementation supports low-power state override, and the TRCEVENTCTL1R.LPOVERRIDE field is implemented.

The trace unit must support low-power state override if it can enter a low-power mode where the resources and event trace generation are disabled.

ATBTRIG, bit[22] Indicates if the implementation can support ATB triggers:

0 The implementation does not support ATB triggers.

1 The implementation supports ATB triggers, and the TRCEVENTCTL1R.ATBTRIG field is implemented.

TRACEIDSIZE, bits[21:16]

Indicates the trace ID width. The permitted value is:

`0x07` The implementation supports a 7-bit trace ID. This defines the width of the TRCTRACEIDR.TRACEID field.

All other values are reserved.

Note

The CoreSight ATB requires a 7-bit trace ID width.

Bits[15:12] RES0.

NUMEXTINSEL, bits[11:9]

Indicates how many external input select resources are implemented. The permitted values are:

0b000	No external input select resources are available and TRCEXTINSELR is not implemented.
0b001	The implementation has one external input select resource.
0b010	The implementation has two external input select resources.
0b011	The implementation has three external input select resources.
0b100	The implementation has four external input select resources.

All other values are reserved.

For ETMv4.3 or later, if TRCIDR4.NUMRSPAIR is 0b0000, this field takes the value 0b000.

The number of external input selectors, defined by TRCIDR5.NUMEXTINSEL must be equal to or less than the number of external inputs defined by TRCIDR5.NUMEXTIN.

See TRCEXTINSELR for how to select an input select resource.

NUMEXTIN, bits[8:0]

Indicates how many external inputs are implemented. The permitted values are:

0b000000000	No external inputs are available. If NUMEXTIN is zero, NUMEXTINSEL must also be zero.
0b000000001	The implementation has one external input.
0b000000010	The implementation has two external inputs.
.	.
.	.
.	.
0b100000000	The implementation has 256 external inputs.

All other values > 0b100000000 are reserved.

For ETMv4.3 or later, if TRCIDR4.NUMRSPAIR is 0b0000, this field takes the value 0b000000000.

If TRCIDR5.NUMEXTIN greater than four, then at least one external input selector must be implemented.

If TRCIDR5.NUMEXTINSEL is zero and TRCIDR5.NUMEXTIN is not zero, TRCIDR5.NUMEXTIN must be less than or equal to four and the external inputs are flat-mapped through to the external input selector resources. This means that any resource selector programmed to select one of the external input selectors directly selects the external input specified by the SELECT field of the resource selector. For example, a resource selector with GROUP = 0b0000 and SELECT = 0x0002 selects external input 1. The valid values which can be programmed into the SELECT field are defined by TRCIDR5.NUMEXTIN.

See TRCEXTINSELR for how to select an external input.

7.3.36 TRCIDR6, ID Register 6

The TRCIDR6 characteristics are:

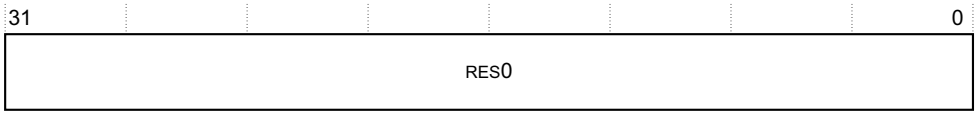
Purpose Returns zero. Register is reserved.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register that returns RES0. See also *Register summary on page 7-336*.

The TRCIDR6 bit assignments are:



Bits[31:0] RES0.

7.3.37 TRCIDR7, ID Register 7

The TRCIDR7 characteristics are:

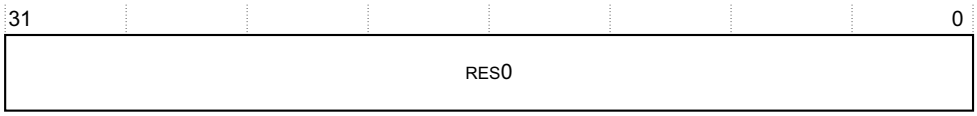
Purpose Returns zero. Register is reserved.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register that returns RES0. See also *Register summary on page 7-336*.

The TRCIDR7 bit assignments are:



Bits[31:0] RES0.

7.3.38 TRCIDR8, ID Register 8

The TRCIDR8 characteristics are:

Purpose Returns the maximum speculation depth of the instruction trace stream.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR8 bit assignments are:

MAXSPEC, bits[31:0]

Indicates the maximum speculation depth of the instruction trace stream. This is the maximum number of P0 elements in the trace stream that can be speculative at any time.

7.3.39 TRCIDR9, ID Register 9

The TRCIDR9 characteristics are:

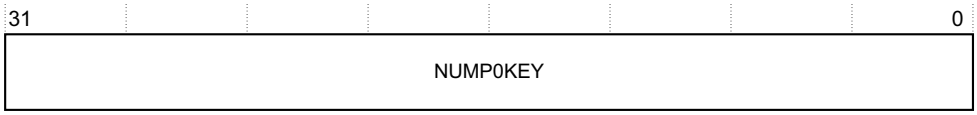
Purpose Returns the number of P0 right-hand keys that the trace unit can use.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR9 bit assignments are:



NUMP0KEY, bits[31:0]

Indicates the number of P0 right-hand keys that the trace unit can use. A value of 0 or 1 indicates one P0 key.

The value of this bit can be less than the value of TRCIDR8.MAXSPEC.

Arm recommends a minimum of 32 P0 keys for an implementation that supports data tracing. If TRCIDR9.NUMP0KEY < 32 this can result in a large number of data synchronization markers in the trace stream.

7.3.40 TRCIDR10, ID Register 10

The TRCIDR10 characteristics are:

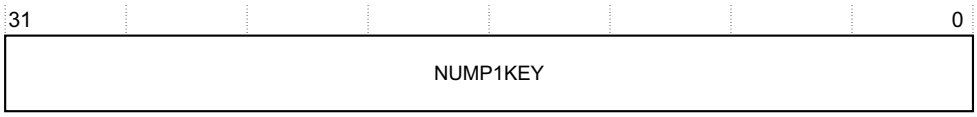
Purpose Returns the number of P1 right-hand keys that the trace unit can use.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR10 bit assignments are:



NUMP1KEY, bits[31:0]

Indicates the number of P1 right-hand keys that the trace unit can use. The number includes normal and special keys.

7.3.41 TRCIDR11, ID Register 11

The TRCIDR11 characteristics are:

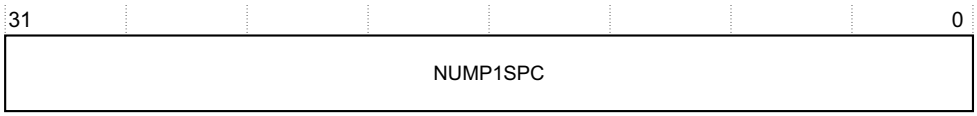
Purpose Returns the number of special P1 right-hand keys that the trace unit can use.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR11 bit assignments are:



NUMP1SPC, bits[31:0]

Indicates the number of special P1 right-hand keys that the trace unit can use.

7.3.42 TRCIDR12, ID Register 12

The TRCIDR12 characteristics are:

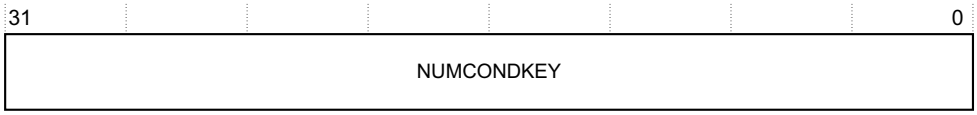
Purpose Returns the number of conditional instruction right-hand keys that the trace unit can use.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR12 bit assignments are:



NUMCONDKEY, bits[31:0]

Indicates the number of conditional instruction right-hand keys that the trace unit can use. The number includes normal and special keys.

7.3.43 TRCIDR13, ID Register 13

The TRCIDR13 characteristics are:

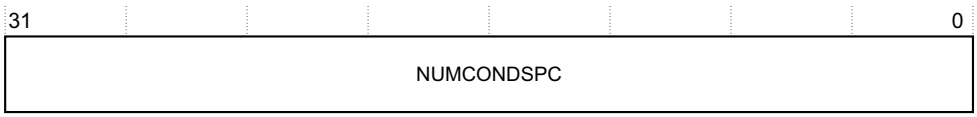
Purpose Returns the number of special conditional instruction right-hand keys that the trace unit can use.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register with an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCIDR13 bit assignments are:



NUMCONDSPC, bits[31:0]

Indicates the number of special conditional instruction right-hand keys that the trace unit can use.

7.3.44 TRCIMSPEC0, Implementation Defined register 0

The TRCIMSPEC0 characteristics are:

Purpose The TRCIMSPEC*n* registers are reserved for the future implementation of up to eight IMPLEMENTATION DEFINED registers. When a trace unit does not implement these registers, TRCIMSPEC0 must still be partially implemented so that a debugger can implement a general mechanism for detecting the IMPLEMENTATION DEFINED registers.

TRCIMSPEC0 shows the presence of any IMPLEMENTATION DEFINED features, and provides an interface to enable the features that are provided.

Note

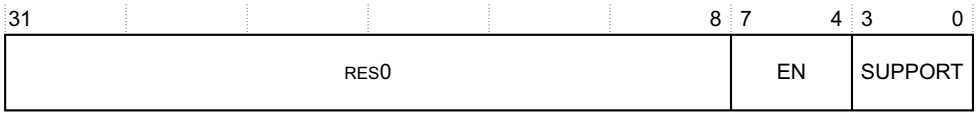
IMPLEMENTATION DEFINED registers 1-7 are defined by the implementation. See *TRCIMSPECn, Implementation Defined registers, n=1-7*.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RW trace register with an IMPLEMENTATION DEFINED reset value. This register is reset by a trace unit reset. See also *Register summary on page 7-336*.

The TRCIMSPEC0 bit assignments are:



Bits[31:8] RES0.

EN, bits[7:4]

If SUPPORT > 0, this field controls whether the IMPLEMENTATION DEFINED features are enabled. The permitted values are:

`0b0000`	The IMPLEMENTATION DEFINED features are not enabled. The trace unit must behave as if the IMPLEMENTATION DEFINED features are not supported.
`0b0001-0b1111`	The trace unit behavior is IMPLEMENTATION DEFINED.

This field is set to `0b0000` on a trace unit reset.

If SUPPORT==0, this field is RES0.

SUPPORT, bits[3:0] Indicates whether the implementation supports IMPLEMENTATION DEFINED features. This field is read-only. The permitted values are:

`0b0000`	No IMPLEMENTATION DEFINED features are supported. The EN field is RES0.
`0b0001-0b1111`	IMPLEMENTATION DEFINED features are supported. Use of these values requires written permission from Arm.

7.3.45 TRCIMSPECn, Implementation Defined registers, n=1-7

These trace registers might return information that is specific to an implementation, or enable features specific to an implementation to be programmed. The product *Technical Reference Manual* describes these registers.

7.3.46 TRCITCTRL, Integration Mode Control register

The TRCITCTRL characteristics are:

Purpose Controls whether the trace unit is in integration mode.

Usage constraints

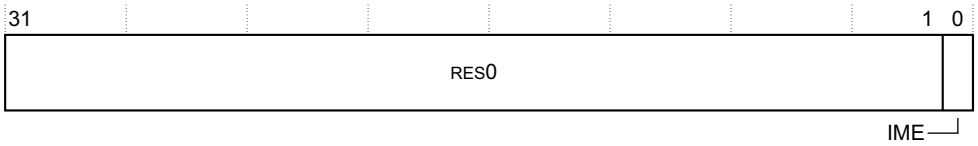
- Accessible only from the memory-mapped interface or from an external agent such as a debugger.
- It is IMPLEMENTATION DEFINED whether an implementation permits accesses using the memory-mapped interface or debug interface when any of the following apply:
  - TRCPDSR.POWER==0, the trace unit core power domain is powered down.
  - TRCPDSR.OSLK==1, the OS Lock is locked.
- UNPREDICTABLE behavior occurs if the IME bit is set to 1 when any of the following apply:
  - TRCPRGCTLR.EN==1, the trace unit is enabled.
  - TRCSTATR.IDLE==0, the trace unit is not idle.

  In these scenarios, the trace unit might generate incorrect or corrupt trace and the trace unit resources might behave unexpectedly.
- If the IME bit changes from one to zero then Arm recommends that the trace unit is reset. Otherwise the trace unit might generate incorrect or corrupt trace and the trace unit resources might behave unexpectedly.

Configurations Available in all implementations.

Attributes A 32-bit RW management register. The register is reset to zero but it is IMPLEMENTATION DEFINED whether it is reset by a trace unit reset or an external trace reset. See also *Register summary on page 7-336*.

The TRCITCTRL bit assignments are:



Bits[31:1] RES0.

IME, bit[0] Integration mode enable bit:

0 The trace unit is not in integration mode.

1 The trace unit is in integration mode. This mode enables:

- A debug agent to perform topology detection.
- *System-on-Chip* (SoC) test software to perform integration testing.

It is IMPLEMENTATION DEFINED whether this register is reset by a trace unit reset or an external trace reset. In either case, it is reset to zero.

If no topology detection or integration functionality is implemented, this field can be RES0.

7.3.47 TRCLAR, Software Lock Access Register

The TRCLAR characteristics are:

Purpose Controls whether the Software Lock is locked. When the Software Lock is implemented and locked, the trace unit:

- Ignores write accesses from the memory-mapped interface, to all trace unit registers other than the TRCLAR.
- Does not change the TRCPDSR.STICKYPD bit if a read access occurs, on the memory-mapped interface, to the TRCPDSR.

The Software Lock has no effect on accesses from an external debug agent or from system instructions.

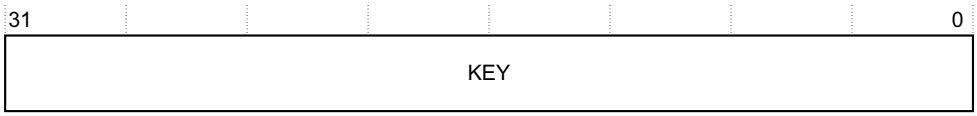
Arm deprecates the implementation of the Software Lock.

Usage constraints Accessible only from the memory-mapped interface.

Configurations Implemented only when the trace unit supports accesses from the memory-mapped interface.

Attributes A 32-bit WO management register. See also *Register summary on page 7-336*.

The TRCLAR bit assignments are:



KEY, bits[31:0] The trace unit unlocks the Software Lock when software writes `0xC5ACCE55` to this field. If software writes any other value to this field then the trace unit locks the Software Lock.

Software can use the Software Lock to prevent accidental modification of the trace unit registers by software being debugged. For example, software that accidentally initializes an incorrect region of memory might disable the trace unit and make it impossible to trace the software. To prevent this, on-chip software that uses the memory-mapped interface to access the trace unit must access the trace unit registers as follows:

1. Write `0xC5ACCE55` to the TRCLAR, to unlock the Software Lock.
2. Perform the required accesses to the trace unit registers.
3. Write to the TRCLAR with any value except `0xC5ACCE55`, such as `0x0`, to lock the Software Lock.

7.3.48 TRCLSR, Software Lock Status Register

The TRCLSR characteristics are:

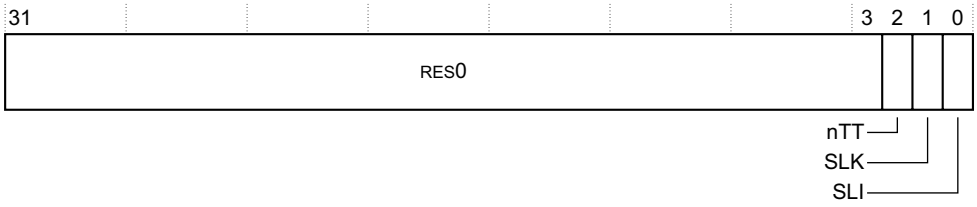
Purpose Software can use the memory-mapped interface to read this register and discover if the Software Lock is implemented and if so whether it is locked. Arm deprecates the implementation of the Software Lock.

Usage constraints Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Implemented only when the trace unit supports accesses from the memory-mapped interface.

Attributes A 32-bit RO management register. The register is set to an IMPLEMENTATION DEFINED value on an external trace reset. See also *Register summary on page 7-336*.

The TRCLSR bit assignments are:



Bits[31:3] RES0.

nTT, bit[2] Indicates that the TRCLAR is a 32-bit register:

RES0 If TRCLSR.SLI==0.

RAZ If TRCLSR.SLI==1.

SLK, bit[1] When TRCLSR.SLI ==1, this bit returns the Software Lock status:

0 Software Lock is unlocked.

1 Software Lock is locked.

This bit is set to 1 on an external trace reset.

When TRCLSR.SLI==0, this bit is RES0.

SLI, bit[0] For a read access from the memory-mapped interface, this bit indicates if the trace unit implements a Software Lock for accesses from the memory-mapped interface:

0 The Software Lock is not implemented.

1 The Software Lock is implemented.

Note

Arm deprecates the implementation of the Software Lock and recommends that this bit is 0.

For a read access from the external debugger interface, this bit is RAZ.

For Armv8-A PEs, if any PE debug components implements the Armv8.4 separate Secure and Non-secure views of that component, then implementation of the Software Lock is prohibited.

7.3.49 TRCOSLAR, OS Lock Access Register

The TRCOSLAR characteristics are:

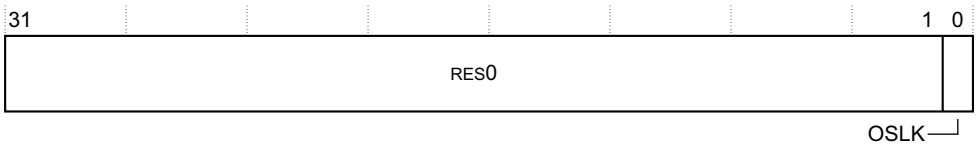
Purpose Controls whether the OS Lock is locked.

Usage constraints There are no usage constraints.

Configurations Always implemented if the OS Lock is implemented, that is whenTRCOSLSR.OSLM is not `0b00`.

Attributes A 32-bit WO management register. See also *Register summary on page 7-336*.

The TRCOSLAR bit assignments are:



Bits[31:1] RES0.

OSLK, bit[0] OS Lock control bit:

0 Unlocks the OS Lock.

1 Locks the OS Lock. This setting disables the trace unit. See *Trace unit behavior when the trace unit is disabled on page 3-100* for more information.

7.3.50 TRCOSLSR, OS Lock Status Register

The TRCOSLSR characteristics are:

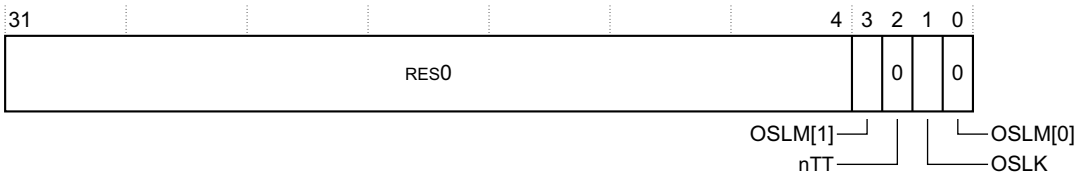
Purpose Returns the status of the OS Lock.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register is set to `0xA` on a trace unit reset. See also *Register summary on page 7-336*.

The TRCOSLSR bit assignments are:



Bits[31:4] RES0.

OSLM, bits[3, 0] OS Lock model field. This field indicates the OS Lock model is implemented.

- For Armv7-A, Armv7-R, Armv8-A and Armv8-R PEs these bits are always `0b10` to indicate that the OS Lock is implemented
- For Armv6-M, Armv7-M, and Armv8-M PEs these bits can take the following values:

  `0b10` OS Lock is implemented. TRCOSLSR is implemented and TRCOSLSR.OSLK and TRCPDSR.OSLK indicate the current state of the OS Lock.

  `0b00` OS Lock is not implemented. TRCOSLAR is not implemented, and TRCOSLSR.OSLK and TRCPDSR.OSLK are RES0.

Note

Arm recommends the OS Lock is not implemented on trace units for Armv6-M, Armv7-M, and Armv8-M PEs.

nTT, bit[2] This bit is RAZ, which indicates that software must perform a 32-bit write to update the TRCOSLAR.

OSLK, bit[1] OS Lock status bit:

0 The OS Lock is unlocked.

1 The OS Lock is locked.

If the OS Lock is implemented, the reset value is 1.

If the OS Lock is not implemented this bit is RES0.

When the trace unit core power domain is powered down the value is UNKNOWN. TRCPDSR indicates if the trace unit core power domain is powered down.

7.3.51 TRCPDCR, PowerDown Control Register

The TRCPDCR characteristics are:

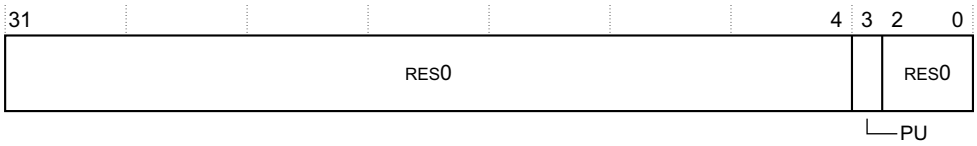
Purpose Requests the system to provide power to the trace unit.

Usage constraints Accessible only from the memory-mapped interface or from an external agent such as a debugger.

Configurations Available in all implementations.

Attributes A 32-bit RW management register. The register is set to zero on an external trace reset. See also *Register summary on page 7-336*.

The TRCPDCR bit assignments are:



Bits[31:4] RES0.

PU, bit[3] Powerup request bit:

0 The system can remove power from the trace unit. The TRCPDSR indicates if the trace unit is powered down.

1 The system must provide power to the trace unit.

The reset value is 0.

Note

Typically, a trace unit drives a signal representing the value of this bit to a power controller to request that the trace unit core power domain is powered up. However, if the trace unit and the PE are in the same power domain then the implementation might combine the PU status with a signal from the PE.

When the Unified Power Domain Model is implemented, TRCPDCR.PU is RES0.

Bits[2:0] RES0.

7.3.52 TRCPDSR, PowerDown Status Register

The TRCPDSR characteristics are:

Purpose Returns the following information about the trace unit:

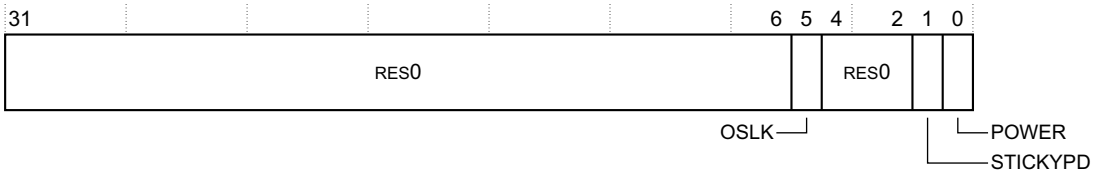
- OS Lock status.
- Core power domain status.
- Power interruption status.

Usage constraints Accessible only from the memory-mapped interface or from an external agent such as a debugger.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. See also *Register summary on page 7-336*.

The TRCPDSR bit assignments are:



Bits[31:6] RES0.

OSLK, bit[5] OS Lock status bit:

0 The OS Lock is unlocked.

1 The OS Lock is locked.

This field is reset to 1 on a trace unit reset.

The value is UNKNOWN when the trace unit core power domain is powered down, that is, when POWER==0.

If the OS Lock is not implemented, this bit is RES0.

Bits[4:2] RES0.

STICKYPD, bit[1] Sticky powerdown status bit. Indicates whether the trace register state is valid:

0 If POWER==1 then the state of TRCOSLSR and the trace registers are valid.

If POWER==0 then it is UNKNOWN whether the state of TRCOSLSR and the trace registers are valid.

1 The state of TRCOSLSR and the trace registers might not be valid.

The trace unit sets this bit to 1 if either:

- The trace unit is reset.
- The power to the trace unit core power domain is removed and the trace register state is not valid.

The STICKYPD field is read-sensitive. On a read of the TRCPDSR, this field is cleared to 0 after the register has been read, unless one of the following applies:

- The access is a memory-mapped access and the Software Lock is implemented and locked.
- The trace unit core power domain is powered down.

The TRCLAR controls whether the Software Lock is locked.

This field is reset to 1 on a trace unit reset.

POWER, bit[0] Power status bit:

0 The trace unit core power domain is not powered. The trace registers are not accessible and they all return an error response.

1 The trace unit core power domain is powered. The trace registers are accessible.

When the Unified Power Domain Model is implemented, TRCPDSR.POWER is RAO.

Table 7-7 shows how to interpret the TRCPDSR[1:0] bits.

Table 7-7 TRCPDSR[1:0] encodings

STICKYPD bit	POWER bit	Meaning
0	0	The trace unit core power domain is not powered so the trace registers are inaccessible. The trace register state might not be valid. When POWER==0 if: • The trace register state is valid, when power is restored and POWER==1 then the implementation must restore the true value for STICKYPD. • The trace register state is not valid, when power is restored and POWER==1 then the implementation must set STICKYPD=1. This permits an implementation to indicate STICKYPD==0 when trace unit core power is removed, if it is unclear whether the trace register state is lost. Arm recommends that if the trace register state is lost then an implementation sets STICKYPD=1 when POWER==0. This encoding is also used for trace units which support a retention state for the trace unit core power domain, where power is removed from the trace unit core power domain but trace unit state is preserved through the power down. On leaving a retention state, the trace register state is still valid and the value of STICKYPD is restored from before the power down.
0	1	The trace unit core power domain is powered and the trace registers are accessible.
1	0	The trace unit core power domain is not powered so the trace registers are inaccessible. The register state is not valid although prior to the power removal it might have been valid.
1	1	The trace unit core power domain is powered and the trace registers are accessible. A trace unit reset or power interruption has occurred so the trace register state might not be valid.

7.3.53 TRCPIDR0, Peripheral ID0 Register

The TRCPIDR0 characteristics are:

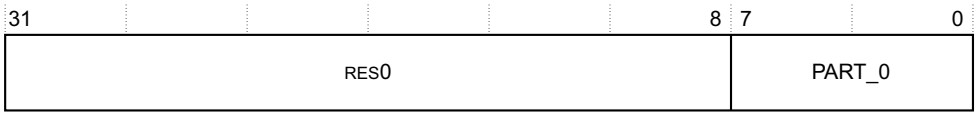
Purpose Returns information that helps identify the peripheral. If software reads the TRCPIDR[7:0] register group then it can determine the 64-bit CoreSight Peripheral ID for the trace unit.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register has an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCPIDR0 bit assignments are:



Bits[31:8] RES0.

PART_0, bits[7:0] Bits[7:0] of the IMPLEMENTATION DEFINED part number. TRCPIDR1.PART_1 provides the remaining four bits that comprise the part number identifier.

7.3.54 TRCPIDR1, Peripheral ID1 Register

The TRCPIDR1 characteristics are:

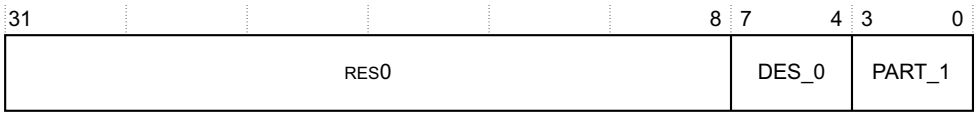
Purpose Returns information that helps identify the peripheral. If software reads the TRCPIDR[7:0] register group then it can determine the 64-bit CoreSight Peripheral ID for the trace unit.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register has an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCPIDR1 bit assignments are:



Bits[31:8] RES0.

DES_0, bits[7:4] Bits[3:0] of the IMPLEMENTATION DEFINED JEP106 identification code. TRCPIDR2.DES_1 and TRCPIDR4.DES_2 provide the additional bits that enables software to determine the designer identifier.

For an implementation designed by Arm the JEP106 identification code is `0x3B` and therefore this field is `0xB`.

PART_1, bits[3:0] Bits[11:8] of the IMPLEMENTATION DEFINED part number. TRCPIDR0.PART_0 provides the remaining eight bits that comprise the part number identifier.

7.3.55 TRCPIDR2, Peripheral ID2 Register

The TRCPIDR2 characteristics are:

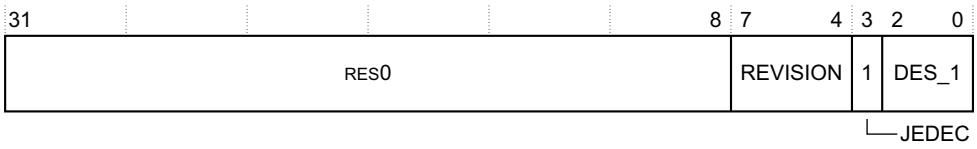
Purpose Returns information that helps identify the peripheral. If software reads the TRCPIDR[7:0] register group then it can determine the 64-bit CoreSight Peripheral ID for the trace unit.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register has an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCPIDR2 bit assignments are:



Bits[31:8] RES0.

REVISION, bits[7:4] The IMPLEMENTATION DEFINED revision number for the trace unit implementation. See also TRCIDR1.REVISION.

JEDEC, bit[3] RAO. Indicates that the TRCPIDR1[7:4], TRCPIDR2[6:4] and TRCPIDR4.DES_2 fields represent a JEP106 identification code.

DES_1, bits[2:0] Bits[6:4] of the IMPLEMENTATION DEFINED JEP106 identification code. TRCPIDR1.DES_0 and TRCPIDR4.DES_2 provide the additional bits that enables software to determine the designer identifier.

For an implementation designed by Arm the JEP106 identification code is `0x3B` and therefore this field is `0b011`.

7.3.56 TRCPIDR3, Peripheral ID3 Register

The TRCPIDR3 characteristics are:

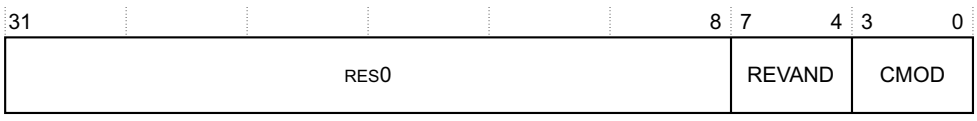
Purpose Returns information that helps identify the peripheral. If software reads the TRCPIDR[7:0] register group then it can determine the 64-bit CoreSight Peripheral ID for the trace unit.

Usage constraints
- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register has an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCPIDR3 bit assignments are:



Bits[31:8] RES0.

REVAND, bits[7:4] The IMPLEMENTATION DEFINED manufacturing revision number for the implementation. After silicon is available, if metal fixes are necessary, the manufacturer can alter the top metal layer so that this field can indicate any post-fabrication silicon changes.

CMOD, bits[3:0] An IMPLEMENTATION DEFINED value that indicates an endorsed modification to the implementation.

If the system designer cannot modify the implementation supplied by the PE designer then this field is RES0.

7.3.57 TRCPIDR4, Peripheral ID4 Register

The TRCPIDR4 characteristics are:

Purpose Returns information that helps identify the peripheral. If software reads the TRCPIDR[7:0] register group then it can determine the 64-bit CoreSight Peripheral ID for the trace unit.

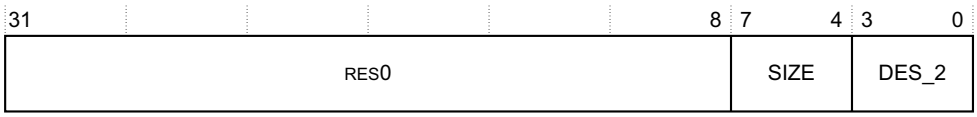
Usage constraints

- Only bits[7:0] are valid.
- Accessible only from the memory-mapped interface or the external debugger interface.

Configurations Available in all implementations.

Attributes A 32-bit RO management register. The register has an IMPLEMENTATION DEFINED value. See also *Register summary on page 7-336*.

The TRCPIDR4 bit assignments are:



Bits[31:8] RES0.

SIZE, bits[7:4] RES0. This indicates that the trace unit memory map occupies 4KB.

DES_2, bits[3:0] The IMPLEMENTATION DEFINED JEP106 continuation code. TRCPIDR1.DES_0 and TRCPIDR2.DES_1 provide the additional bits that enables software to determine the designer identifier.

For an implementation designed by Arm this field is `0x4`.

7.3.58 TRCPIDR5, TRCPIDR6, TRCPIDR7, Peripheral ID5 to Peripheral ID7 Registers

The characteristics for TRCPIDR[5,6,7] are:

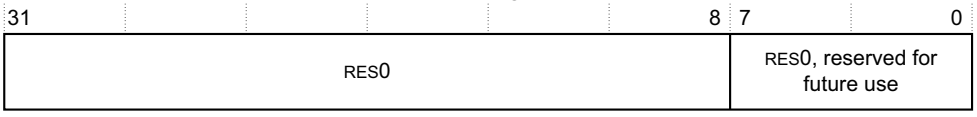
Purpose Reserved for future expansion of the CoreSight peripheral identification information.

Usage constraints These registers are unused.

Configurations These registers are defined as reserved registers.

Attributes A 32-bit RO management register. These registers have a value of zero. See also *Register summary on page 7-336*.

The TRCPIDR5,TRCPIDR6,and TRCPIDR7 bit assignments are:



Bits[31:8] RES0.

Bits[7:0] RES0, reserved for future use.

7.3.59 TRCPRGCTLR, Programming Control Register

The TRCPRGCTLR characteristics are:

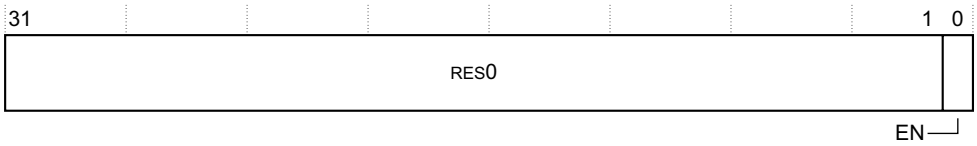
Purpose Enables the trace unit.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RW trace register. The register is set to zero on a trace unit reset. See also *Register summary on page 7-336*.

The TRCPRGCTLR bit assignments are:



Bits[31:1] RES0.

EN, bit[0] Trace unit enable bit:

0 The trace unit is disabled. All trace resources are inactive and no trace is generated. See *Trace unit behavior when the trace unit is disabled on page 3-100*.

1 The trace unit is enabled. See *Trace unit behavior when the trace unit is enabled on page 3-100*.

On a trace unit reset the value is 0.

7.3.60 TRCPROCSELR, Processing Element Select Control Register

The TRCPROCSELR characteristics are:

Purpose Controls which PE to trace.

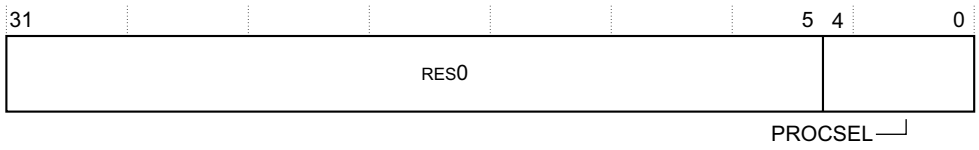
Usage constraints

- Might ignore writes when the trace unit is enabled or not idle.
- Before writing to this register, ensure that TRCSTATR.IDLE==1 so that the trace unit can synchronize with the chosen PE.

Configurations Implemented if TRCIDR3.NUMPROC is greater than zero.

Attributes A 32-bit RW trace register. The register is set to zero on a trace unit reset. See also *Register summary on page 7-336*.

The TRCPROCSELR bit assignments are:



Bits[31:3] RES0.

PROCSEL, bits[4:0] PE select bits that select the PE to trace. Writes that are:

- ≤ TRCIDR3.NUMPROC select the PE to trace.

- > TRCIDR3.NUMPROC causes UNPREDICTABLE behavior, such as any of:
  - No PE is traced.
  - It is not predictable which PE is traced.
  - TRCPROCSELR.PROCSEL returns UNKNOWN.

The implemented width of this field is dependent on the value of TRCIDR3.NUMPROC, meaning that the field might be smaller than three bits. Unimplemented bits are RES0. If TRCIDR3.NUMPROC is zero, indicating that one PE is supported, then TRCPROCSELR is not implemented.

7.3.61 TRCQCTLR, Q Element Control Register

The TRCQCTLR characteristics are:

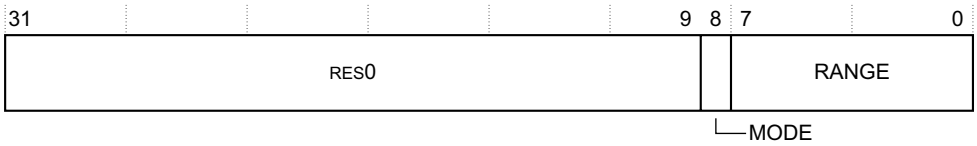
Purpose Controls when Q elements are enabled.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- Only bits[8:0] are valid.
- This register must be programmed if it is implemented and TRCCONFIGR.QE is set to any value other than `0b00`.

Configurations TRCIDR0.QFILT indicates if this register is implemented.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCQCTLR bit assignments are:



Bits[31:9] RES0

MODE, bit[8] Selects whether the address range comparators selected by the RANGE field indicate address ranges where the trace unit is permitted to generate Q elements or address ranges where the trace unit is not permitted to generate Q elements:

0 Exclude mode. The address range comparators selected by the RANGE field indicate address ranges where the trace unit cannot generate Q elements. If no ranges are selected, Q elements are permitted across the entire memory map.

1 Include mode. The address range comparators selected by the RANGE field indicate address ranges where the trace unit can generate Q elements. If all the implemented bits in RANGE are set to 0 then Q elements are disabled.

RANGE, bits[7:0] Specifies the address range comparators to be used for controlling Q elements.

One bit is provided for each implemented address range comparator. If this bit is set to 1, then the address range comparator indicated by that bit is selected for use. For example, if bit[0] is set to 1, then address range comparator 0 is selected for use.

The implemented width of the RANGE field is defined by TRCIDR4.NUMACPAIRS. Unimplemented bits in the RANGE field are RES0.

7.3.62 TRCRSCTLRn, Resource Selection Control Registers, n=2-31

The TRCRSCTLRn characteristics are:

Purpose Controls the selection of the resources in the trace unit.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.

- If software selects a non-implemented resource then CONSTRAINED UNPREDICTABLE behavior of the resource selector occurs, so the resource selector might fire unexpectedly or might not fire. Reads of the TRCRSCTLRn might return UNKNOWN.

Configurations The TRCIDR4.NUMRSPAIR field sets the value of n and therefore controls how many TRCRSCTLRs are implemented. Resource selectors are implemented in pairs. Each odd numbered resource selector is part of a pair with the even numbered resource selector that is numbered as one less than it. For example, resource selectors 2 and 3 form a pair.

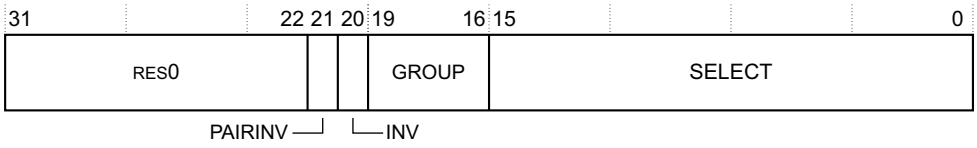
The minimum implementation of resource selector pairs depends on the ETM architecture version:

- For ETMv4.2 or earlier, resource selector pair 0 is always implemented and is Reserved.
- For ETMv4.3 or later, if TRCIDR4.NUMRSPAIR is `0b0000`, no resource selectors are implemented and the TRUE or FALSE resource selectors are not available. For all other values of TRCIDR4.NUMRSPAIR, the behavior is the same as for ETMv4.2 or earlier.

If implemented, resource selector 0 always returns FALSE, and resource selector 1 always returns TRUE.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCRSCTLRn bit assignments are:



Bits[31:22] RES0.

PAIRINV, bit[21]

For TRCRSCTLRn where n=2, 4, 6, 8, ..., or 30, this bit controls whether the combined result from a resource pair is inverted:

0 The combined result is not inverted.

1 The combined result is inverted.

For TRCRSCTLRn where n=3, 5, 7, 9, ..., 31, this bit is RES0.

INV, bit[20] Controls whether the resource, that GROUP and SELECT selects, is inverted:

0 The selected resource is not inverted.

1 The selected resource is inverted.

GROUP, bits[19:16] Selects a group of resources. See Table 7-8 on page 7-400 for how to select a group and a resource.

It is IMPLEMENTATION DEFINED whether all of the bits are supported.

SELECT, bits[15:0] Selects one or more resources from the group that the GROUP field selects. Each bit represents a resource from the selected group. See Table 7-8 on page 7-400 for how to select a group and a resource.

It is IMPLEMENTATION DEFINED whether all of the bits are supported.

Table 7-8 lists which resources are selected depending on the values of the GROUP and SELECT fields.

Table 7-8 Resource selection with the GROUP and SELECT fields

GROUP	SELECT	Resource
0b0000	0-3	External input selector 0-3
	4-15	Reserved
0b0001	0-7	PE comparator inputs 0-7
	8-15	Reserved
0b0010	0-3	Counter at zero 0-3
	4-7	Sequencer states 0-3
	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 ID comparator 0-7
	8-15	Reserved
0b0111	0-7	Virtual context identifier comparator 0-7
	8-15	Reserved
0b1000 - 0b1111	0-15	Reserved

7.3.63 TRCSEQEVRn, Sequencer State Transition Control Registers, n=0-2

The TRCSEQEVRn characteristics are:

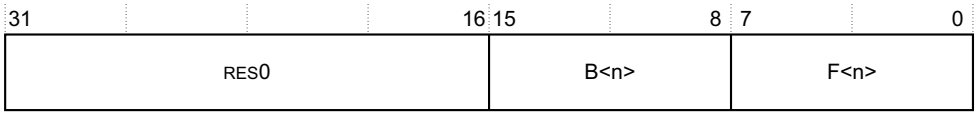
Purpose Moves the sequencer state:
- Backwards, from state n+1 to state n when a programmed event occurs.
- Forwards, from state n to state n+1 when a programmed event occurs.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- When the sequencer is used, all sequencer state transitions must be programmed with a valid event.

Configurations
- The number of TRCSEQEVRs is IMPLEMENTATION DEFINED and is indicated by TRCIDR5.NUMSEQSTATE.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCSEQEVRn bit assignments are:



Bits[31:16] RES0.

B<n>, bits[15:8] Backward field. An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes. When the event occurs then the sequencer state moves from state n+1 to state n. For example, for TRCSEQEVR2, if B2==0x14 then when event 0x14 occurs, the sequencer moves from state 3 to state 2.

F<n>, bits[7:0] Forward field. An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes. When the event occurs then the sequencer state moves from state n to state n+1. For example, for TRCSEQEVR1, if F1==0x12 then when event 0x12 occurs, the sequencer moves from state 1 to state 2.

7.3.64 TRCSEQRSTEVR, Sequencer Reset Control Register

The TRCSEQRSTEVR characteristics are:

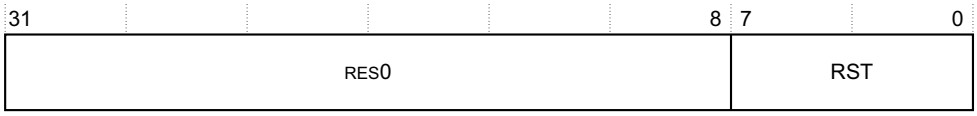
Purpose Moves the sequencer to state 0 when a programmed event occurs.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- When the sequencer is used, all sequencer state transitions must be programmed with a valid event.

Configurations Only implemented when TRCIDR5.NUMSEQSTATE > 0.

Attributes A 32-bit RW trace register. This unit is set to an UNKNOWN value on trace unit reset. See also *Register summary on page 7-336*.

The TRCSEQRSTEVR bit assignments are:



Bits[31:8] RES0.

RST, bits[7:0] An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes. When the event occurs then the sequencer state moves to state 0.

7.3.65 TRCSEQSTR, Sequencer State Register

The TRCSEQSTR characteristics are:

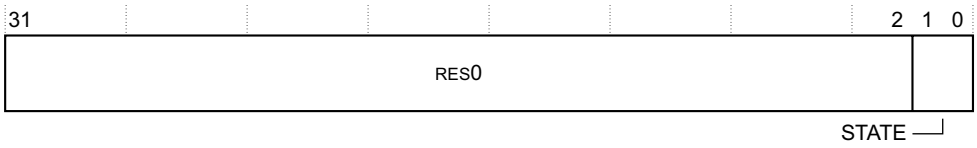
Purpose Use this to set, or read, the sequencer state.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- Only returns stable data when TRCSTATR.PMSTABLE==1.
- Software must use this register to set the initial state of the sequencer before the sequencer is used.

Configurations Only implemented when TRCIDR5.NUMSEQSTATE > 0.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCSEQSTR bit assignments are:



Bits[31:2] RES0.

STATE, bits[1:0] Sets or returns the state of the sequencer:

- 0b00 State 0.
- 0b01 State 1.
- 0b10 State 2.
- 0b11 State 3.

7.3.66 TRCSSCCRn, Single-shot Comparator Control Registers, n=0-7

The TRCSSCCRn characteristics are:

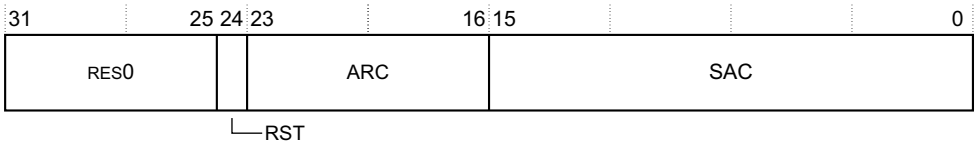
Purpose Controls the corresponding single-shot comparator resource.

Usage constraints Might ignore writes when the trace unit is enabled or not idle.

Configurations The TRCIDR4.NUMSSCC field sets the value of n and therefore controls how many TRCSSCCRs are implemented. TRCSSCCR<n> is implemented if n is less than TRCIDR4.NUMSSCC.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCSSCCRn bit assignments are:



Bits[31:25] RES0.

RST, bit[24] Controls whether the single-shot comparator resource is reset when it fires.

- **0** When the single-shot comparator resource fires, it is not reset.
- **1** When the single-shot comparator resource fires, it is reset. This enables the single-shot comparator resource to fire multiple times.

ARC, bits[23:16] Selects one or more address range comparators for single-shot control.

Each bit represents an address range comparator pair, so bit[*n*–16] controls the selection of address range comparator pair *n*–16. If bit[*n*–16] is:

- **0** The address range comparator pair *n*–16 is not selected for single-shot control.
- **1** The address range comparator pair *n*–16 is selected for single-shot control.

The implemented width of this field is IMPLEMENTATION DEFINED. The field contains a number of implemented bits equal to the value of TRCIDR4.NUMACPAIRS. All unimplemented bits are RES0.

SAC, bits[15:0] Selects one or more single address comparators for single-shot control.

Each bit represents a single address comparator, so bit[*n*] controls the selection of single address comparator *n*. If bit[*n*] is:

- **0** The single address comparator *n*, is not selected for single-shot control.
- **1** The single address comparator *n*, is selected for single-shot control.

The width of this field is IMPLEMENTATION DEFINED. The field contains a number of implemented bits equal to the two times the value of TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

7.3.67 TRCSSCSRn, Single-shot Comparator Status Registers, n=0-7

The TRCSSCSRn characteristics are:

Purpose Returns the status of the corresponding single-shot comparator.

Usage constraints

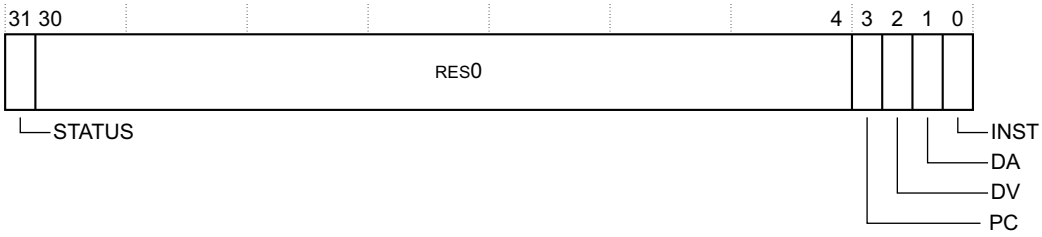
- Might ignore writes when the trace unit is enabled or not idle.
- The STATUS bit value is only stable when TRCSTATR.PMSTABLE==1.
- CONSTRAINED UNPREDICTABLE behavior of a single-shot comparator resource occurs if:
  - — DV==0 and software selects any comparators programmed for data address comparisons with a data value comparison using the associated TRCSSCCR<n>.
  - — DA==0 and software selects any comparators programmed for data address comparisons using the associated TRCSSCCR<n>.
  - — INST==0 and software selects any comparators programmed for instruction address comparisons using the associated TRCSSCCR<n>.

  In these scenarios, the single-shot comparator resource might match unexpectedly or might not match.

Configurations The TRCIDR4.NUMSSCC field sets the value of n and therefore controls how many TRCSSCSRs are implemented. TRCSSCSR<n> is implemented if n is less than TRCIDR4.NUMSSCC.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCSSCSRn bit assignments are:



STATUS, bit[31] Single-shot status bit. Indicates if any of the comparators selected by this Single-shot Comparator control have matched. The selected comparators are defined by TRCSSCCRn.ARC, TRCSSCCRn.SAC, and TRCSSPCICRn.PC:

0 No match has occurred.

When the first match occurs, this field takes a value of `0b1`. It remains at `0b1` until explicitly modified by a write to this register.

1 One or more matches has occurred. If TRCSSCCRn.RST==0 then:

- There is only one match and no more matches are possible.
- Software must reset this bit to 0 to re-enable the single-shot control.

The reset value is UNKNOWN. STATUS must be written to set an initial state when programming the trace unit, if the single-shot comparator is to be used.

Bits[30:4] RES0.

PC, bit[3] PE comparator input support. Indicates if the trace unit supports Single-shot PE comparator inputs. This field is read-only.

0 Single-shot PE comparator inputs are not supported.

Selecting any PE comparator inputs using the associated TRCSSPCICRn results in CONSTRAINED UNPREDICTABLE behavior of the Single-shot comparator resource. The comparator might match unexpectedly or might not match.

1 Single-shot PE comparator inputs are supported.

If PE comparator inputs are not implemented, this bit is RES0.

DV, bit[2] Data value comparator support bit. Indicates if the trace unit supports data address with data value comparisons. This field is read-only:

0 Single-shot data address with data value comparisons are not supported.

1 Single-shot data address with data value comparisons are supported.

DA, bit[1] Data address comparator support bit. Indicates if the trace unit supports data address comparisons. This field is read-only:

0 Single-shot data address comparisons are not supported.

1 Single-shot data address comparisons are supported.

INST, bit[0] Instruction address comparator support bit. Indicates if the trace unit supports instruction address comparisons. This field is read-only:

0 Single-shot instruction address comparisons are not supported.

1 Single-shot instruction address comparisons are supported.

7.3.68 TRCSSPCICRn, Single-shot Processing Element Comparator Input Control Register, n=0-7

The TRCSSPCICR characteristics are:

Purpose Selects the PE comparator inputs for Single-shot control.

Usage constraints

Can only be written when the trace unit is disabled.

Each Single-shot Processing Element Comparator Input Control Register has an associated Single-shot Comparator Status register which identifies the capabilities of the Single-shot Processing Element Comparator Input Control Register.

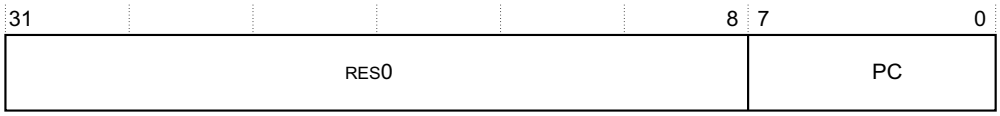
Configurations

There are up to 8 Single-shot Processing Element Comparator Input Control Registers, defined by a combination of TRCIDR4.NUMSSCC and TRCSSCSRn.PC.

Implemented if n < TRCIDR4.NUMSSCC and TRCSSCSRn.PC is `0b1`.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The bit assignments are:



Bits[31:8] RES0.

PC, bits[7:0] Selects one or more PE comparator inputs for Single-shot control.

TRCIDR4.NUMPC defines the size of the PC field.

1 bit is provided for each implemented PE comparator input.

For example, if bit[1] == 1 this selects PE comparator input 1 for Single-shot control.

7.3.69 TRCSTALLCTLR, Stall Control Register

The TRCSTALLCTLR characteristics are:

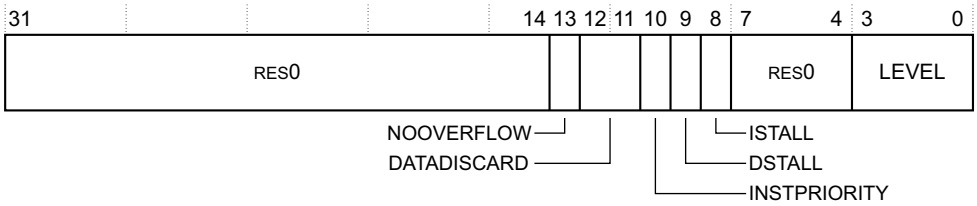
Purpose Enables trace unit functionality that prevents trace unit buffer overflows.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- Must be programmed if TRCIDR3.STALLCTL==1.

Configurations Implemented when TRCIDR3.STALLCTL==1.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCSTALLCTLR bit assignments are:



Bits[31:14] RES0.

NOOVERFLOW, bit[13]

Trace overflow prevention bit:

0 Trace overflow prevention is disabled.

1 Trace overflow prevention is enabled. This might cause a significant performance impact. See *Trace unit behavior on a trace buffer overflow on page 3-106* for more information.

TRCIDR3.NOOVERFLOW indicates whether this bit is supported. If it is not supported then this bit is RES0.

DATADISCARD, bits[12:11]

Data discard field. Controls if a trace unit can discard data trace elements when the data trace buffer space is less than LEVEL:

`0b00` The trace unit must not discard any data trace elements.

`0b01` The trace unit can discard P1 and P2 elements associated with data loads.

`0b10` The trace unit can discard P1 and P2 elements associated with data stores.

`0b11` The trace unit can discard P1 and P2 elements associated with both data loads and stores.

When a trace unit discards the first P1 element, it generates a Suppression element in the data trace stream. See *Suppression data trace element on page 5-227*.

TRCIDR0.TRCDATA indicates whether this field is supported. If it is not supported then this field is RES0.

INSTPRIORITY, bit[10]

Prioritize instruction trace bit. Controls if a trace unit can prioritize instruction trace when the instruction trace buffer space is less than LEVEL:

0 The trace unit must not prioritize instruction trace.

1 The trace unit can prioritize instruction trace. A trace unit might prioritize instruction trace by preventing output of data trace, or other means which ensure that the instruction trace has a higher priority than the data trace.

TRCIDR0.TRCDATA indicates whether this bit is supported. If it is not supported then this bit is RES0.

DSTALL, bit[9] Data stall bit. Controls if a trace unit can stall the PE when the data trace buffer space is less than LEVEL:

0 The trace unit must not stall the PE.

1 The trace unit can stall the PE.

TRCIDR0.TRCDATA indicates whether this bit is supported. If it is not supported then this bit is RES0.

If all invasive debug is disabled in the PE, the trace unit must not stall the PE and this field is ignored.

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 the trace units then the stall has minimal effect on the PEs which are not being traced.

ISTALL, bit[8] Instruction stall bit. Controls if a trace unit can stall the PE when the instruction trace buffer space is less than LEVEL:

0 The trace unit must not stall the PE.

1 The trace unit can stall the PE.

If all invasive debug is disabled in the PE, the trace unit must not stall the PE and this field is ignored.

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 the trace units then the stall has minimal effect on the PEs which are not being traced.

Bits[7:4] RES0.

LEVEL, bits[3:0] Threshold level field. The field can support 16 monotonic levels from `0b0000` to `0b1111`, where:

`0b0000` Zero invasion. This setting has a greater risk of a FIFO overflow.

Note

For some implementations, invasion might occur at `0b0000`.

`0b1111` Maximum invasion occurs but there is less risk of a FIFO overflow.

Note

It is IMPLEMENTATION DEFINED whether some of the least significant bits are supported. Arm recommends that bits[3:2] are supported.

If LEVEL is nonzero then a trace unit might suppress the generation of:

- Global timestamps in the instruction trace stream and the data trace stream.
- Cycle counting in the instruction trace stream, although the cumulative cycle count remains correct.

7.3.70 TRCSTATR, Trace Status Register

The TRCSTATR characteristics are:

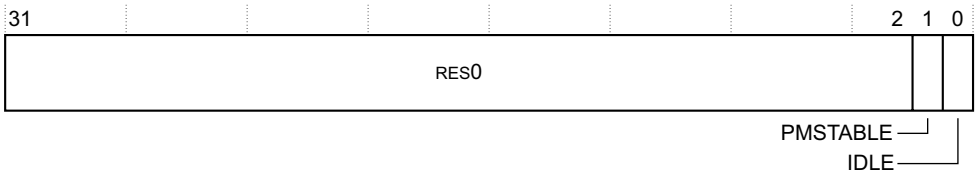
Purpose Returns the trace unit status.

Usage constraints There are no usage constraints.

Configurations Available in all implementations.

Attributes A 32-bit RO trace register. The register is set to an UNPREDICTABLE value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCSTATR bit assignments are:



Bits[31:2] RES0.

PMSTABLE, bit[1] Programmers' model stable bit:

0 The programmers' model is not stable.

1 The programmers' model is stable. When polled, the trace unit trace registers return stable data.

Note

This bit is only valid when either:

- TRCPRGCTLR.EN==0
- The OS Lock is locked.

The programmers' model is stable when all of the following are true:

- TRCPRGCTLR.EN==0 or the OS Lock is locked.
- Reads from trace unit registers return stable data, such as reads from:
  - *TRCVICTLR, ViewInst Main Control Register on page 7-413*.
  - *TRCSEQSTR, Sequencer State Register on page 7-401*.
  - *TRCCNTVRn, Counter Value Registers, n=0-3 on page 7-361*.
  - *TRCSSCSRn, Single-shot Comparator Status Registers, n=0-7 on page 7-403*.

IDLE, bit[0] Idle status bit:

0 The trace unit is not idle.

1 The trace unit is idle.

The trace unit is idle when all of the following are true:

- TRCPRGCTLR.EN==0 or the OS Lock is locked.
- The trace unit is drained of any trace.
- With the exception of the programming interfaces, all external interfaces on the trace unit are quiescent.
- When enabling the trace unit, when TRCPRGCTLR.EN==1 and the OS Lock is unlocked, the trace unit might not immediately leave the idle state, but must leave the idle state in finite time. When enabling the trace unit, you must poll TRCSTATR.IDLE until it is `0b0` to ensure the trace unit is not idle. See *Use of the trace unit main enable bit on page 4-165* for more details on using TRCSTATR.IDLE.

7.3.71 TRCSYNCPR, Synchronization Period Register

The TRCSYNCPR characteristics are:

Purpose Controls how often trace synchronization requests occur.

Usage constraints

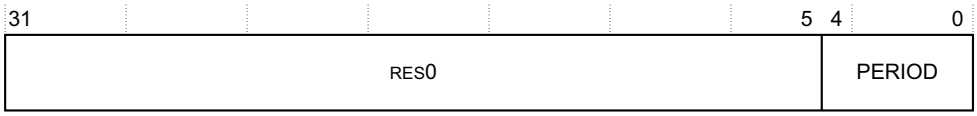
- Might ignore writes when the trace unit is enabled or not idle.

- It is IMPLEMENTATION DEFINED whether writes are permitted:
  - If TRCIDR3.SYNCPR==0 then the register is a RW register.
  - If TRCIDR3.SYNCPR==1 then the register is RO and the synchronization period, given in PERIOD, is IMPLEMENTATION DEFINED.
- If writes are permitted then the register must be programmed.
- If a reserved value is programmed into TRCSYNCPR.PERIOD, then the behavior of the synchronization period counter is CONSTRAINED UNPREDICTABLE and one of the following behaviors occurs:
  - No trace synchronization requests are generated by this counter.
  - Trace synchronization requests occur at the specified period.
  - Trace synchronization requests occur at some other UNKNOWN period which can vary.

Configurations Available in all implementations.

Attributes A 32-bit RW or RO trace register. The reset value is UNKNOWN when the register is RW, and IMPLEMENTATION DEFINED when the register is RO. This register is reset on a trace unit reset. See the Usage constraints. See also *Register summary on page 7-336*.

The TRCSYNCPR bit assignments are:



Bits[31:5] RES0.

PERIOD, bits[4:0] Controls how many bytes of trace, the sum of instruction and data, that a trace unit can generate before a trace synchronization request occurs. The number of bytes is always a power of two and the permitted values are:

`0b00000`	Trace synchronization requests are disabled. This setting does not disable other types of trace synchronization request.
`0b01000`	Trace synchronization request occurs after 2^8, or 256, bytes of trace.
`0b01001`	Trace synchronization request occurs after 2^9, or 512, bytes of trace.
`0b01010`	Trace synchronization request occurs after 2^{10}, or 1024, bytes of trace.
.	.
.	.
.	.
`0b10100`	Trace synchronization request occurs after 2^{20}, or 1048576, bytes of trace.

7.3.72 TRCTRACEIDR, Trace ID Register

The TRCTRACEIDR characteristics are:

Purpose Sets the trace ID for instruction trace. If data trace is enabled then it also sets the trace ID for data trace, to (trace ID for instruction trace) + 1.

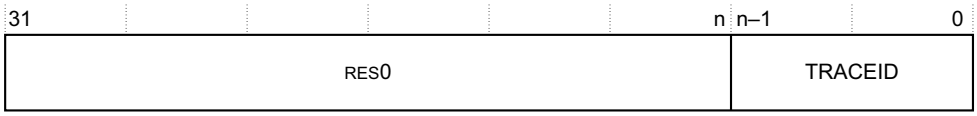
Usage constraints
- This register must always be programmed as part of trace unit initialization.
- Might ignore writes when the trace unit is enabled or not idle.

Configurations Available in all implementations.

The TRACEID field width is IMPLEMENTATION DEFINED and is set by TRCIDR5.TRACEIDSIZE.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCTRACEIDR bit assignments are:



Bits[31:n] RES0.

TRACEID, bits[n–1:0]

Trace ID field. Sets the trace ID value for instruction trace.

Bit[0] must be zero if data trace is enabled. If data trace is enabled then a trace unit sets the trace ID for data trace, to TRACEID+1.

The implemented width of the field is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR5.TRACEIDSIZE. Unimplemented bits are RES0.

If an implementation supports *AMBA* ATB, then:

- The width of the field is 7 bits.
- Writing a reserved trace ID value does not affect behavior of the trace unit but it might cause UNPREDICTABLE behavior of the trace capture infrastructure. See the *AMBA 3 ATB Protocol Specification* for information about which **ATID** bus values are reserved.

7.3.73 TRCTSCTLR, Global Timestamp Control Register

The TRCTSCTLR characteristics are:

Purpose Controls the insertion of global timestamps in the trace streams.

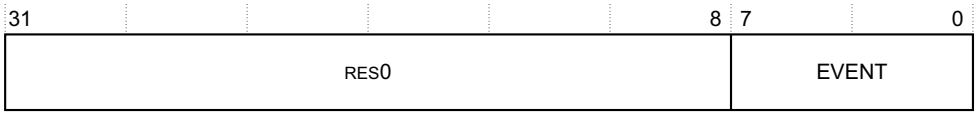
Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- Must be programmed if TRCCONFIGR.TS==1.

Configurations Implemented when TRCIDR0.TSSIZE is nonzero.

Note
- For ETM4.3 or later, if TRCIDR4.NUMRSPAIR has the value `0b0000`, the trace unit has limited resources. However, TRCTSCTLR is implemented whenever TRCIDR0.TSSIZE indicates that global timestamping is implemented, regardless of the value of TRCIDR4.NUMRSPAIR. See also the field description of TRCIDR4.NUMRSPAIR.
- Global timestamping can only be used if the system contains a timestamp source.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCTSCTLR bit assignments are:



Bits[31:8] RES0.

EVENT, bits[7:0] An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes. When the selected event is triggered, the trace unit inserts a global timestamp into the trace streams.

For ETMv4.3 or later, if TRCIDR4.NUMRSPAIR is `0b0000`, this field behaves as if the FALSE event is selected, and is RES0.

7.3.74 TRCVDARCCTLR, ViewData Include/Exclude Address Range Comparator Control Register

The TRCVDARCCTLR characteristics are:

Purpose Use this to set, or read, the address range comparators for:

- ViewData include control.
- ViewData exclude control.

Usage constraints

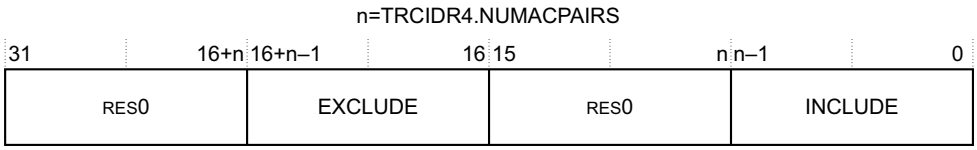
- Might ignore writes when the trace unit is enabled or not idle.
- This register must be programmed when both of the following apply:
  - Data tracing is enabled, that is, TRCCONFIGR.DA==1 or TRCCONFIGR.DV==1.
  - One or more address comparators are implemented, that is, TRCIDR4.NUMACPAIRS > 0.
- CONSTRAINED UNPREDICTABLE tracing occurs if software writes to this register and selects an address range comparator that is programmed to be sensitive to a data value comparator. Data transfers might be traced unexpectedly or might not be traced.

Configurations Implemented only when TRCIDR4.NUMACPAIRS > 0 and data tracing is implemented.

The width of the INCLUDE and EXCLUDE fields are IMPLEMENTATION DEFINED.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVDARCCTLR bit assignments are:



Bits[31:16+n] RES0.

EXCLUDE, bits[16+n–1:16]

Exclude range field. Selects which address range comparator pairs are in use with ViewData exclude control. Each bit represents an address range comparator pair, so bit[*m*] controls the selection of address range comparator pair *m*–16. If bit[*m*] is:

0 The address range that address range comparator pair *m*–16 defines, is not selected for ViewData exclude control.

1 The address range that address range comparator pair *m*–16 defines, is selected for ViewData exclude control.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

Bits[15:n] RES0.

INCLUDE, bits[n–1:0]

Include range field. Selects which address range comparator pairs are in use with ViewData include control. Each bit represents an address range comparator pair, so bit[*m*] controls the selection of address range comparator pair *m*. If bit[*m*] is:

0 The address range that address range comparator pair *m* defines, is not selected for ViewData include control.

1 The address range that address range comparator pair *m* defines, is selected for ViewData include control.

If no single address comparators and no address range comparators are selected to be included, then all data transfers are included by default. The exclude control then indicates which data transfers are excluded.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

7.3.75 TRCVDCTLR, ViewData Main Control Register

The TRCVDCTLR characteristics are:

Purpose Controls data trace filtering.

Usage constraints
- Might ignore writes when the trace unit is enabled or not idle.
- This register must be programmed when data tracing is enabled, that is, when either TRCCONFIGR.DA==1 or TRCCONFIGR.DV==1.

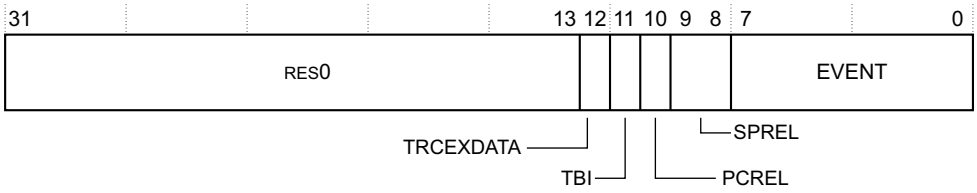
Configurations Implemented only in trace units that implement data tracing.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

Precise filtering of data trace is not possible using data value comparisons. Software can use the standard event to achieve imprecise filtering.

It is not possible to trace a data transfer when the parent instruction is not traced. To trace data values requires that both the following conditions are met:
- The parent instruction is traced, by using the ViewInst function.
- The data address is traced, by using the ViewData function.

The TRCVDCTLR bit assignments are:



Bits[31:13] RES0.

TRCEXDATA, bit[12]

Controls the tracing of data transfers for exceptions and exception returns on Armv6-M, Armv7-M, and Armv8-M PEs:

0 Exception and exception return data transfers are not traced.

1 Exception and exception return data transfers are traced if the other aspects of ViewData indicate that the data transfers must be traced.

The portions of the data transfers which are traced are governed by the TRCCONFIGR.DA and TRCCONFIGR.DV fields, in the same way as all other data transfers.

If TRCCONFIGR.INSTP0 indicates that only load instructions are P0 instructions, then the data store transfers performed when an exception occurs are not traced. If TRCCONFIGR.INSTP0 indicates that only store instructions are P0 instructions, then the data load transfers performed when an exception return occurs are not traced.

If this field is set to 1 when TRCCONFIGR.INSTP0 is `0b00`, then the behavior of the trace unit is UNPREDICTABLE.

This field is implemented if TRCIDR0.TRCEXDATA is 1.

TBI, bit[11] Controls which information a trace unit populates in bits[63:56] of the data address:

0 The trace unit assigns bits[63:56] to have the same value as bit[55] of the data address, that is, it sign-extends the value.

1 The trace unit assigns bits[63:56] to have the same value as bits[63:56] of the data address.

TRCIDR2.DASIZE indicates whether this bit is implemented.

PCREL, bit[10] Controls whether a trace unit traces data for transfers that are relative to the *Program Counter* (PC):

0 The trace unit does not affect the tracing of PC-relative transfers.

1 The trace unit does not trace the address or value portions of PC-relative transfers.

This bit only affects PC-relative transfers that use the PC as a base register plus an immediate offset.

SPREL, bits[9:8] Controls whether a trace unit traces data for transfers that are relative to the *Stack Pointer* (SP):

`0b00` The trace unit does not affect the tracing of SP-relative transfers.

`0b01` Reserved.

`0b10` The trace unit does not trace the address portion of SP-relative transfers. If data value tracing is enabled then the trace unit generates a P1 data address element.

`0b11` The trace unit does not trace the address or value portions of SP-relative transfers.

This field only affects SP-relative transfers that use the SP as a base register plus an immediate offset.

On Armv6-M, Armv7-M and Armv8-M PEs this field affects the stack push which is a set of SP-relative stores that take place when an exception occurs, and the stack pop, which is a set of SP-relative loads that take place when an exception return occurs.

EVENT, bits[7:0] An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes.

From ETMv4.3, if TRCIDR4.NUMRSPAIR is `0b0000`, this field behaves as if the TRUE event is selected:

- Bits[7:1] are RES0.
- Bit[0] is RES1.

7.3.76 TRCVDSACCTLR, ViewData Include-Exclude Single Address Comparator Control Register

The TRCVDSACCTLR characteristics are:

Purpose

Use this to set, or read, the single address comparators for:

- ViewData include control.
- ViewData exclude control.

Usage constraints

- Might ignore writes when the trace unit is enabled or not idle.
- This register must be programmed when both of the following apply:
  - Data tracing is enabled, that is, TRCCONFIGR.DA==1 or TRCCONFIGR.DV==1.
  - One or more address comparators are implemented, that is, TRCIDR4.NUMACPAIRS > 0.
- CONSTRAINED UNPREDICTABLE behavior of ViewData occurs if software writes to this register and selects a single address comparator that is either:
  - Programmed to be sensitive to a data value comparator.
  - Programmed to be an instruction address comparator.

  In these situations, data transfers might be traced unexpectedly or might not be traced.

Configurations

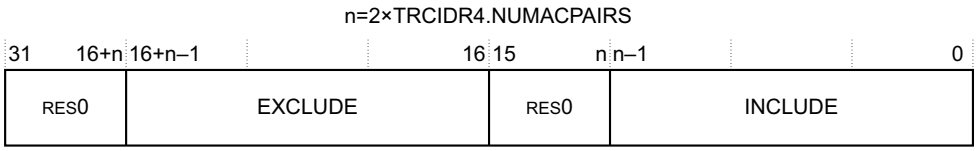
Implemented only when TRCIDR4.NUMACPAIRS > 0 and when data tracing is implemented.

The width of the INCLUDE and EXCLUDE fields are IMPLEMENTATION DEFINED.

Attributes

A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVDSACCTLR bit assignments are:



Bits[31:16+n] RES0.

EXCLUDE, bits[16+n–1:16]

Selects which single address comparators are in use with ViewData exclude control. Each bit represents a single address comparator, so bit[*m*] controls the selection of single address comparator *m*–16. If bit[*m*] is:

0 The single address comparator *m*–16, is not selected for exclude control.

1 The single address comparator *m*–16, is selected for exclude control.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of 2×TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

Bits[15:n] RES0.

INCLUDE, bits[n–1:0]

Selects which single address comparators are in use with ViewData include control. Each bit represents a single address comparator, so bit[*n*] controls the selection of single address comparator *n*. If bit[*n*] is:

0 The single address comparator *n*, is not selected for include control.

1 The single address comparator *n*, is selected for include control.

If no single address comparators and no address range comparators are selected to be included, then all data transfers are included by default. The exclude control then indicates which data transfers are excluded.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of 2×TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

7.3.77 TRCVICTLR, ViewInst Main Control Register

The TRCVICTLR characteristics are:

Purpose Controls instruction trace filtering.

Usage constraints

- Might ignore writes when the trace unit is enabled or not idle.
- Only returns stable data when TRCSTATR.PMSTABLE==1.
- Must be programmed, particularly to set the value of the SSSTATUS bit, which sets the state of the start/stop logic.

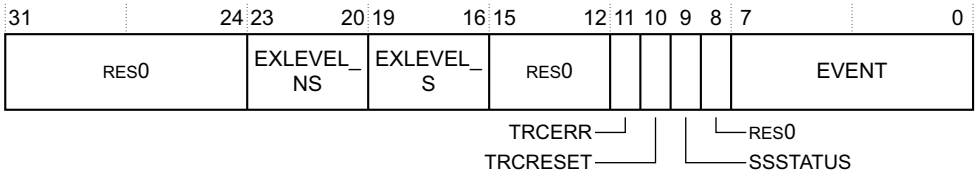
Configurations Available in all ETM implementations.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

Precise filtering of instruction trace is not possible using data address or data value comparisons. Software can use the ViewInst enabling event to achieve imprecise filtering, controlled by TRCVICTLR.EVENT.

It is not possible to trace a data transfer when the parent instruction is not traced. ViewInst must be active for the instruction because this enables the trace unit to trace the data transfer.

The TRCVICTLR bit assignments are:



Bits[31:24] RES0.

EXLEVEL_NS, bits[23:20]

In Non-secure state, each bit controls whether instruction tracing is enabled for the corresponding Exception level:

0 The trace unit generates instruction trace, in Non-secure state, for Exception level *n*.

For Armv8-M PEs, the trace unit does not distinguish between the Secure and Non-secure states, and treats both states as Secure state. This field always reads as zero, indicating no Non-secure states are supported.

1 The trace unit does not generate instruction trace, in Non-secure state, for Exception level *n*.

The Exception levels are:

Bit[20] Exception level 0.

Bit[21] Exception level 1.

Bit[22] Exception level 2.

Bit[23] RES0. EXLEVEL_NS[3] is never implemented.

The content of **EXLEVEL_NS** is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR3.EXLEVEL_NS. If instruction tracing is not implemented for a given Exception level, the corresponding bit in this field is not implemented. Unimplemented bits are RES0.

EXLEVEL_S, bits[19:16]

In Secure state, each bit controls whether instruction tracing is enabled for the corresponding Exception level:

0 The trace unit generates instruction trace, in Secure state, for Exception level *n*.

1 The trace unit does not generate instruction trace, in Secure state, for Exception level *n*.

The Exception levels are:

Bit[16] Exception level 0.

Bit[17] Exception level 1.

Bit[18] Exception level 2.

Bit[19] Exception level 3.

The content of **EXLEVEL_S** is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR3.EXLEVEL_S. If instruction tracing is not implemented for a given Exception level, the corresponding bit in this field is not implemented. Unimplemented bits are RES0.

Bits[15:12] RES0.

TRCERR, bit[11]

When TRCIDR3.TRCERR==1, this bit controls whether a trace unit must trace a System error exception:

0 The trace unit does not trace a System error exception unless it traces the exception or instruction immediately prior to the System error exception.

1 The trace unit always traces a System error exception, regardless of the value of ViewInst.

Note

A System error exception is:

- An asynchronous Data Abort, in the Armv7 architecture.
- An SError interrupt, in the Armv8-A and Armv8-R architecture.
- One of the following exceptions on Armv6-M, Armv7-M, and Armv8-M:
  - MemManage.
  - BusFault.
  - HardFault.
  - Lockup.
  - SecureFault.

When TRCIDR3.TRCERR==0, this bit is RES0.

TRCRESET, bit[10] Controls whether a trace unit must trace a Reset exception:

0 The trace unit does not trace a Reset exception unless it traces the exception or instruction immediately prior to the Reset exception.

1 The trace unit always traces a Reset exception.

SSSTATUS, bit[9] When TRCIDR4.NUMACPAIRS > 0 or TRCIDR4.NUMPC > 0, this bit returns the status of the start/stop logic:

0 The start/stop logic is in the stopped state.

1 The start/stop logic is in the started state.

The bit only returns stable data when TRCSTATR.PMSTABLE==1.

Before software enables the trace unit, TRCPRGCTLR.EN==1, it must write to this bit to set the initial state of the start/stop logic. If the start/stop logic is not used then set this bit to 1. Arm recommends that the value of this bit is set before each trace run begins.

If the trace unit is disabled while a Start or Stop point is still speculative, then the value of TRCVICTLR.SSSTATUS in UNKNOWN and might present the result of a speculative start or stop point.

If software which is running on the PE being traced disables the trace unit, either by clearing TRCPRGCTLR.EN or locking the OS Lock, Arm recommends that a `DSB` and an `ISB` instruction are executed before disabling the trace unit to prevent any start or stop points being speculative at the point of disabling the trace unit. This procedure assumes that all Start or Stop points occur before the barrier instructions are executed. The procedure does not guarantee that there are no speculative Start or Stop points when disabling, although it helps minimize the probability.

When TRCIDR4.NUMACPAIRS== 0 and TRCIDR4.NUMPC== 0 this bit is RES1. This indicates that the start/stop logic is not implemented.

Bit[8] RES0.

EVENT, bits[7:0] An event selector, as *Activating a trace unit event with a selected trace unit resource or pair of trace unit resources on page 4-177* describes.

From ETMv4.3, if TRCIDR4.NUMRSPAIR is `0b0000`, this field behaves as if the TRUE event is selected:

- Bits[7:1] are RES0.
- Bit[0] is RES1.

7.3.78 TRCVIIECTLR, ViewInst Include-Exclude Control Register

The TRCVIIECTLR characteristics are:

Purpose Use this to set, or read, the address range comparators for:

- ViewInst include control.

- ViewInst exclude control.

Usage constraints

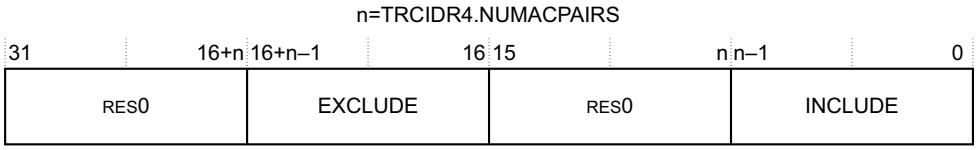
- Might ignore writes when the trace unit is enabled or not idle.
- This register must be programmed when one or more address comparators are implemented, that is, when TRCIDR4.NUMACPAIRS > 0.
- CONSTRAINED UNPREDICTABLE tracing occurs if software writes to this register and selects an address range comparator pair that is not programmed to be an instruction address comparator. That is, instructions might be traced unexpectedly or might not be traced.

Configurations Implemented only when TRCIDR4.NUMACPAIRS > 0.

The width of the INCLUDE and EXCLUDE fields are IMPLEMENTATION DEFINED.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVIIECTLR bit assignment are:



Bits[31:16+n] RES0.

EXCLUDE, bits[16+n–1:16]

Exclude range field. Selects which address range comparator pairs are in use with ViewInst exclude control. Each bit represents an address range comparator pair, so bit[*m*] controls the selection of address range comparator pair *m*–16. If bit[*m*] is:

0 The address range that address range comparator pair *m*–16 defines, is not selected for ViewInst exclude control.

1 The address range that address range comparator pair *m*–16 defines, is selected for ViewInst exclude control.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

Bits[15:n] RES0.

INCLUDE, bits[n–1:0]

Include range field. Selects which address range comparator pairs are in use with ViewInst include control. Each bit represents an address range comparator pair, so bit[*m*] controls the selection of address range comparator pair *m*. If bit[*m*] is:

0 The address range that address range comparator pair *m* defines, is not selected for ViewInst include control.

1 The address range that address range comparator pair *m* defines, is selected for ViewInst include control.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

Selecting no include comparators indicates that all instructions are included by default. The exclude control then indicates which ranges are excluded.

7.3.79 TRCVIPCSSCTLR, ViewInst Start/Stop Processing Element Comparator Control Register

The TRCVIPCSSCTLR characteristics are:

Purpose Use this to set, or read, which PE comparator inputs can control the ViewInst start/stop logic.

Usage constraints

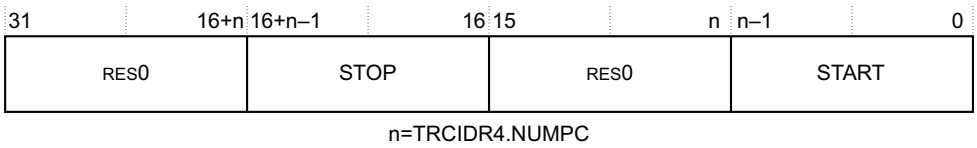
- Might ignore writes when the trace unit is enabled or not idle.
- If implemented then this register must be programmed.
- CONSTRAINED UNPREDICTABLE behavior of the start/stop logic occurs if a single PE comparator input is programmed as both a stop resource and a start resource. That is, the start/stop logic is either active or inactive for the instruction and the start/stop logic is either active or inactive after the instruction.

Configurations Implemented only when TRCIDR4.NUMPC > 0.

The width of the START and STOP fields are IMPLEMENTATION DEFINED.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVIPCSSCTLR bit assignments are:



Bits[31:16+n] RES0.

STOP, bits[16+n–1:16]

Selects which PE comparator inputs are in use with ViewInst start/stop control, for the purpose of stopping trace. Each bit represents a PE comparator input, so bit[*m*] controls the selection of PE comparator input *m*–16. If bit[*m*] is:

0 The single PE comparator input *m*–16, is not selected as a stop resource.

1 The single PE comparator input *m*–16, is selected as a stop resource.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR4.NUMPC. Unimplemented bits are RES0.

Bits[15:n] RES0.

START, bits[n–1:0] Selects which PE comparator inputs are in use with ViewInst start/stop control, for the purpose of starting trace. Each bit represents a PE comparator input, so bit[*n*] controls the selection of PE comparator input *n*. If bit[*n*] is:

0 The single PE comparator input *n*, is not selected as a start resource.

1 The single PE comparator input *n*, is selected as a start resource.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of TRCIDR4.NUMPC. Unimplemented bits are RES0.

7.3.80 TRCVISSCTLR, ViewInst Start/Stop Control Register

The TRCVISSCTLR characteristics are:

Purpose Use this to set, or read, the single address comparators that control the ViewInst start/stop logic. The start/stop logic is active for an instruction which causes a start and remains active up to and including an instruction which causes a stop, and then the start/stop logic becomes inactive.

Usage constraints

- Might ignore writes when the trace unit is enabled or not idle.
- If implemented then this register must be programmed.
- CONSTRAINED UNPREDICTABLE tracing occurs if a single address comparator is programmed as a stop resource or a start resource when that address comparator is not programmed to be an instruction address comparator. Instructions might or might not be traced.

- CONSTRAINED UNPREDICTABLE tracing occurs if:
  - A single address comparator is programmed as both a stop resource and a start resource.
  - Two or more single address comparators trigger on the same instruction, when these comparators are stop resources and start resources.

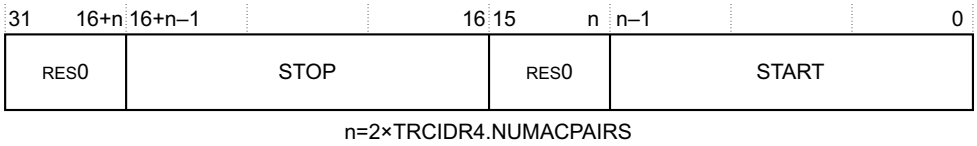
  It is UNPREDICTABLE whether the start/stop logic is active for the instruction and it is UNPREDICTABLE whether the start/stop logic is active after the instruction.

Configurations Implemented only when TRCIDR4.NUMACPAIRS > 0.

The width of the START and STOP fields are IMPLEMENTATION DEFINED.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVISSCTLR bit assignments are:



Bits[31:16+n] RES0.

STOP, bits[16+n–1:16]

Selects which single address comparators are in use with ViewInst start/stop control, for the purpose of stopping trace. Each bit represents a single address comparator, so bit[*m*] controls the selection of single address comparator *m*–16. If bit[*m*] is:

0 The single address comparator *m*–16, is not selected as a stop resource.

1 The single address comparator *m*–16, is selected as a stop resource.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of 2×TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

Bits[15:n] RES0.

START, bits[n–1:0] Selects which single address comparators are in use with ViewInst start/stop control, for the purpose of starting trace. Each bit represents a single address comparator, so bit[*n*] controls the selection of single address comparator *n*. If bit[*n*] is:

0 The single address comparator *n*, is not selected as a start resource.

1 The single address comparator *n*, is selected as a start resource.

The implemented width of the field, *n*, is IMPLEMENTATION DEFINED and is defined by the value of 2×TRCIDR4.NUMACPAIRS. Unimplemented bits are RES0.

7.3.81 TRCVMIDCCTLR0, Virtual context identifier Comparator Control Register 0

The TRCVMIDCCTLR0 characteristics are:

Purpose Contains Virtual machine identifier mask values for the TRCVMIDCVRn registers, where n=0-3.

Usage constraints

- Might ignore writes when the trace unit is enabled or not idle.
- If software uses the TRCVMIDCVRn registers, where n=0-3, then it must program this register.
- If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVRn to `0x00`.
- If any bit is `0b1` and the relevant byte in TRCVMIDCVRn is not `0x00`, the behavior of the Virtual context identifier comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Configurations

- Only implemented when Virtual context identifier tracing is implemented, TRCIDR4.NUMVMIDC > 0, indicating that at least one Virtual context identifier comparator is implemented, and TRCIDR2.VMIDSIZE > 8 bits.
- The number of COMP<n> fields that the register contains is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMVMIDC.
- The implemented width of a COMP<n> field is IMPLEMENTATION DEFINED and is set by TRCIDR2.VMIDSIZE. Unimplemented bits are RES0.

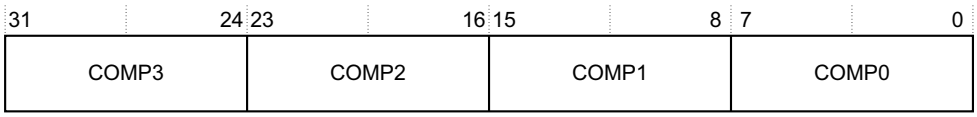
Note

Prior to ETMv4.1, all fields in these registers are zero-width because the maximum supported Virtual context identifier size is 8 bits. From ETMv4.1, and where TRCIDR2.VMIDSIZE indicates a Virtual context identifier larger than 8 bits, these fields are present and based on TRCIDR2.VMIDSIZE.

Attributes

A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVMIDCCTLR0 bit assignments are:



COMP3, bits[31:24] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==3. Each bit in this field corresponds to a byte in TRCVMIDCVR3. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR3 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR3 when it performs the Virtual context identifier comparison.

For example, if bit[30]==1 then the trace unit ignores TRCVMIDCVR3.VALUE[55:48].

Supported only if TRCIDR4.NUMVMIDC≥0b0100, otherwise bits[31:24] are RES0.

COMP2, bits[23:16] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==2. Each bit in this field corresponds to a byte in TRCVMIDCVR2. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR2 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR2 when it performs the Virtual context identifier comparison.

For example, if bit[21]==1 then the trace unit ignores TRCVMIDCVR2.VALUE[47:40].

Supported only if TRCIDR4.NUMVMIDC≥0b0011, otherwise bits[23:16] are RES0.

COMP1, bits[15:8] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==1. Each bit in this field corresponds to a byte in TRCVMIDCVR1. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR1 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR1 when it performs the Virtual context identifier comparison.

For example, if bit[12]==1 then the trace unit ignores TRCVMIDCVR1.VALUE[39:32].

Supported only if TRCIDR4.NUMVMIDC≥0b0010, otherwise bits[15:8] are RES0.

COMP0, bits[7:0] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==0. Each bit in this field corresponds to a byte in TRCVMIDCVR0. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR0 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR0 when it performs the Virtual context identifier comparison.

For example, if bit[3]==1 then the trace unit ignores TRCVMIDCVR0.VALUE[31:24].

Supported only if TRCIDR4.NUMVMIDC≥0b0001, otherwise bits[7:0] are RES0.

7.3.82 TRCVMIDCCTLR1, Virtual context identifier Comparator Control Register 1

The TRCVMIDCCTLR1 characteristics are:

Purpose Contains Virtual context identifier mask values for the TRCVMIDCVRn registers, where n=4-7.

Usage constraints

- Might ignore writes when the trace unit is enabled or not idle.
- If software uses the TRCVMIDCVRn registers, where n=4-7, then it must program this register.
- If software sets a mask bit to 1 then it must program the relevant byte in TRCVMIDCVRn to 0x00.
- If any bit is 0b1 and the relevant byte in TRCVMIDCVRn is not 0x00, the behavior of the VMID comparator is CONSTRAINED UNPREDICTABLE. In this scenario the comparator might match unexpectedly or might not match.

Configurations

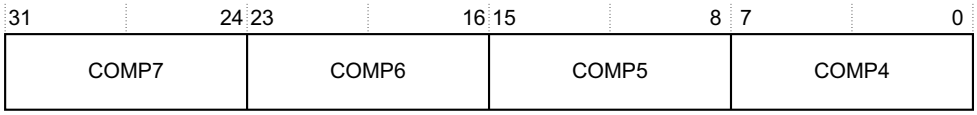
- Only implemented when Virtual context identifier tracing is implemented, TRCIDR4.NUMVMIDC > 4, indicating that at least 5 Virtual context identifier comparators are implemented, and TRCIDR2.VMIDSIZE > 8 bits.
- The number of COMP<n> fields that the register contains is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMVMIDC–4.
- The implemented width of a COMP<n> field is IMPLEMENTATION DEFINED and is set by TRCIDR2.VMIDSIZE. Unimplemented bits are RES0.

Note

Prior to ETMv4.1, all fields in these registers are zero-width because the maximum supported Virtual context identifier size is 8 bits. From ETMv4.1, and where TRCIDR2.VMIDSIZE indicates a Virtual context identifier larger than 8 bits, these fields are present and based on TRCIDR2.VMIDSIZE.

Attributes A 32-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVMIDCCTLR1 bit assignments are:



COMP7, bits[31:24] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==7. Each bit in this field corresponds to a byte in TRCVMIDCVR7. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR7 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR7 when it performs the Virtual context identifier comparison.

For example, if bit[30]==1 then the trace unit ignores TRCVMIDCVR7.VALUE[55:48].

Supported only if TRCIDR4.NUMVMIDC==0b1000, otherwise bits[31:24] are RES0.

COMP6, bits[23:16] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==6. Each bit in this field corresponds to a byte in TRCVMIDCVR6. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR6 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR6 when it performs the Virtual context identifier comparison.

For example, if bit[21]==1 then the trace unit ignores TRCVMIDCVR6.VALUE[47:40].

Supported only if TRCIDR4.NUMVMIDC≥0b0111, otherwise bits[23:16] are RES0.

COMP5, bits[15:8] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==5. Each bit in this field corresponds to a byte in TRCVMIDCVR5. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR5 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR5 when it performs the Virtual context identifier comparison.

For example, if bit[12]==1 then the trace unit ignores TRCVMIDCVR5.VALUE[39:32].

Supported only if TRCIDR4.NUMVMIDC≥0b0110, otherwise bits[15:8] are RES0.

COMP4, bits[7:0] Controls the mask value that the trace unit applies to TRCVMIDCVRn, where n==4. Each bit in this field corresponds to a byte in TRCVMIDCVR4. When a bit is:

0 The trace unit includes the relevant byte in TRCVMIDCVR4 when it performs the Virtual context identifier comparison.

1 The trace unit ignores the relevant byte in TRCVMIDCVR4 when it performs the Virtual context identifier comparison.

For example, if bit[3]==1 then the trace unit ignores TRCVMIDCVR4.VALUE[31:24].

Supported only if TRCIDR4.NUMVMIDC≥0b0101, otherwise bits[7:0] are RES0.

7.3.83 TRCVMIDCVRn, Virtual context identifier Comparator Value Registers, n=0-7

The TRCVMIDCVRn characteristics are:

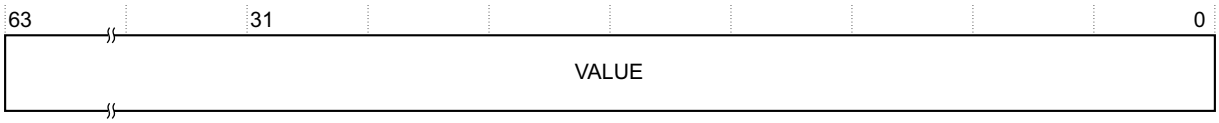
Purpose Contains a Virtual context identifier value.

Usage constraints Might ignore writes when the trace unit is enabled or not idle.

Configurations The number, n, of TRCVMIDCVRn is IMPLEMENTATION DEFINED and is set by TRCIDR4.NUMVMIDC.

Attributes A 64-bit RW trace register. The register is set to an UNKNOWN value on a trace unit reset. See also *Register summary on page 7-336*.

The TRCVMIDCVRn bit assignments are:



VALUE, bits[63:0] Virtual context identifier value. The implemented width of this field is IMPLEMENTATION DEFINED, and is set by TRCIDR2.VMIDSIZE. Unimplemented bits are RES0.

When TRCIDR2.VMIDSIZE indicates a 16-bit Virtual context identifier, this field is 16 bits wide.

When TRCIDR2.VMIDSIZE indicates a 32-bit Virtual context identifier, this field is 32 bits wide.

When TRCCONFIGR.VMIDOPT is `0b0`, and for Armv8-A PEs where TRCIDR2.VMIDSIZE indicates at least a 16-bit Virtual context identifier, if the VTCR_EL2.VS bit in the PE is `0b0`, then the upper 8 bits of the Virtual context identifier are always zero and the Virtual context identifier comparator must compare all 16 bits of TRCVMIDCVRn with the full 16-bit Virtual context identifier.

After a PE reset, the trace unit assumes that the Virtual context identifier is zero until the PE updates the Virtual context identifier.

Appendix A
Examples of Trace

This appendix gives some examples of trace obtained by using an ETMv4 trace unit. It contains the following sections:

A.1 An example of basic program trace

The example in Table A-1 shows basic program trace, where only branch instructions are traced as P0 elements. In this example the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table A-1 Example of basic program trace

PE execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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.
-	`IRQ`	exception_element(IRQ, `0x2014`)	An IRQ occurs. The Exception element indicates the STR instruction was executed.
-	commit all execution	commit_element(3)	This commits the two Atom elements generated for the branch instructions at `0x1000` and `0x200C`, plus the *Exception element* generated for the IRQ exception.

A.2 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* on page A-426.
- *Example of basic program trace when two consecutive exceptions occur* on page A-427.

A.2.1 Basic program trace when an exception occurs, example one

The example in Table A-2 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 A-2 Example of basic program trace when an exception occurs, example one

PE execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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_element(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 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_element (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_element(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.

A.2.2 Basic program trace when an exception occurs, example two

The example in Table A-3 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 A-3 Example of basic program trace when an exception occurs, example two

PE execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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_element(1) exception_element(data fault, `0x2004`)	The Cancel element 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 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 P0 element at `0x200C`. However, the trace analyzer canceled this because of the Cancel element. 2. The `MOV` instruction is now implied by the Exception P0 element.
-	commit all execution	commit_element(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 data fault exception.

A.2.3 Example of basic program trace when two consecutive exceptions occur

The example in Table A-4 extends the example shown in Table A-3 on page A-426, 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 A-4 Example of basic program trace when two consecutive exceptions occur

PE execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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` Exception vector is `0x4000`	cancel_element(1) exception_element (data fault, `0x2004`) Address(`0x4000`)	The Cancel element 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 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 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 P0 element at `0x200C`. However, the trace analyzer canceled this because of the cancel element. 2. The `MOV` instruction is now implied from the Exception P0 element. The trace analyzer also generates an Address element indicating the target of the exception.
-	`IRQ`	exception_element (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.
-	commit all execution	commit_element(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.

A.3 Examples of basic program trace when execution is mispredicted

This section contains two examples that each show a different method of tracing the same piece of program code, that includes the PE mispredicting the outcome of a conditional branch instruction.

- In the first example, the trace unit signals to a trace analyzer that the branch is mispredicted by generating a Mispredict element. The Mispredict element corrects the status of the Atom element that was generated to indicate execution of the conditional branch instruction.
- In the second example, the trace unit cancels the *Atom element* that was generated for the mispredicted branch instruction, and then retraces the mispredicted branch instruction with a new *Atom element* that has the correct status.

Table A-5 shows the first example, and Table A-6 on page A-429 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 A-5 Example of basic program trace with a mispredicted branch, example one

PE execution		Trace elements	Notes
`0x1000`	`B -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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.
`0x2014`	`B -> 0x4000`	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the previous `STR` instruction and the `B` instruction.
-	`BEQ` at `0x200C` is mispredicted	cancel_element(1) mispredict_element()	The cancel element cancels the E *Atom element* that was generated for the branch at `0x2014`. The trace analyzer must discard the E *Atom element*, plus the `STR` instruction that it implied. Because the PE mispredicted the outcome of the conditional branch instruction at `0x200C`, the trace unit generates a Mispredict element to signal to the trace analyzer that the N *Atom element* that was generated has now changed to an E *Atom element*.
`0x3000`	`MOV`	-	This instruction is not traced as a P0 element, therefore no trace element is generated.
`0x3004`	`B -> 0x5000`	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the previous `MOV` instruction.
-	All instructions now committed	commit_element(3)	This commits the *Atom elements* generated for the three branch instructions at `0x1000`, `0x200C`, and `0x3004`.

Table A-6 Example of basic program trace with a mispredicted branch, example two

PE execution		Trace elements	Notes
0x1000	B -> 0x2000	trace_info_element(…) trace_on_element() context_element(…) address_element(0x1000) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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.
0x2014	B -> 0x4000	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the previous STR instruction and the B instruction.
-	BEQ at 0x200C is mispredicted	cancel_element(2)	The cancel element cancels both the E *Atom element* that was generated for the branch at 0x2014 and the N *Atom element* that was generated for the mispredicted branch at 0x200C. The trace analyzer must discard both *Atom element*s, plus all instructions that they imply.
0x200C	BEQ -> 0x3000 (taken)	atom_element(E)	The mispredicted branch is traced again with an *Atom element* that shows the correct status. That is, the branch at 0x200C is now taken, therefore is traced with an E *Atom element*. The E *Atom element* implies the execution of the four instructions at 0x2000, 0x2004, 0x2008, and 0x200C.
0x3000	MOV	-	This instruction is not traced as a P0 element, therefore no trace element is generated.
0x3004	B -> 0x5000	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the previous MOV instruction and the B instruction.
-	All instructions now committed	commit_element(3)	This commits the *Atom elements* generated for the three branch instructions at 0x1000, 0x200C, and 0x3004.

A.4 Examples of basic program trace with cycle counting enabled

This section contains two examples:

- *Basic program trace when cycle counting is enabled and the cycle count threshold is set to 16*.
- *Basic program trace when both cycle counting and global timestamping are enabled* on page A-431.

A.4.1 Basic program trace when cycle counting is enabled and the cycle count threshold is set to 16

The example in Table A-7 shows basic program trace, where only branch instructions are traced as P0 elements, with cycle counting enabled. In this example, the cycle count threshold is set to 16 and the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed

Table A-7 Basic program flow trace with cycle counting enabled, cycle count threshold set to 16

Cycle	PE execution		Trace elements	Notes
0	`0x1000`	`B -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An Address element. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
1	`0x2000`	`MOV`	-	None of these instructions are traced as P0 elements, therefore no trace elements are generated.
2	`0x2004`	`LDR`	-	
3	`0x2008`	`CMP`	-	
4	-	Commit branch at `0x1000`	commit_element(1) cycle_count_element(0)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`. Because cycle counting is enabled, the trace unit generates a Cycle Count element. However, because this is the first Cycle Count element generated, the value of the cycle count is UNKNOWN, therefore the Cycle Count element shows a value of zero.
10	`0x200C`	`BEQ -> 0x3000` (not taken)	atom_element(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.
11	`0x2010`	`STR`	-	This instruction is not traced as a P0 element, therefore no trace element is generated.
13	-	Commit branch at `0x200C`	commit_element(1)	This commits the N *Atom element* that was generated for the branch instruction at `0x200C`. No Cycle Count element is generated because there are fewer than 16 cycles since the last Commit element that had a Cycle Count element associated with it.
20	`0x2014`	`B -> 0x4000`	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` instruction and the `B` instruction.
22	-	Commit branch at `0x2014`	commit_element(1) cycle_count_element(18)	This commits the E Atom generated for the Branch instruction at `0x2014`. Because cycle counting is enabled and more than 16 cycles have passed since the last Commit element that had a Cycle Count element associated with it, the trace unit generates a Cycle Count element. The cycle count value is 18.

A.4.2 Basic program trace when both cycle counting and global timestamping are enabled

The example in Table A-8 shows basic program trace with both cycle counting and global timestamping enabled. In this example, the cycle count threshold is set to 16 and the trace unit is programmed to start tracing when the instruction at `0x1000` is accessed.

Table A-8 Basic program flow trace with cycle counting and timestamping enabled, cycle count threshold set to 16

Cycle	PE execution		Trace elements	Notes
0	`0x1000`	`B -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An Address element. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
1	`0x2000`	`MOV`	-	None of these instructions are traced as P0 elements, therefore no trace elements are generated.
2	`0x2004`	`LDR`	-	
3	`0x2008`	`CMP`	-	
4	-	Commit branch at `0x1000`	commit_element(1) cycle_count_element(0)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`. Because cycle counting is enabled, the trace unit generates a Cycle Count element. However, because this is the first Cycle Count element generated, the value of the cycle count is UNKNOWN, therefore the Cycle Count element shows a value of zero.
10	`0x200C`	`BEQ -> 0x3000` (not taken)	atom_element(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.
11	`0x2010`	`STR`	-	This instruction is not traced as a P0 element, therefore no trace element is generated.
12	-		timestamp_element (…,6)	The timestamp value in this Timestamp element corresponds to the time of the N *Atom element* that was generated at cycle number 10. Because cycle counting is enabled, the Timestamp element contains a cycle count value. The value of the cycle count is six because there are six cycles between the N *Atom element* at cycle number 10, and the last Commit element that had a Cycle Count element associated with it, that is at cycle number four.
13	-	Commit branch at `0x200C`	commit_element(1)	This commits the N *Atom element* that was generated for the branch instruction at `0x200C`. No Cycle Count element is generated because there are fewer than 16 cycles since the last Commit element that had a Cycle Count element associated with it.

Table A-8 Basic program flow trace with cycle counting and timestamping enabled, cycle count threshold set to 16

Cycle	PE execution		Trace elements	Notes
20	`0x2014`	`B -> 0x4000`	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` instruction and the `B` instruction.
22	-	Commit branch at `0x2014`	commit_element(1) cycle_count_element (18)	This commits the E Atom generated for the Branch instruction at `0x2014`. Because cycle counting is enabled and more than 16 cycles have passed since the last Commit element that had a Cycle Count element associated with it, the trace unit generates a Cycle Count element. The cycle count value is 18.
24	-		timestamp_element (…,16)	The timestamp value in this Timestamp element corresponds to the time of the E *Atom element* that was generated at cycle number 20. The value of the cycle count that the Timestamp element contains is 16, because there are 16 cycles between the E *Atom element* at cycle number 20, and the last Commit element that had a Cycle Count element associated with it, that is at cycle number four.

A.5 Examples of basic program trace with filtering applied

This section is split into the following:

- *An example of basic program trace with filtering applied*.
- *Examples of basic program trace with filtering applied when an exception occurs* on page A-434.

A.5.1 An example of basic program trace with filtering applied

The example in Table A-9 shows basic program trace, where only branch instructions are traced as P0 elements, 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 A-9 Basic program trace with filtering applied

PE execution		Traced?	Trace elements	Notes
`0x1000`	`B -> 0x2000`	Yes	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An Address element. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
`0x2000`	`MOV`	No	-	The filtering applied to the trace means that none of these instructions are traced.
`0x2004`	`LDR`	No	-	
`0x2008`	`CMP`	No	-	
`0x200C`	`BEQ -> 0x3000` (not taken)	No	-	
`0x2010`	`STR`	Yes	trace_on_element() address_element(`0x2010`)	This instruction is not traced as a P0 element, therefore no trace element is generated. However, tracing of this instruction is required, so the trace unit generates a Trace On element and an Address element that contains the address of this instruction.
`0x2014`	`BEQ -> 0x4000` (taken)	Yes	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` instruction and the `BEQ` instruction.
-	Cancel back to and including `0x2000`	-	cancel_element(1)	This cancels the E *Atom element* that was generated for the branch at `0x2014`. The trace analyzer must discard the E *Atom element*, plus the `STR` instruction that it implied.
-	commit all execution	-	commit_element(1)	This commits the E Atom generated for the Branch instruction at `0x1000`.

A.5.2 Examples of basic program trace with filtering applied when an exception occurs

This section contains three examples that show trace behavior when filtering is applied and when the piece of code includes an exception.

The examples demonstrate that whether the exception is traced depends on whether the filtering applied permits tracing of the most recent instruction prior to the exception. That is:

- If the instruction executed prior to the exception is traced, then the trace unit must generate an Exception element for the exception. The trace unit must generate the *Exception element* regardless of whether ViewInst is active or inactive at the time when the exception occurs.
- If the instruction executed prior to the exception is not traced because it is filtered out of the trace, then no *Exception element* is generated.

Table A-10 summarizes what each of the examples show.

Table A-10 Summary of what the examples in this section show

Example number	Is ViewInst active when the exception occurs?	Was the instruction prior to the exception traced?	Is an Exception element generated?
One, see Table A-11 on page A-435	Y	Y	Y
Two, see Table A-12 on page A-436	N	Y	Y
Three, see Table A-13 on page A-437	N	N	N

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.

Note

- In the example shown in Table A-11 on page A-435, the trace unit is programmed to exclude all instructions in the address range `0x2000` to `0x200F` inclusive.
- In the examples shown in Table A-12 on page A-436 and Table A-13 on page A-437, the trace unit is programmed to exclude all instructions in the address range `0x2000` to `0x2017` inclusive.

Table A-11 Basic program trace with filtering when an exception occurs, example one

PE execution		Traced?	Trace elements	Notes
`0x1000`	`B -> 0x2000`	Yes	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An Address element. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
`0x2000`	`MOV`	No	-	The filtering applied to the trace means that none of these instructions are traced.
`0x2004`	`LDR`	No	-	
`0x2008`	`CMP`	No	-	
`0x200C`	`BEQ -> 0x3000` (not taken)	No	-	
`0x2010`	`STR`	Yes	trace_on_element() address_element(`0x2010`)	This instruction is not traced as a P0 element, therefore no trace element is generated. However, tracing of this instruction is required, so the trace unit generates a Trace On element and an Address element that contains the address of this instruction.
`0x2014`	`BEQ -> 0x4000` (taken)	Yes	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` instruction and the `BEQ` instruction.
-	Cancel back to and including `0x2000`	-	cancel_element(1)	This cancels the E *Atom element* that was generated for the branch at `0x2014`. The trace analyzer must discard the E *Atom element*, plus the `STR` instruction that it implied.
-	`IRQ`	-	address_element(`0x2000`) exception_element(IRQ, `0x2000`)	The address element indicates where execution has returned to before the exception. Because the PE has canceled all execution between `0x2014` and `0x2000`, including the instructions at `0x2014` and `0x2000`, then this means that the instruction executed prior to the exception is the Branch instruction at `0x1000`. Because this branch was traced, the trace unit must generate an Exception element for the exception.
-	commit all execution	-	commit_element(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.

Table A-12 Basic program trace with filtering when an exception occurs, example two

PE execution		Traced?	Trace elements	Notes
`0x1000`	`B -> 0x2000`	Yes	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An Address element. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
`0x2000`	`MOV`	No	-	The filtering applied to the trace means that none of these instructions are traced.
`0x2004`	`LDR`	No	-	
`0x2008`	`CMP`	No	-	
`0x200C`	`BEQ -> 0x3000` (not taken)	No	-	
`0x2010`	`STR`	No	-	
`0x2014`	`BEQ -> 0x4000` (taken)	No	-	
-	Cancel back to and including `0x2000`	-	-	No Cancel element is generated because there is no trace to cancel.
-	`IRQ`	-	exception_element(IRQ, `0x2000`)	Because the PE has canceled all execution between `0x2014` and `0x2000`, including the instructions at `0x2014` and `0x2000`, then this means that the instruction executed prior to the exception is the Branch instruction at `0x1000`. Because this branch was traced, the trace unit must generate an Exception element for the exception.
-	commit all execution	-	commit_element(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.

Table A-13 Basic program trace with filtering when an exception occurs, example three

PE execution		Traced?	Trace elements	Notes
`0x1000`	`B -> 0x2000`	Yes	trace_info_element(…) trace_on_element() context_element(…) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An Address element. The instruction executed is a taken branch, so in addition, the trace unit must generate an E Atom element.
`0x2000`	`MOV`	No	-	The filtering applied to the trace means that none of these instructions are traced.
`0x2004`	`LDR`	No	-	
`0x2008`	`CMP`	No	-	
`0x200C`	`BEQ -> 0x3000` (not taken)	No	-	
`0x2010`	`STR`	No	-	
`0x2014`	`BEQ -> 0x4000` (taken)	No	-	
-	Cancel back to and including `0x2010`	-	-	No Cancel element is generated because there is no trace to cancel.
-	`IRQ`	-	-	Because the PE has canceled the instructions at `0x2014` and `0x2010`, this means that the instruction executed prior to the exception is the conditional branch instruction at `0x200C`. Because this branch was not traced, no Exception element is generated.
-	commit all execution	-	commit_element(1)	This commits the E *Atom element* that was generated for the Branch instruction at `0x1000`.

A.6 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 A-14 shows the first example, and Table A-15 on page A-439 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 A-14 Basic program trace when Branch with Link instructions are executed and the return stack is disabled

PE execution		Trace elements	Notes
`0x1000`	`BL -> 0x2000`	trace_info_element(…) trace_on_element() context_element(…) address_element (`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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`	`BLEQ -> 0x3000` (not taken)	atom_element(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`	-	This instruction is not traced as a P0 element, therefore no trace element is generated.
`0x2014`	`BX LR`	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the `STR` instruction and the `BX` instruction.
`0x1004`	`MOV`	address_element (`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 Address element to indicate the target of that branch.
`0x1008`	`B -> 0x4000`	atom_element(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_element(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 A-15 Basic program trace when Branch with Link instructions are executed and the return stack is enabled

PE Execution		Trace elements	Notes
0x1000	BL -> 0x2000	trace_info_element(…) trace_on_element() context_element(…) address_element (0x1000) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An Address element. The instruction executed is a taken branch, so the trace unit must also 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	-	
0x200C	BLEQ -> 0x3000 (not taken)	atom_element(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	-	This instruction is not traced as a P0 element, therefore no trace element is generated.
0x2014	BX LR	atom_element(E)	This branch is taken, so the trace unit generates an E *Atom element*. The E *Atom element* implies the execution of the STR instruction and the BX instruction.
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 Address element is not required. The top entry on the return stack, address 0x1004, is popped from the return stack.
0x1008	B -> 0x4000	atom_element(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_element(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.

A.7 Examples of operations that change the execution context

When the 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. See *Context instruction trace element* on page 5-211 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* on page A-441.
- *Exception immediately before ISB* on page A-442.

A.7.1 Exception in software executed after context synchronization

Table A-16 shows a write to the CONTEXTIDR register, followed by an `ISB` to synchronize that write, followed by an exception that changes the context again.

Table A-16 Program trace containing a context changing operation

PE execution		Context ID	Trace elements	Notes
`0x1000`	`B-> 0x2000`	`0xAA`	trace_info_element(…) trace_on_element() context_element(`0xAA`) address_element(`0x1000`) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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_element(`0xBB`)	A Context element is traced to indicate the new context.
-	`IRQ`	`0xBB`	exception_element(IRQ,`0x2010`)	An IRQ exception occurs. The trace unit generates an *Exception element*.
`0x3000`	`B -> 0x4000`	`0xCC`	context_element(`0xCC`) address_element(`0x3000`) atom_element(E)	A Context element is traced to indicate the new context. An Address element is also traced, because an *Exception element* is always followed by an 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*.

A.7.2 Exception immediately after ISB

Table A-17 shows the same execution as Table A-16 on page A-440 but the exception occurs one instruction earlier. This means that no execution takes place between the ISB and the exception.

Table A-17 Program trace containing a context changing operation (exception immediately after ISB)

PE execution		Context ID	Trace elements	Notes
0x1000	B -> 0x2000	0xAA	trace_info_element(…) trace_on_element() context_element(0xAA) address_element(0x1000) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(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_element(0xBB) exception_element(IRQ,0x200C)	A Context element is traced to indicate the new context. An IRQ exception occurs. The trace unit generates an *Exception element* but no execution is implied since the target of the ISB.
0x3000	B -> 0x4000	0xCC	context_element(0xCC) address_element(0x3000) atom_element(E)	A Context element is traced to indicate the new context. An Address element is also traced, because an *Exception element* is always followed by an 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*.

A.7.3 Exception immediately before ISB

Table A-18 shows the same as Table A-17 on page A-441 but the exception occurs one instruction earlier. This means that the exception occurs before the ISB instruction that was present in previous examples.

Table A-18 Program trace containing a context changing operation (exception immediately before ISB)

PE execution		Context ID	Trace elements	Notes
0x1000	B -> 0x2000	0xAA	trace_info_element(…) trace_on_element() context_element(0xAA) atom_element(E)	Tracing begins here, therefore the trace unit must generate both: • A Context element. • An 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_element(IRQ,0x2008)	An IRQ exception occurs. The trace unit generates an *Exception element*.
0x3000	B -> 0x4000	0xCC	context_element(0xCC) address_element(0x3000) atom_element(E)	A Context element is traced to indicate the new context. An Address element is also traced, because an *Exception element* is always followed by an 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*.

A.7.4 Tracing branch future instructions

The branch future instructions are traced using *Atom elements* and the implicit branches caused by branch future instructions are also traced using *Atom elements*. The following examples show how the tracing of branch futures instructions interacts with filtering and periodic synchronization.

Typical branch future trace

Table A-19 shows the tracing of a BF instruction and the implicit branch.

Table A-19 Tracing of a BF instruction and the implicit branch

PE Execution	Trace elements	Notes
0x1000 BF 0x1010,0x2000	trace_info_element()	
	trace_on_element()	
	context_element()	
	address_element(0x1000)	
	atom_element(E)	
0x1004 MOV		
0x1008 MOV		
0x100C CMP		
Implicit branch due to the BF	atom_element(E)	*Atom element* due to the implicit branch
0x2000 (continue execution)	(tracing continues)	

Branch future trace when branch future is disabled

Table A-20 shows a sequence of elements generated for a BF instruction where the branch future behavior is not enabled or is not implemented. TRCIDR0.BF is 0b01 in this example.

Table A-20 Tracing of a BF instruction with branch future disabled

PE Execution	Trace elements	Notes
0x1000 BF 0x1010,0x2000	trace_info_element()	The state of LO_BRANCH_INFO.VALID is not changed in the PE so the BF instruction is traced as an N atom.
	trace_on_element()	
	context_element()	
	address_element(0x1000)	
	atom_element(N)	
0x1004 MOV		
0x1008 MOV		
0x100C CMP		
0x1010 B 0x2000	atom_element(E)	
0x2000 (continue execution)	(tracing continues)	

Multiple branch future instructions

It is possible to have multiple branch future instructions where the some of the branch future instructions do not update LO_BRANCH_INFO. Table A-21 shows how such a scenario could be traced.

Table A-21 Tracing multiple branch future instructions

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x0FFC BFNE 0x1010,0x3000	atom_element(E)	The BFNE passes its condition code check and LO_BRANCH_INFO.VALID is set to 1 by this instruction so the BFNE instruction is traced as an E *Atom element*.
0x1000 BFEQ 0x1010,0x2000	atom_element(N)	The BFEQ fails its condition code check so LO_BRANCH_INFO.VALID is not updated by this instruction and this instruction is traced as an N *Atom element*.
0x1004 MOV		
0x1008 MOV		
0x100C CMP		
Implicit branch due to the BFNE	atom_element(E)	*Atom element* due to the implicit branch. This implicit branch is a direct branch to 0x3000 and does not require an *Address element* to be generated for the target of the implicit branch.
0x3000 (continue execution)	(tracing continues)	

Branch future flush

Table A-22 shows a scenario where LO_BRANCH_INFO is invalidated in the PE, before reaching the BF branch point.

Table A-22 Branch future flush

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x1000 BF 0x1010,0x2000	atom_element(E)	LO_BRANCH_INFO.VALID is set in the PE so the BF instruction is traced with an E *Atom element*.
0x1004 MOV		
LO_BRANCH_INFO.VALID is cleared in the PE	branch_future_flush()	A *Branch Future Flush element* is generated because the BF instruction was traced
0x1008 MOV		
0x100C CMP		
0x1010 B.cond 0x2000	atom_element()	There is no implicit branch because LO_BRANCH_INFO.VALID is clear. The *Atom element* generated here is based on the condition code check status of this branch instruction.

Branch future flush with other Atom elements

Table A-23 shows a scenario where LO_BRANCH_INFO is invalidated in the PE, before reaching the BF branch point. Furthermore, other *Atom elements* are generated between the branch future instruction and the Branch Future Flush element.

Table A-23 Branch future flush with other Atom elements

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x1000 BF 0x1010,0x2000	atom_element(E)	LO_BRANCH_INFO.VALID is set in the PE so the **BF** instruction is traced with an E *Atom element*.
0x1004 B 0x100C	atom_element(E)	
LO_BRANCH_INFO.VALID is cleared in the PE	branch_future_flush()	A *Branch Future Flush element* is generated because the **BF** instruction was traced
0x100C CMP		There is no implicit branch because LO_BRANCH_INFO.VALID is clear.
0x1010 B.cond 0x2000	atom_element(N)	The *Atom element* generated here is based on the condition code check status of this branch instruction.

Branch future with instruction filtering

Table A-24 shows a scenario where the branch future instruction was not traced due to ViewInst filtering.

Table A-24 Branch future with instruction filtering

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x1004 MOV	trace_on_element()	Tracing becomes active on this MOV instruction.
	context_element()	
	address_element(0x1004)	
0x1008 MOV		
0x100C CMP		
Implicit branch due to an earlier BF	resynchronization()	The *Resynchronization element* is generated because an implicit branch occurs but the corresponding branch future instruction was not traced.
	atom_element(E)	This *Atom element* is generated for the implicit branch, even though the trace unit generated a *Resynchronization element.*
0x2000 (continue execution)	address_element(0x2000)	The *Address element* is generated because the *Resynchronization element* was generated.

Branch future with periodic synchronization

Table A-25 shows a scenario where the trace analyzer might start analysis in the middle of a trace stream. Trace analysis starts at a *Trace Info element,* which occurs between a branch future instruction and the subsequent implicit branch, and a *Resynchronization element* is needed.

Table A-25 Branch future with periodic synchronization

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x1000 BF 0x1010,0x2000	atom_element(E)	
0x1004 MOV	trace_info_element()	Periodic synchronization occurs here.
0x1008 MOV		
0x100C CMP		
Implicit branch	resynchronization()	The BF instruction was traced before the *Trace Info element* and therefore a *Resynchronization element* is generated.
	atom_element(E)	This *Atom element* is generated for the implicit branch, even though the trace unit generated a *Resynchronization element*.
0x2000 (continue execution)	address_element(0x2000)	The *Address element* is generated due to both the *Resynchronization element* and the *Trace Info element*.

Branch future with periodic synchronization and other address elements

One of the requirements of periodic synchronization is the tracing the address of the target of a *P0 element*. This address might not be sufficient when a `BF` instruction has been traced. This is an example where an additional *Address element* must be traced due to an implicit branch.

Table A-26 Branch future with periodic synchronization and other address elements

PE Execution	Trace elements	Notes
`(previous execution)`	`(previous trace)`	
`0x1000 BF 0x1010,0x2000`	`atom_element(E)`	
`0x1004 MOV`	`trace_info_element()`	Periodic synchronization occurs here.
	`context_element()`	
	`address_element(0x1004)`	This *Address element* was generated due to the *Trace Info element*.
`0x1008 MOV`		
`0x100C CMP`		
Implicit branch	`resynchronization()`	A *Resynchronization element* is required because the tracing of the **`BF`** was before the *Trace Info element*.
	`atom_element(E)`	This *Atom element* is generated for the implicit branch, even though the trace unit generated a *Resynchronization element*.
`0x2000 (continue execution)`	`address_element(0x2000)`	The *Address element* is generated due the *Resynchronization element*.

A.7.5 Tracing WLS and LE loops

Typical WLS, LE trace

Table A-27 Typical WLS, LE trace

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x0FFC MOV r3, #0x3		
0x1000 WLS r14, r3, 0x1010	atom_element(N)	
0x1004 LDR		
0x1008 STR		
0x100C LE r14, 0x1004	atom_element(E)	
0x1004 LDR		
0x1008 STR		
Implicit LE instruction	atom_element(E)	The E *Atom element* is generated due to the implicit LE instruction.
0x1004 LDR		
0x1008 STR		
Implicit LE instruction	atom_element(N)	The N *Atom element* is generated due to the implicit LE instruction, and this time the LE is not taken and execution exits the loop.
0x1010 (execution continues)	(tracing continues)	

A.7.6 Tracing vector instructions

Typical partial execution of a single vector instruction

Table A-28 shows an exception occurring when a single vector instruction is partially executed.

Table A-28 Typical partial execution of a single vector instruction

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x1000 VLDR		
Exception, where RETPSR.ECI = 0b00000001	exception(0x1002)	The VLDR has executed a single beat, so the *Exception element* indicates the VLDR has been executed.
(exception handler)	(trace)	
LDR PC (EXC_RETURN)	atom_element(E)	Execution returns from the exception, to resume the VLDR.
0x1000 VLDR	address_element(0x1000)	Execution returns to the VLDR.
0x1004 B 0x2000	atom_element(E)	This *Atom element* indicates the VLDR has been executed.

A.7.7 Typical partial execution of two vector instructions

Table A-29 shows an exception occurring when two vector instructions have been partially executed.

Table A-29 Typical partial execution of two vector instructions

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x1000 VLDR		
0x1004 VMUL		
Exception, where RETPSR.ECI = 0b00000101	exception(0x1006)	Both the VLDR and VMUL have executed at least a single beat, so the *Exception element* indicates the VLDR and VMUL have both been executed.
(exception handler)	(trace)	
LDR PC (EXC_RETURN)	atom_element(E)	Execution returns from the exception, to resume the VLDR.
0x1000 VLDR	address_element(0x1000)	Execution returns to the VLDR.
0x1004 VMUL		
0x1008 B 0x2000	atom_element(E)	This *Atom element* indicates the VLDR and VMUL have been executed.

A.7.8 Typical partial execution of two vector instructions with a Loop End

Table A-30 shows an exception occurring when two vector instructions have partially executed, and an implicit branch due to a Loop End instruction occurred between the vector instructions.

The Loop End instruction appears to be executed twice, both before and after the exception, even though this is only a single iteration of the loop.

Note that the *Exception element* preferred exception return address is not related to the return address stored on the stack. Instead, the preferred exception return address is of the youngest partially executed instruction plus 0x2.

Table A-30 Typical partial execution of two vector instructions with a Loop End

PE Execution	Trace elements	Notes
(previous execution)	(previous trace)	
0x1040 VLDR		
0x1044 LE LR, 0x1000	atom_element(E)	This *Atom element* indicates the LE instruction has been taken, and the VLDR was executed. The LE returns to the start of the loop at 0x1000.
0x1000 VMUL		
Exception, where RETPSR.ECI = 0b00000101	exception(0x1002)	Both the VLDR and VMUL have executed at least a single beat, so the *Exception element* indicates the VLDR and VMUL have both been executed.
(exception handler)	(trace)	
LDR PC (EXC_RETURN)	atom_element(E)	Execution returns from the exception, to resume the VLDR.
0x1040 VLDR	address_element(0x1040)	Execution returns to the VLDR.

Table A-30 Typical partial execution of two vector instructions with a Loop End (continued)

PE Execution	Trace elements	Notes
`0x1044 LE LR, 0x1000`	`atom_element(E)`	This *Atom element* indicates the `LE` and the `VLDR` have been executed.
`0x1000 VMUL`		
`0x1004 B 0x2000`	`atom_element(E)`	This *Atom element* indicates the `VMUL` has been executed, plus the branch instruction.

Appendix B
Required Architecture Versions

This appendix lists the required version of the ETM architecture for each version of the Arm architecture. It contains the following sections:

B.1 Required ETM architecture version for each version of the Arm architecture

Depending on the version of the Arm architecture implemented, there are rules that dictate which version of the ETM architecture is required to support the Arm architecture.

B.1.1 Armv8-A architecture requirements

If the Armv8-A architecture is implemented, the following rules apply:

- If the PE implements Armv8.3-DoPD, the trace unit must implement ETMv4.2 and must implement the Unified Power Domain model.
- If the PE implements Secure EL2, at least ETMv4.4 is required.
- If the PE implements Armv8.4-Trace, at least ETMv4.4 is required.
- If the PE implements Pointer Authentication, at least ETMv4.3 is required.
- If the virtual address is larger than 48 bits, at least ETMv4.2 is required.
- If the Virtualization Host Extensions are implemented, at least ETMv4.1 is required.
- If the VTTBR.VMID is larger than 8 bits, at least ETMv4.1 is required.
- In all other cases, ETMv4.0 is sufficient.

B.1.2 Armv8-R architecture requirements

If the Armv8-R architecture is implemented, the following rule applies:

- At least ETMv4.2 is required.

B.1.3 Armv8-M architecture requirements

If the Armv8-M architecture is implemented, the following rules apply:

- If the Security Extension is implemented, at least ETMv4.2 is required.
- In all other cases, ETMv4.0 is sufficient.
- If the PE implements the LOB feature, the trace unit must have TRCIDR0.BF equal to 0b01.
- If the PE implements the LOB feature, the trace unit must implement ETMv4.5.

B.1.4 Armv7-A or Armv7-R architecture requirements

If the Armv7-A or Armv7-R architecture is implemented, the following rule applies:

- At least ETMv4.0 is required.

B.1.5 Armv7-M or Armv6-M architecture requirements

If the Armv7-M or Armv6-M architecture is implemented, the following rule applies:

- At least ETMv4.0 is required.

Appendix C
Recommended Configurations

This appendix contains a set of recommended configurations for trace unit implementations. It contains the following sections:

C.1 Configuration overview

Arm recommends the following configurations:

1. Basic program flow with cycle counting and conditional instruction trace

This configuration:

- Relies on PE comparators.
- Is targeted at low-cost markets.

Arm expects that this configuration is used with the Armv7-M and Armv8-M architecture profiles.

2. Basic program flow with cycle counting

In this configuration, the trace unit has its own comparators.

Arm expects that this configuration is used with the Armv7-A and Armv8-A architecture profiles.

3. Full instruction and data trace

Arm expects that this configuration is used with the Armv7-R, Armv8-R, Armv7-M, and Armv8-M architecture profiles.

C.2 Configuration parameters

Table C-1 shows the configuration parameters for the recommended configurations.

In Table C-1:

Yes Indicates that the feature is implemented.

No Indicates that the feature is not implemented.

IMP DEF Indicates that it is IMPLEMENTATION DEFINED whether the feature is implemented.

Table C-1 Recommended configurations

Parameter	Description	Configuration		
		1	2	3
ATBTRIG	ATB trigger support	Yes	Yes	Yes
CCITMIN	Instruction trace cycle counting minimum threshold Cycle counter size, in bits	4	4	4
CCSIZE	Cycle counter size in bits	≥12	≥12	≥12
CIDSIZE	Context ID size in bytes	0	4	4
COMMOPT	Commit mode	IMP DEF[a]	IMP DEF[a]	IMP DEF[a]
CONDTYPE	Method for tracing conditional instruction results	IMP DEF	N/A	Yes
DASIZE	Data address size in bytes	0	0	4
DVSIZE	Data value size in bytes	0	0	4
EXLEVEL_NS	Exception levels that are implemented in Non-secure state	IMP DEF	IMP DEF	IMP DEF
EXLEVEL_S	Exception levels that are implemented in Secure state	IMP DEF	IMP DEF	IMP DEF
IASIZE	Instruction address size in bytes	4	4 or 8	4
INSTP0	Support for explicit tracing of data load and store instructions	No	No	Yes
LPOVERRIDE	Low-power behavior override	IMP DEF[b]	IMP DEF[b]	IMP DEF[b]
NOOVERFLOW	Support for overflow avoidance	IMP DEF	IMP DEF	IMP DEF
NUMACPAIRS	Number of address comparator pairs	0	4	4
NUMCIDC	Number of Context ID comparators	0	1	1
NUMCNTR	Number of counters	1	2	2
NUMDVC	Number of data value comparators	0	0	2
NUMEVENT	Number of events that are supported in the trace	2	4	4
NUMEXTIN	Number of external inputs	≥2[e]	≥4[e]	≥4[e]
NUMEXTINSEL	Number of external input selectors	0 or 2[c]	0 or 4	0 or 4
NUMPC	Number of PE comparator inputs	IMP DEF[d]	0	0
NUMPROC	Number of processes available for tracing	IMP DEF	IMP DEF	IMP DEF
NUMRSPAIR	Number of resource selection pairs	2	8	8
NUMSEQSTATE	Number of sequencer states	0	4	4
NUMSSCC	Number of single-shot comparator controls	IMP DEF	≥1[e]	≥1[e]
NUMVMIDC	Number of Virtual context identifier comparators	0	IMP DEF[k]	IMP DEF[k]
OS Lock	OS Lock	IMP DEF[f]	Yes	IMP DEF[f]

Table C-1 Recommended configurations (continued)

Parameter	Description	Configuration		
		1	2	3
QFILT	Q element filtering support	IMP DEF	IMP DEF	IMP DEF
QSUPP	Q element support	IMP DEF	IMP DEF	IMP DEF
REDFUNCNTR	Reduced function counter	Yes	No	No
RETSTACK	Return stack support	Yes	Yes	Yes
STALLCTL	Stall control support	Yes	Yes	Yes
SUPPDAC	Data address comparators	No	No	Yes
SW Lock	Software Lock	IMP DEF[g]	IMP DEF[g]	IMP DEF[g]
SYNCPR	Synchronization period support	RO	RW	RW
SYSSTALL	System supports stall control	IMP DEF	IMP DEF	IMP DEF
TRACEIDSIZE	Size of trace ID	7 bits	7 bits	7 bits
TRCBB	Support for branch broadcast tracing	Yes	Yes	Yes
TRCCCI	Support for cycle counting in the instruction trace	Yes	Yes	Yes
TRCCOND	Support for conditional instruction tracing	Yes	No	Yes
TRCDATA	Support for tracing of data	No	No	Yes
TRCERR	Support for tracing of System errors	Yes	Yes	Yes
TRCEXDATA	Tracing of exception data transfers	No	No	IMP DEF[h]
TSSIZE	Global timestamp size, bits	64[i]	64[i]	64[i]
VMIDOPT	*Virtual context identifier* choice	No	IMP DEF[j]	IMP DEF[j]
VMIDSIZE	*Virtual context identifier* size	0	IMP DEF[k]	IMP DEF[k]
WFXMODE	`WFx` instructions are classified as branch instructions	IMP DEF	IMP DEF	IMP DEF

a. COMMOPT is dependent on the maximum speculation depth of the trace unit.
b. The low-power behavior override causes the trace unit to remain active and responsive to input when the PE or other parts of the system might be in a low-power state. As a result, it can continue to trace events that occur during this time.
c. If the number of external inputs is 4 or less, a trace unit might implement zero external input selectors. The external inputs are then flat-mapped through to the resource selectors.
d. The number of PE comparator inputs must be the same as the number of DWT comparators on Armv7-M and Armv8-M PEs.
e. The actual value is IMPLEMENTATION DEFINED.
f. The OS Lock is optional on Armv6-M, Armv7-M, and Armv8-M processors. Arm recommends that the OS Lock is not implemented on trace units for Armv6-M, Armv7-M and Armv8-M PEs.
g. The Software Lock is optional and implementation of the Software Lock is deprecated.
h. TRCEXDATA only implemented if data trace is implemented, and only on Armv6-M, Armv7-M, or Armv8-M processors.
i. Recommended value. Actual value is IMPLEMENTATION DEFINED.
j. From ETMv4.1, if the PE implements the Virtualization Host Extensions then VMIDOPT is implemented.
k. This depends on whether the PE supports the Virtualization Extensions:
   - TRCIDR2.VMIDSIZE defines the architectural requirements for VMIDSIZE.
   - If the PE implements EL2 or the Virtualization Extensions, then NUMVMIDC is 1.

Appendix D
Filtering Examples

This appendix gives examples of instruction address range filtering, and describes the typical trace output for each example. It contains the following section:

- *About the filtering examples on page D-458*.

D.1 About the filtering examples

The following examples assume the existence of a contiguous block of instructions, from the target of a P0 element, up to and including the next P0 element. The target of the first P0 element is address A, and the address of the second P0 element is B. There are no P0 elements between address A and address B.

These examples use a single address range comparator, programmed with low address ADDR_LOW and high address ADDR_HIGH.

The instruction address range comparators are not the only function of the trace unit that control whether or not a particular instruction block is traced. See Figure 4-1 on page 4-119. These examples assume that the start/stop control is active and that the imprecise enabling event is active. The instruction address range comparators then control whether or not the trace unit generates trace for a particular instruction.

Note

- These same examples can be applied independently to every block of contiguous instructions bounded by two P0 elements. These examples do not describe the behavior of the trace unit for the blocks before or after the block from address A to address B. These other blocks must be considered independently from the block from address A to address B.
- In some trace unit implementations, address B might be of the first byte of the second P0 element instruction, and in some implementations address B might be of the last byte of the second P0 element instruction.

The following sections describe these filtering examples:

- *Comparator range covers the complete block of instructions*.
- *Comparator range covers neither address A nor address B*.
- *Comparator range covers address A but not address B on page D-459*.
- *Comparator range covers address B but not address A on page D-459*.

D.1.1 Comparator range covers the complete block of instructions

This example describes the behavior of an address range comparator programmed with an address range that includes the complete range of addresses between address A and address B. That is, a comparator programmed so that:

ADDR_LOW ≤ (address A) < (address B) ≤ ADDR_ HIGH

If the ViewInst function is programmed to include the region selected by the address range comparator, then the entire block of instructions from address A to address B is traced.

If the ViewInst function is programmed to exclude the region selected by the address range comparator, then the entire block of instructions from A to B is not traced.

D.1.2 Comparator range covers neither address A nor address B

This example describes the behavior of an address range comparator programmed to select a region between address A and address B.

(address A) ≤ ADDR_LOW < ADDR_HIGH ≤ (address B)

If the ViewInst function is programmed to include the region selected by the address range comparator, then the block of instructions from ADDR_LOW to ADDR_HIGH is traced. This means:

- Some implementations trace the range of instructions from ADDR_LOW to address B, because the only way to trace the instructions from ADDR_LOW to ADDR_HIGH is to trace until the next P0 element.
- Some implementations might trace the entire block of instructions from address A to address B.

If the ViewInst function is programmed to exclude the region selected by the address range comparator, then the instructions from address A to ADDR_LOW and from ADDR_HIGH to address B are traced. This means that the entire block of instructions from address A to address B is traced, because the only way to trace from address A to ADDR_LOW is to trace until the next P0 element. This means that tracing is continually active from address A to address B, and therefore this would not constitute a discontinuity in the trace.

D.1.3 Comparator range covers address A but not address B

This example describes the behavior that results from programming an address range comparator to cover the beginning of the region between address A and address B, but not the end.

ADDR_LOW ≤ (address A) < ADDR_HIGH < (address B)

If the ViewInst function is programmed to include the region selected by the address range comparator, then the range of instructions from address A to ADDR_HIGH is traced. The range of instructions from address A to address B is traced, because the only way to trace the instructions from address A to ADDR_HIGH is to trace up to the next P0 element.

If the ViewInst function is programmed to exclude the region selected by the address range comparator, then the range of instructions from ADDR_HIGH to address B is traced. Some implementations might trace the entire block of instructions from address A to address B.

D.1.4 Comparator range covers address B but not address A

This example describes the behavior that results from programming an address range comparator to cover the end of the region between address A and address B, but not the beginning.

(address A) < ADDR_LOW < (address B) ≤ ADDR_HIGH

If the ViewInst function is programmed to include the region selected by the address range comparator, then the range of instructions from ADDR_LOW to address B is traced. Some implementations might trace the entire block of instructions from address A to address B.

If the ViewInst function is programmed to exclude the region selected by the address range comparator, then the instructions from address A to ADDR_LOW are traced. The range of instructions from address A to address B is traced, because the only way to trace the instructions from address A to ADDR_LOW is to trace up to the next P0 element.

Appendix E
Resource Selection Examples

This appendix gives programming examples for the ETMv4 resource selectors. It contains the following sections:

- *Programming the ETMv4 to assert an external output on SAC0 or SAC1 on page E-462*.
- *Programming the ETMv4 to set the ViewInst filter on SAC5 or Counter 1 at 0 on page E-463*.

E.1 Programming the ETMv4 to assert an external output on SAC0 or SAC1

This example shows how the ETMv4 macrocell can be programmed to generate an external signal if the PE accesses either of two addresses. This is done by programming a resource selector to signal if either of two address comparators matches, then programming the selector to drive an external output. The resource selector is controlled by the relevant TRCRSCTLRn register, where n is the number of the resource selector to be programmed.

This example uses address comparators 0 and 1, and resource selector 2:

1. Program resource selector 2 to select resources 0 and 1 from group four, with no result inversion. This selects single address comparator 0 and single address comparator 1, causing a signal to be generated when either address comparator matches.

   To do this, set TRCRSCTLR2 to `0x00040003`.

2. Program external output zero of the ETMv4 to be asserted when resource selector 2 generates a signal. Set the bit in TRCEVENTCTL0R.EVENT0 that corresponds to resource selector 2.

   To do this, set TRCEVENTCTL0R.EVENT0 to `0x02`.

E.2 Programming the ETMv4 to set the ViewInst filter on SAC5 or Counter 1 at 0

This example shows how the ETMv4 macrocell can be programmed to trace instructions when an address comparison matches, or when a counter reaches 0. This operation cannot be performed by a single resource selector because the resources to be selected are from different groups. This means a resource selector pair must be used. The first resource selector is programmed to signal when single address comparator 5 matches the specified address, and the second resource selector is programmed to signal when counter 1 reaches zero. However, as Figure 4-21 on page 4-176 shows, the output from a resource selector pair is the logical AND of the two resource selector outputs, and this example requires the logical OR.

DeMorgan's Theorem gives a solution to this problem. This states that:

!A && !B = !(A || B)

The truth table in Table E-1 shows this theorem.

Table E-1 Truth table showing DeMorgans's theorem

A	B	!A	!B	!A && !B	!(!A && !B)	A || B
0	0	1	1	1	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	0	1	1

In Table E-1, A and B are the outputs of the resource selectors for the address comparison and the counter. These can be inverted by setting the appropriate TRCRSCTLRn bits.

The resource selector pair gives the logical AND of these inverted outputs. As Table E-1 shows, inverting the output of the resource selector pair gives the required OR result.

To implement this example:

1. Select a resource selector pair from those available. This example uses the resource selector pair composed from resource selectors 2 and 3.

   To use resource selector 2 for the address comparison:

   - Program resource selector 2 to select resource 5 from group four, and set the result inversion bit to 1. This selects single address comparator 5, and inverts the output.
   - Set the TRCRSCTLR2.PAIRINV bit to 1. This inverts the output from the resource selector pair.

   To do this, set TRCRSCTLR2 to `0x00340020`.

2. Use resource selector 3 for the counter. Program resource selector 3 to select resource 1 of group two, and set the result inversion bit. This selects counter one, and inverts the output.

   This is the second resource selector of a pair. TRCRSCTLRn.PAIRINV bit applies only to the first selector of a pair, so it is ignored here.

   To program this selector, set TRCRSCTLR3 to `0x00120001`.

3. Program the ViewInst Main Control Register TRCVICTLR so that instruction trace is generated when the resource selector pair generates a signal.

   To do this, in the TRCVICTLR.EVENT field, set:

   - The SEL subfield to the number of the resource selector pair programmed in the previous two steps.
   - The TYPE bit to 1, indicating that the value of the SEL field is a resource selector pair.

Appendix F
Instruction Categories

This appendix shows the categorization of the instructions from each instruction set:

- *Branch instructions* on page F-466.
- *Load and store instructions* on page F-470.
- *Conditional instructions* on page F-480.
- *Flag setting instructions* on page F-481.
- *32-bit T32 instructions* on page F-482.

Note

The ETMv4 architecture supports the following instruction sets:

- Armv8:
  - In AArch64 state, A64.
  - In AArch32 state, A32 and T32.
- Armv7:
  - Arm and Thumb.

This specification defines a P0 instruction as any of the following:

- All branch instructions, as specified in this appendix.
- All load instructions as specified in this appendix, when TRCCONFIGR.INSTP0 selects load instructions to be P0 instructions.
- All store instructions as specified in this appendix, when TRCCONFIGR.INSTP0 selects store instructions to be P0 instructions.

When a P0 instruction is traced, it generates a P0 element. Other instructions do not directly generate a P0 element, although the only way to trace an instruction which is not a P0 instruction is for a P0 element to be generated for a subsequent P0 instruction or an exception and that P0 element can imply the execution of a non-P0 instruction.

F.1 Branch instructions

This section shows which instructions are categorized as branch instructions, for the following instruction sets:

- *A64 instruction set*.
- *A32 instruction set* on page F-467.
- *T32 instruction set* on page F-467.

F.1.1 A64 instruction set

The following tables show which instructions are categorized as branch instructions.

Table F-1 A64 instruction set, direct branches

Instruction	Description	Link?	Return from exception?
B	Unconditional branch	-	-
B.cond	Conditional branch	-	-
CBZ or CBNZ[a]	Compare with zero and branch	-	-
TBZ or TBNZ[a]	Test and branch	-	-
BL	Branch and link	Yes	-
ISB	Instruction Synchronization Barrier	-	-
WFI[b]	Wait For Interrupt	-	-
WFE[b]	Wait For Event	-	-

a. CBZ and CBNZ instructions, and TBZ and TBNZ instructions, are traced with an E *Atom element* if the branch is predicted as taken, or an N *Atom element* if the branch is predicted as not taken.

b. Whether WFI and WFE are classified as branch instructions depends on the value of TRCIDR2.WFXMODE. See *TRCIDR2, ID Register 2* on page 7-374.

Table F-2 A64 instruction set, indirect branches

Instruction	Description	Link?	Return from exception?
ERET	Return From Exception	-	Yes
ERETAA or ERETAB	Authenticate and exception return	-	Yes
RET	Return from subroutine	-	-
RETAA or RETAB	Authenticate and function return	-	-
BR	Branch to register	-	-
BRAA, BRAB, BRAAZ, or BRABZ	Authenticate and branch	-	-
BLR	Branch and link to register	Yes	-
BLRAA, BLRAB, BLRAAZ, or BLRABZ	Authenticate and branch with link	Yes	-

F.1.2 A32 instruction set

The A32 instruction set is the same as the Arm instruction set.

Table F-3 A32 instruction set, direct branches

Instruction	Description	Link?	Return from exception?[a]
`B`	Unconditional branch	-	-
`B<cc>`	Conditional branch	-	-
`BL`	Branch with Link	Yes	-
`BLX <immed>`	Branch with Link and Exchange	Yes	-
`ISB`	Instruction Synchronization Barrier, including CP15 encodings	-	-
`WFI`[b]	Wait For Interrupt	-	-
`WFE`[b]	Wait For Event	-	-

a. This column only applies to Armv7-A, Armv7-R, Armv8-A, and Armv8-R.
b. Whether WFI and WFE are classified as branch instructions depends on the value of TRCIDR2.WFXMODE. See *TRCIDR2, ID Register 2* on page 7-374.

Table F-4 A32 instruction set, indirect branches

Instruction	Description	Link?	Return from exception?[a]
`RFE`	Return From Exception	-	Yes
Data processing instructions that modify the PC	-	-	Yes[b]
`BX`	Branch and Exchange	-	-
`BLX <reg>`	Branch with Link and Exchange	Yes	-
`BXJ`	Branch and Exchange Jazelle	-	-
`LDR` or `LDRT` to the PC	Load a word to the PC	-	-
`LDM` including the PC	Load Multiple including to the PC	-	Yes[c]
`ERET`	Exception Return	-	Yes

a. This column only applies to Armv7-A, Armv7-R, Armv8-A, and Armv8-R.
b. Only those data processing instructions that modify the CPSR, such as `SUBS PC, LR`, or `MOVS PC, LR`.
c. Only those classified as `LDM` (exception return).

F.1.3 T32 instruction set

The T32 instruction set is the same as the Thumb instruction set.

The following tables show which instructions are categorized as branch instructions.

Table F-5 T32 instruction set, 32-bit instructions, direct branches

Instruction	Description	Link?	Return from exception?[a]
B	Unconditional branch	-	-
B<cc>	Conditional branch	-	-
BF[b]	Branch Future	-	-
BFCSEL[b]	Branch Future Conditional Select	-	-
BFL[bc]	Branch Future with Link	-	-
BFLX[bcd]	Branch Future with Link and Exchange	-	-
BFX[bd]	Branch Future with Exchange	-	-
BL	Branch with Link	Yes	-
BLX <immed>	Branch with Link and Exchange	Yes	-
ISB	Instruction Synchronization Barrier, including CP15 encodings	-	-
LE	Loop End	-	-
LETP	Loop End with Tail Predication	-	-
WFI[e]	Wait For Interrupt	-	-
WFE[e]	Wait For Event	-	-
WLS	While Loop Start	-	-
WLSTP	While Loop Start with Tail Predication	-	-

a. This column only applies to Armv7-A, Armv7-R, Armv8-A, and Armv8-R.

b. Whether branch future instructions are classified as branch instructions depends on the value of TRCIDR0.BF.

c. BFL and BFLX instructions are not branch with link instructions, however any resulting implicit branch caused by these instructions is a branch with link instruction and therefore might cause a return stack push. See *Return stack* on page 5-222.

d. BFX and BFLX instructions are direct branches, however any resulting implicit branch caused by these instructions is an indirect branch. See *Atom instruction trace element* on page 5-192.

e. Whether WFI and WFE are classified as branch instructions depends on the value of TRCIDR2.WFXMODE. See *TRCIDR2, ID Register 2* on page 7-374.

Table F-6 T32 instruction set, 32-bit instructions, indirect branches

Instruction	Description	Link?	Return from exception?[a]
RFE[b]	Return From Exception	-	Yes
BXJ[b]	Branch and Exchange Jazelle	-	-
LDR to the PC	Load a word to the PC	-	-
LDM including the PC	Load Multiple including to the PC	-	Yes[c]

Table F-6 T32 instruction set, 32-bit instructions, indirect branches (continued)

Instruction	Description	Link?	Return from exception?[a]
TBB or TBH	Table Branch	-	-
SUBS PC, LR[b]	Data processing instruction that modifies the PC	-	Yes[d]
ERET[b]	Exception Return	-	Yes

a. This column only applies to Armv7-A, Armv7-R, Armv8-A, and Armv8-R.
b. These instructions do not exist in Armv7-M and Armv8-M and are not branch instructions.
c. Only those classified as LDM (exception return).
d. Only those data processing instructions that modify the CPSR, such as SUBS PC, LR or MOVS PC, LR.

Table F-7 T32 instruction set, 16-bit instructions, direct branches

Instruction	Description	Link?	Return from exception?[a]
B	Unconditional branch	-	-
B<cc>	Conditional branch	-	-
CBZ or CBNZ[b]	Compare and Branch on Zero, or Nonzero	-	-
WFI[c]	Wait For Interrupt	-	-
WFE[c]	Wait For Event	-	-

a. This column only applies to Armv7-A, Armv7-R, Armv8-A, and Armv8-R.
b. CBZ and CBNZ instructions are traced with an E *Atom element* if the branch is predicted as taken, or an N *Atom element* if the branch is predicted as not taken.
c. Whether WFI and WFE are classified as branch instructions depends on the value of TRCIDR2.WFXMODE. See *TRCIDR2, ID Register 2* on page 7-374.

Table F-8 T32 instruction set, 16-bit instructions, indirect branches

Instruction	Description	Link?	Return from exception?[a]
ADD or MOV to the PC	Data processing instruction that modifies the PC	-	-
BX	Branch and Exchange	-	-
BLX <reg>	Branch with Link and Exchange	Yes	-
BLXNS	Branch with Link and Exchange Non-secure	Yes	-
BXNS	Branch and Exchange Non-secure	-	-
POP including the PC	Pop from the stack including the PC	-	-

a. This column only applies to Armv7-A, Armv7-R, Armv8-A, and Armv8-R.

F.2 Load and store instructions

This section shows which instructions are categorized as load and store instructions, for the following instruction sets:

- Armv7, Armv8-R, and Armv8-M:
  - A32 and T32. See *Armv7, Armv8-R, and Armv8-M A32 and T32 instruction sets*.

Note

ETMv4 does not support data tracing on Armv7-A and Armv8-A.

This section also describes the meanings of the transfer indexes that are contained in P1 elements:

- *P1 element transfer index meanings* on page F-477.

F.2.1 Armv7, Armv8-R, and Armv8-M A32 and T32 instruction sets

Table F-9 lists the instructions that are categorized as load and store instructions.

Table F-9 Armv7 A32 and T32 instruction sets, load and store instructions

Instruction	Description	Transfer index scheme[a]
Single-transfer instructions		
LDR{T}	Load word	Default
LDR{B|H}{T}	Load byte or halfword	Default
LDRS{B|H}{T}	Load signed byte or halfword	Default
LDA	Load acquire word	Default
LDA{B|H}	Load acquire byte or halfword	Default
LDREX	Load exclusive word	Default
LDREX{B|H}	Load exclusive byte or halfword	Default
LDAEX	Load acquire exclusive word	Default
LDAEX{B|H}	Load acquire exclusive byte or halfword	Default
STR{T}	Store word	Default
STR{B|H}{T}	Store byte or halfword	Default
STL	Store release word	Default
STL{B|H}	Store release byte or halfword	Default
TB{B|H}	Table branch byte or halfword	Default
VLDR.32	Vector load, 32-bit option	Default
VLLDM	Floating-point Lazy Load Multiple	See Table F-18 on page F-479[b]
VLSTM	Floating-point Lazy Store Multiple	See Table F-18 on page F-479[b]
VSTR.32	Vector store, 32-bit option	Default
Multiple-transfer instructions		

Table F-9 Armv7 A32 and T32 instruction sets, load and store instructions (continued)

Instruction	Description	Transfer index scheme[a]
BLXNS	Branch with Link and Exchange Non-secure	Address order 32
LDC/LDC2	Load Coprocessor	Address order 32
LDM	Load Multiple	Address order 32
LDRD	Load Register Dual	Address order 32
LDREXD	Load Register Exclusive Dual	Address order 32
LDAEXD	Load acquire exclusive dual	Address order 32
RFE	Return From Exception	Address order 32
PUSH	Push multiple registers	Address order 32
POP	Pop multiple registers	Address order 32
SRS	Store Return State	Address order 32
STC/STC2	Store Coprocessor	Address order 32
STM	Store Multiple	Address order 32
STRD	Store Register Dual	Address order 32
STREX	Store-exclusive word	Exclusive single 32
STREX{B|H}	Store exclusive byte or halfword	Exclusive single 32
STREXD	Store exclusive dual	Exclusive multiple 32
STLEX	Store release exclusive word	Exclusive single 32
STLEX{B|H}	Store release exclusive byte or halfword	Exclusive single 32
STLEXD	Store release exclusive dual	Exclusive multiple 32
SWP	Swap a word	Swap
SWPB	Swap a byte	Swap
VLDM	Load extension register multiple	Address order 32
VLD<n>	Load extension register multiple	Address order 32
VLDR.64	Load extension register, 64-bit option	Address order 32
VPOP	Load extension register multiple	Address order 32
VSTM	Store extension register multiple	Address order 32
VST<n>	Store extension register multiple	Address order 32
VSTR.64	Store extension register, 64-bit option	Address order 32
VPUSH	Store extension register multiple	Address order 32

a. See *P1 element transfer index meanings* on page F-477.

b. VLLDM and VLSTM instructions are only considered to be load or store operations on an Armv8-M PE that includes the Main extension. On an Armv8-M PE that includes the Main extension, these instructions are always considered to be load or store operations, regardless of the current state of the PE, or whether the FP extension is implemented or not.

The SWP and SWPB instructions are treated as both load and store instructions. This means that the SWP and SWPB instructions are traced as P0 elements whenever TRCCONFIGR.INSTP0 is not 0b00.

Note

This section describes all the possible load and store instructions in the A32 and T32 instruction sets. In some PE architectures, not all of these instructions are required. For those PEs, these instruction encodings are not load and store instructions. For example, in Armv7-M and Armv8-M the RFE and SRS instructions are not included and are therefore not load and store instructions.

F.2.2 General-purpose register loads and stores

This section covers the loads and stores of general-purpose registers R0 to R15, including all variants of addressing modes, exclusives, and swaps.

The ETMv4 architecture states that the trace unit observes the register view of a general purpose register load or store.

This means that for the following sequence, the same value is observed by the trace unit for all four data transfers:

1. `SETEND LE`
2. `STR r0, [r8]`
3. `LDR r1, [r8]`
4. `SETEND BE`
5. `STR r0, [r8]`
6. `LDR r1,[r8]`

Unaligned transfers have no effect on the value observed by the ETM and it is always the register view.

Table F-10 General-purpose register stores

Instruction[a]	Endianness	Value in register	Value observed by ETM
STR (word)	Little/Big	0x04030201	Transfer 0: 0x04030201
STRH	Little/Big	0x04030201	Transfer 0: 0x0201
STRB	Little/Big	0x04030201	Transfer 0: 0x01

a. Includes exclusive variants and SWP/SWPB.

For store-multiple instructions such as STM and STRD, these are treated as a sequence of operations from the set in Table F-10. For load-multiple instructions, such as LDM and LDRD, these are treated as a sequence of operations from the set in Table F-11.

Table F-11 General-purpose register loads

Instruction[a]	Endianness	Value loaded into register	Value observed by ETM
LDR (word)	Little/Big	0x04030201	Transfer 0: 0x04030201
LDRH	Little/Big	0x0201	Transfer 0: 0x0201
LDRB	Little/Big	0x01	Transfer 0: 0x01

a. Includes exclusive variants and SWP/SWPB.

For load-multiple instructions such as LDM and LDRD, these are treated as a sequence of operations from the set in Table F-11.

F.2.3 Extension register loads and stores

This section covers Vector load and stores, excluding multiple-element load and stores which are described in Table F-12.

For loads and stores of single-precision registers, the trace unit observes the register view. This means that for the following sequence, the same value is observed by the trace unit for all four data transfers:

1. `SETEND LE`
2. `VSTR s0, [r8]`
3. `VLDR s1, [r8]`
4. `SETEND BE`
5. `VSTR s0, [r8]`
6. `VLDR s1, [r8]`

For loads and stores of double-precision registers, doubleword transfers are observed as two sequential word-sized transfers, where each word is one half of the register view of the doubleword.

In Little Endian mode, the lowest addressed transfer is the least significant half of the doubleword and the next addressed transfer is the most significant half of the doubleword. In Big Endian mode, the lowest addressed transfer is the most significant half of the doubleword and the next addressed transfer is the least significant half of the doubleword.

For the following sequence, the same data values are observed by the trace unit for both data instructions:

1. `VSTR d0, [r8]`
2. `VLDR d1, [r8]`

Unlike for general-purpose register loads and stores however, in Big Endian mode the values that are observed are not the same as for Little Endian mode. This is because the order of the words for each instruction is reversed in Big Endian mode when compared with Little Endian mode.

Table F-12 Extension register stores

Instruction	Endianness	Value in register	Value observed by ETM
`VSTR` (word)	Little/Big	`0x04030201`	Transfer 0: `0x04030201`
`VSTR` (doubleword)	Little	`0x0807060504030201`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VSTR` (doubleword)	Big	`0x0807060504030201`	Transfer 0: `0x08070605` Transfer 1: `0x04030201`

`VSTM` is treated as a sequence of operations from the set in Table F-12.

Table F-13 Extension register loads

Instruction	Endianness	Value in register	Value observed by ETM
`VLDR` (word)	Little/Big	`0x04030201`	Transfer 0: `0x04030201`
`VLDR` (doubleword)	Little	`0x0807060504030201`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VLDR` (doubleword)	Big	`0x0807060504030201`	Transfer 0: `0x08070605` Transfer 1: `0x04030201`

`VLDM` is treated as a sequence of operations from the set in Table F-13.

F.2.4 Multiple element loads and stores

This section describes multiple element loads and stores. Instructions which load or store single elements are described in *Single element structure loads and stores to and from a single lane* on page F-476, and instructions which load a single element to multiple lanes are described in *Single element structure loads to multiple lanes* on page F-476.

Multiple element load and store instructions load or store a contiguous block of memory that is constructed from various portions of one or more extension registers. To align with most common implementations, the ETM observes the data transfers as a sequence of words to memory, each of which can contain one or more elements, and can also contain a partial element if that element crosses a word boundary. Within each element, the bytes are observed with the register view of that element, regardless of endianness.

For loads and stores of 64-bit elements, the 64-bit element transfers are observed as two sequential word-sized transfers, where each word is one half of the register view of the element. In Little Endian mode, the lowest addressed transfer is the least significant half of the element and the next addressed transfer is the most significant half of the element. In Big Endian mode, the lowest addressed transfer is the most significant half of the element and the next addressed transfer is the least significant half of the element.

Note

`VST1.64` is the same as `VSTR` (doubleword) and `VLD1.64` is the same as `VLDR` (doubleword).

Table F-14 Multiple element loads and stores

Instruction	Endianness	Value in register[a]	Values observed by ETM
`VST1.8 {d0}` `VLD1.8 {d0}`	Little/Big	`0x0807060504030201`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VST1.16 {d0}` `VLD1.16 {d0}`	Little/Big	`0x0807060504030201`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VST1.32 {d0}` `VLD1.32 {d0}`	Little/Big	`0x0807060504030201`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VST1.64 {d0}` `VLD1.64 {d0}`	Little	`0x0807060504030201`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VST1.64 {d0}` `VLD1.64 {d0}`	Big	`0x0807060504030201`	Transfer 0: `0x08070605` Transfer 1: `0x04030201`
`VST2.8 {dx,dy}` `VLD2.8 {dx,dy}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211`	Transfer 0: `0x12021101` Transfer 1: `0x14041303` Transfer 2: `0x16061505` Transfer 3: `0x18081707`
`VST2.16 {dx,dy}` `VLD2.16 {dx,dy}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211`	Transfer 0: `0x12110201` Transfer 1: `0x14130403` Transfer 2: `0x16150605` Transfer 3: `0x18170807`
`VST2.32 {dx,dy}` `VLD2.32 {dx,dy}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211`	Transfer 0: `0x04030201` Transfer 1: `0x14131211` Transfer 2: `0x08070605` Transfer 3: `0x18171615`

Table F-14 Multiple element loads and stores (continued)

Instruction	Endianness	Value in register[a]	Values observed by ETM
`VST3.8 {dx,dy,dz}` `VLD3.8 {dx,dy,dz}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211` dz: `0x2827262524232221`	Transfer 0: `0x02211101` Transfer 1: `0x13032212` Transfer 2: `0x24140423` Transfer 3: `0x06251505` Transfer 4: `0x17072616` Transfer 5: `0x28180827`
`VST3.16 {dx,dy,dz}` `VLD3.16 {dx,dy,dz}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211` dz: `0x2827262524232221`	Transfer 0: `0x12110201` Transfer 1: `0x04032221` Transfer 2: `0x24231413` Transfer 3: `0x16150605` Transfer 4: `0x08072625` Transfer 5: `0x28271817`
`VST3.32 {dx,dy,dz}` `VLD3.32 {dx,dy,dz}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211` dz: `0x2827262524232221`	Transfer 0: `0x04030201` Transfer 1: `0x14131211` Transfer 2: `0x24232221`
`VST4.8 {dx,dy,dz,dv}` `VLD4.8 {dx,dy,dz,dv}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211` dz: `0x2827262524232221` dv: `0x3837363534333231`	Transfer 0: `0x31211101` Transfer 1: `0x32221202` Transfer 2: `0x33231303` Transfer 3: `0x34241404` Transfer 4: `0x35251505` Transfer 5: `0x36261606` Transfer 6: `0x37271707` Transfer 7: `0x38281808`
`VST4.16 {dx,dy,dz,dv}` `VLD4.16 {dx,dy,dz,dv}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211` dz: `0x2827262524232221` dv: `0x3837363534333231`	Transfer 0: `0x12110201` Transfer 1: `0x32312221` Transfer 2: `0x14130403` Transfer 3: `0x34332423` Transfer 4: `0x16150605` Transfer 5: `0x36352625` Transfer 6: `0x18170807` Transfer 7: `0x38372827`
`VST4.32 {dx,dy,dz,dv}` `VLD4.32 {dx,dy,dz,dv}`	Little/Big	dx: `0x0807060504030201` dy: `0x1817161514131211` dz: `0x2827262524232221` dv: `0x3837363534333231`	Transfer 0: `0x04030201` Transfer 1: `0x14131211` Transfer 2: `0x24232221` Transfer 3: `0x34333231` Transfer 4: `0x08070605` Transfer 5: `0x18171615` Transfer 6: `0x28272625` Transfer 7: `0x38373635`

a. For `VST`, this is the value in the registers stored. For `VLD`, this is the value loaded into the register.

F.2.5 Single element structure loads and stores to and from a single lane

Table F-15 Single element structure

Instruction	Endianness	Value in register[a]	Values observed by ETM
`VST1.8 {d0}` `VLD1.8 {d0}`	Little/Big	d0: `0xXXXXXXXXXXXXXX01`	Transfer 0: `0x01`
`VST1.16 {d0}` `VLD1.16 {d0}`	Little/Big	d0: `0xXXXXXXXXXXXX0201`	Transfer 0: `0x0201`
`VST1.32 {d0}` `VLD1.32 {d0}`	Little/Big	d0: `0xXXXXXXXX04030201`	Transfer 0: `0x04030201`
`VST2.8 {d0,d1}` `VLD2.8 {d0,d1}`	Little/Big	d0: `0xXXXXXXXXXXXXXX01` d1: `0xXXXXXXXXXXXXXX02`	Transfer 0: `0x0201`
`VST2.16 {d0,d1}` `VLD2.16 {d0,d1}`	Little/Big	d0: `0xXXXXXXXXXXXX0201` d1: `0xXXXXXXXXXXXX0403`	Transfer 0: `0x04030201`
`VST2.32 {d0,d1}` `VLD2.32 {d0,d1}`	Little/Big	d0: `0xXXXXXXXX04030201` d1: `0xXXXXXXXX08070605`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VST3.8 {d0,d1,d2}` `VLD3.8 {d0,d1,d2}`	Little/Big	d0: `0xXXXXXXXXXXXXXX01` d1: `0xXXXXXXXXXXXXXX02` d2: `0xXXXXXXXXXXXXXX03`	Transfer 0: `0x030201`
`VST3.16 {d0,d1,d2}` `VLD3.16 {d0,d1,d2}`	Little/Big	d0: `0xXXXXXXXXXXXX0201` d1: `0xXXXXXXXXXXXX0403` d2: `0xXXXXXXXXXXXX0605`	Transfer 0: `0x04030201` Transfer 1: `0x0605`
`VST3.32 {d0,d1,d2}` `VLD3.32 {d0,d1,d2}`	Little/Big	d0: `0xXXXXXXXX04030201` d1: `0xXXXXXXXX08070605` d2: `0xXXXXXXXX14131211`	Transfer 0: `0x04030201` Transfer 1: `0x08070605` Transfer 2: `0x14131211`
`VST4.8 {d0,d1,d2,d3}` `VLD4.8 {d0,d1,d2,d3}`	Little/Big	d0: `0xXXXXXXXXXXXXXX01` d1: `0xXXXXXXXXXXXXXX02` d2: `0xXXXXXXXXXXXXXX03` d3: `0xXXXXXXXXXXXXXX04`	Transfer 0: `0x04030201`
`VST4.16 {d0,d1,d2,d3}` `VLD4.16 {d0,d1,d2,d3}`	Little/Big	d0: `0xXXXXXXXXXXXX0201` d1: `0xXXXXXXXXXXXX0403` d2: `0xXXXXXXXXXXXX0605` d3: `0xXXXXXXXXXXXX0807`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VST4.32 {d0,d1,d2,d3}` `VLD4.32 {d0,d1,d2,d3}`	Little/Big	d0: `0xXXXXXXXX04030201` d1: `0xXXXXXXXX08070605` d2: `0xXXXXXXXX14131211` d3: `0xXXXXXXXX18171615`	Transfer 0: `0x04030201` Transfer 1: `0x08070605` Transfer 2: `0x14131211` Transfer 3: `0x18171615`

a. For VST, this is the value in the registers stored. For VLD, this is the value loaded into the register. In all these examples the value is loaded to or stored from the least significant bytes of the register, although the instructions permit insertion into other positions in the register.

F.2.6 Single element structure loads to multiple lanes

The single element structure load to multiple lanes instructions loads a single element structure from memory and replicates that value across one or more registers.

The ETM observes the load of the single element structure, and does not observe the replicated elements.

Table F-16 Single element structure loads to multiple lanes

Instruction	Endianness	Value loaded into register	Values observed by ETM
`VLD1.8 {d0}`	Little/Big	d0: `0x0101010101010101`	Transfer 0: `0x01`
`VLD1.16 {d0}`	Little/Big	d0: `0x0201020102010201`	Transfer 0: `0x0201`
`VLD1.32 {d0}`	Little/Big	d0: `0x0403020104030201`	Transfer 0: `0x04030201`
`VLD2.8 {d0,d1}`	Little/Big	d0: `0x0101010101010101` d1: `0x0202020202020202`	Transfer 0: `0x0201`
`VLD2.16 {d0,d1}`	Little/Big	d0: `0x0201020102010201` d1: `0x0403040304030403`	Transfer 0: `0x04030201`
`VLD2.32 {d0,d1}`	Little/Big	d0: `0x0403020104030201` d1: `0x0807060508070605`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VLD3.8 {d0,d1,d2}`	Little/Big	d0: `0x0101010101010101` d1: `0x0202020202020202` d2: `0x0303030303030303`	Transfer 0: `0x030201`
`VLD3.16 {d0,d1,d2}`	Little/Big	d0: `0x0201020102010201` d1: `0x0403040304030403` d2: `0x0605060506050605`	Transfer 0: `0x04030201` Transfer 1: `0x0605`
`VLD3.32 {d0,d1,d2}`	Little/Big	d0: `0x0403020104030201` d1: `0x0807060508070605` d2: `0x1413121114131211`	Transfer 0: `0x04030201` Transfer 1: `0x08070605` Transfer 2: `0x14131211`
`VLD4.8 {d0,d1,d2,d3}`	Little/Big	d0: `0x0101010101010101` d1: `0x0202020202020202` d2: `0x0303030303030303` d3: `0x0404040404040404`	Transfer 0: `0x04030201`
`VLD4.16 {d0,d1,d2,d3}`	Little/Big	d0: `0x0201020102010201` d1: `0x0403040304030403` d2: `0x0605060506050605` d3: `0x0807080708070807`	Transfer 0: `0x04030201` Transfer 1: `0x08070605`
`VLD4.32 {d0,d1,d2,d3}`	Little/Big	d0: `0x0403020104030201` d1: `0x0807060508070605` d2: `0x1413121114131211` d3: `0x1817161518171615`	Transfer 0: `0x04030201` Transfer 1: `0x08070605` Transfer 2: `0x14131211` Transfer 3: `0x18171615`

F.2.7 P1 element transfer index meanings

Every P1 element contains a transfer index. The meaning of the transfer index depends on the *transfer index scheme* that the P1 element is using. The transfer index scheme that a P1 element uses depends on the instruction type that the P1 element is associated with.

The transfer index schemes, and the meaning of the transfer index for each scheme, are as follows:

Default The default scheme is used for any single transfer instruction. The transfer index is always 0.

Address order 32

This scheme applies to most instructions that perform multiple data transfers, where those transfers are always words. A new P1 element is generated for each data transfer performed, so that each P1 element indicates the address of a word sized data transfer.

The P1 elements might not be generated in the same order that the data transfers are performed. The transfer index that is contained in each P1 element indicates to a trace analyzer which data transfer each P1 element represents.

For example, `LDM r0, {r2,r5,r6}` from address `0x1000` results in three P1 elements, each with a different transfer index value:

- Transfer index 0 = address `0x1000`. Data value, represented by an associated P2 element, = `r2`.
- Transfer index 1 = address `0x1004`. Data value, represented by an associated P2 element, = `r5`.
- Transfer index 2 = address `0x1008`. Data value, represented by an associated P2 element, = `r6`.

Exclusive single 32

This scheme applies to single transfer store-exclusive instructions. There are always two P1 elements for these instructions, one for the data store and the other for the success indicator. These two P1 elements might be generated in any order. The transfer index indicates whether the P1 element is for the data store or for the success indicator.

For example, `STREX r3, r1, [r0]` to address `0x1000` results in the following P1 elements:

- Transfer index 0 = address `0x1000`. Data value, represented by an associated P2 element, = `r1`.
- Transfer index 1 = no address. The value that is represented in the associated P2 element is the value of the success indicator, as loaded into `r3`.

For more information, see *Data trace behavior on tracing store-exclusive instructions* on page 2-77.

Exclusive multiple 32

This scheme applies to multiple transfer store-exclusive instructions, such as `STREXD`. There are always `n+1` P1 elements for these instructions:

- There are `n` P1 elements for the data stores, with transfer index values from 0 to `n-1`.
- There is one final P1 element for the success indicator of the store-exclusive instruction, with index `n`.

The P1 elements might be generated in any order. The transfer index indicates whether a P1 element is for a data store, and if so indicates which data store, or is for the success indicator.

For example, `STREXD r3, r1, r2, [r0]` to address `0x1000` results in three P1 elements, each with a different transfer index value:

- Transfer index 0 = address `0x1000`. Data value, represented by an associated P2 element, = `r1`.
- Transfer index 1 = address `0x1004`. Data value, represented by an associated P2 element, = `r2`.
- Transfer index 2 = no address. The value that is represented in the associated P2 element is the value of the success indicator, as loaded into `r3`.

For more information, see *Data trace behavior on tracing store-exclusive instructions* on page 2-77.

Swap

This scheme applies to instructions that perform two accesses to the same address, where one access is for a data load and the other is for a data store. There are always two P1 elements for these instructions:

- Transfer index 0 is for the data load.
- Transfer index 1 is for the data store.

When using `VLDR`, `VLDM`, `VSTR, and VSTM` instructions to load or store doubleword registers in big-endian mode, the transfer indexes that are traced are different from little-endian mode. In all modes, each doubleword is traced using a pair of word transfers with adjacent transfer index values. In little-endian mode, the lower transfer index value is used with the least significant word of the doubleword, and the higher transfer index value is used with the most significant word of the doubleword. In big-endian mode, the lower transfer index value is used with the most significant word of the doubleword, and the higher transfer index value is used with the least significant word of the doubleword.

F.2.8 P1 transfer indexes for exceptions

For Armv6-M, Armv7-M, and Armv8-M PEs, when an exception is taken there is a set of stack push transfers, and when an exception return occurs there is a set of stack pop transfers. Table F-17 shows the transfer groups and transfer indexes for these transfers.

Table F-17 Exception stack push and pop data transfers

Group	Registers transferred	Transfer indexes for stack push	Transfer indexes for stack pop
State context	R0-R3	0-3	0-3
	R12	4	4
	LR	5	5
	ReturnAddress	6	6
	xPSR	7	7
FP context	S0-S15	8-23	8-23
	FPSCR	24	24
	Reserved word[a]	25	25
Additional FP context[b]	S16-S31	36-51	36-51
Additional state context[b]	Magic signature	26	26
	Reserved word[a]	27	27
	R4-R11	28-35	28-35

a. These Reserved words are optional and might or might not be traced. When traced, they are traced with the transfer index shown. If a P2 data value is traced for these words, the value is UNKNOWN.

b. Armv8-M only.

For Armv6-M, Armv7-M, and Armv8-M PEs that support the Lazy context save function:

- If the FP context is saved when an exception is taken, the transfers use the transfer indexes that are shown in Table F-17.
- If the FP context is not saved immediately when the exception is taken and is saved later using the Lazy context save operation, the FP context and additional FP context use the transfer indexes that are shown in Table F-18. These transfers are associated with the Lazy FP push exception which is traced in the instruction trace stream.

Table F-18 Lazy FP context save operation

Group	Registers transferred	Transfer indexes for stack push
FP context	S0-S15	0-15
	FPSCR	16
	Reserved word[a]	17
Additional FP context[b]	S16 - S31	18-33

a. The Reserved word is optional and might or might not be traced. When traced, it is traced with the transfer index shown. If a P2 data value is traced for this word, the value is UNKNOWN.

b. Armv8-M only.

F.3 Conditional instructions

This section shows which instructions are categorized as conditional instructions, for the following instruction sets:

- *A32 instruction set*.
- *T32 instruction set*.

F.3.1 A32 instruction set

A32 instruction set on page F-467 lists these instruction sets.

An A32 instruction is a conditional instruction if it is not categorized as a branch instruction and one of the following is true:

- The instruction has a valid 4-bit condition code that is not set to the AL or NV encoding.
- The instruction is a VSEL instruction with a 2-bit condition code.

F.3.2 T32 instruction set

A T32 instruction is a conditional instruction if it is not categorized as a branch instruction and one of the following is true:

- The instruction has a valid 4-bit condition code that is not set to the AL or NV encoding.
- The instruction is in an IT block, where the IT instruction does not use the AL condition code.
- The instruction is a VSEL instruction with a 2-bit condition code.
- The instruction is one of the following Set instructions:
  - `CSET`
  - `CSETM`
- The instruction is one of the following Select instructions:
  - `CSEL`
  - `CSINC`
  - `CSINV`
  - `CSNEG`

When TRCIDR0.CONDTYPE == 0b00, for the instructions `CSET`, `CSETM`, `CSEL`, `CSINC`, `CSINV` and `CSNEG`, the *Conditional Result element* indicates the instruction passed the condition code check when the `fcond` condition holds, and otherwise indicates the instruction failed the condition code check.

ETMv4 does not support reliable tracing of SG instructions when they are placed in an IT block, when conditional tracing is enabled by setting TRCCONFIGR.COND to a non-zero value. If an SG instruction is encountered in an IT block, Arm recommends that it is treated as an unconditional instruction.

F.4 Flag setting instructions

A flag setting instruction is any instruction that might update the condition flags in the *Application Program Status Register* (APSR).

An instruction that might update the condition flags is categorized as a flag setting instructions regardless of whether it updates the flags. This means that even if the instruction fails its own condition code check so that it does not update the flags, it is still categorized as a flag setting instruction.

The following sections show the instructions that are categorized as flag setting instructions, for the following instruction sets:

- *A32 instruction set*.
- *T32 instruction set*.

F.4.1 A32 instruction set

Any instruction that modifies the N, Z, C, or V flags in the APSR is categorized as a flag setting instruction. See the *Arm Architecture Reference Manual Armv7-A and Armv7-R edition* for more information.

Note

The condition flags are also updated whenever an exception is taken.

F.4.2 T32 instruction set

Any instruction that modifies the N, Z, C, or V flags in the APSR is categorized as a flag setting instruction. See the *Arm Architecture Reference Manual Armv7-A and Armv7-R edition*, the *Arm®v8-A Architecture Reference Manual* or the *Arm®v8-M Architecture Reference Manual* for more information.

Note

The condition flags are also updated whenever an exception is taken.

F.5 32-bit T32 instructions

All 32-bit T32 instructions are traced as single instructions. A PE must not take an exception between the two halfwords that constitute a 32-bit T32 instruction.

Appendix G
Standard Layout of the External Inputs

This appendix gives recommendations on the number and type of inputs that are available to a trace unit. It contains the following section:

- *Recommended connection layout on page G-484*.

G.1 Recommended connection layout

Arm recommends that the external inputs are connected in the following way:

[3:0] From cross-trigger interconnect, such as the CoreSight *Cross Trigger Interface* (CTI).

[n+3:4] n CPU performance events.

If the CTI provides fewer than 4 inputs, then the unimplemented external inputs are considered to be permanently zero, and the CPU performance events always start at external input 4.

Appendix H
Pseudocode Definition

This appendix provides a definition of the pseudocode used in this document, and lists the *helper* procedures and functions used by pseudocode to perform useful architecture-specific jobs. It contains the following sections:

H.1 About Arm pseudocode

Arm pseudocode provides precise descriptions of some areas of the architecture. The following sections describe the Armv7 pseudocode in detail:

- *Data types* on page H-487.
- *Expressions* on page H-491.
- *Operators and built-in functions* on page H-493.
- *Statements and program structure* on page H-498.

H.1.1 General limitations of Arm pseudocode

The pseudocode statements `IMPLEMENTATION_DEFINED`, `SEE`, `SUBARCHITECTURE_DEFINED`, `UNDEFINED`, and `UNPREDICTABLE` indicate behavior that differs from that indicated by the pseudocode being executed. If one of them is encountered:

- Earlier behavior indicated by the pseudocode is only specified as occurring to the extent required to determine that the statement is executed.
- No subsequent behavior indicated by the pseudocode occurs. This means that these statements terminate pseudocode execution.

For more information, see *Simple statements* on page H-498.

H.2 Data types

This section describes:

- *General data type rules*.
- *Bitstrings*.
- *Integers* on page H-488.
- *Reals* on page H-488.
- *Booleans* on page H-488.
- *Enumerations* on page H-488.
- *Lists* on page H-489.
- *Arrays* on page H-490.

H.2.1 General data type rules

Arm architecture pseudocode is a strongly-typed language. Every constant and variable is of one of the following types:

- Bitstring.
- Integer.
- Boolean.
- Real.
- Enumeration.
- List.
- Array.

The type of a constant is determined by its syntax. The type of a variable is normally determined by assignment to the variable, with the variable being implicitly declared to be of the same type as whatever is assigned to it. For example, the assignments `x = 1`, `y = '1'`, and `z = TRUE` implicitly declare the variables `x`, `y`, and `z` to have types integer, bitstring of length 1, and Boolean, respectively.

Variables can also have their types declared explicitly by preceding the variable name with the name of the type. This is most often done in function definitions for the arguments and the result of the function.

The remaining subsections describe each data type in more detail.

H.2.2 Bitstrings

A bitstring is a finite-length string of 0s and 1s. Each length of bitstring is a different type. The minimum permitted length of a bitstring is 1.

The type name for bitstrings of length `N` is `bits(N)`. A synonym of `bits(1)` is `bit`.

Bitstring constants are written as a single quotation mark, followed by the string of 0s and 1s, followed by another single quotation mark. For example, the two constants of type `bit` are `'0'` and `'1'`. Spaces can be included in bitstrings for clarity.

A special form of bitstring constant with `'x'` bits is permitted in bitstring comparisons, see *Equality and non-equality testing* on page H-493.

Every bitstring value has a left-to-right order, with the bits being numbered in standard *little-endian* order. That is, the leftmost bit of a bitstring of length *N* is bit (*N*–1) and its right-most bit is bit 0. This order is used as the most-significant-to-least-significant bit order in conversions to and from integers. For bitstring constants and bitstrings derived from encoding diagrams, this order matches the way they are printed.

Bitstrings are the only concrete data type in pseudocode, in the sense that they correspond directly to the contents of registers, memory locations, and instructions. All the remaining data types are abstract.

H.2.3 Integers

Pseudocode integers are unbounded in size and can be either positive or negative. That is, they are mathematical integers rather than what computer languages and architectures commonly call integers. Computer integers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as integers.

The type name for integers is `integer`.

Integer constants are normally written in decimal, such as `0`, `15`, `-1234`. They can also be written in C-style hexadecimal, such as `0x55` or `0x80000000`. Hexadecimal integer constants are treated as positive unless they have a preceding minus sign. For example, `0x80000000` is the integer $+2^{31}$. If -2^{31} needs to be written in hexadecimal, it must be written as `-0x80000000`.

H.2.4 Reals

Pseudocode reals are unbounded in size and precision. That is, they are mathematical real numbers, not computer floating-point numbers. Computer floating-point numbers are represented in pseudocode as bitstrings of the appropriate length, associated with suitable functions to interpret those bitstrings as reals.

The type name for reals is `real`.

Real constants are written in decimal with a decimal point. This means `0` is an integer constant but `0.0` is a real constant.

H.2.5 Booleans

A Boolean is a logical true or false value.

The type name for Booleans is `boolean`. This is not the same type as `bit`, which is a length–1 bitstring. Boolean constants are `TRUE` and `FALSE`.

H.2.6 Enumerations

An enumeration is a defined set of symbolic constants, such as:

```
enumeration InstrSet {InstrSet_A32, InstrSet_T32, InstrSet_A64};
```

An enumeration always contains at least one symbolic constant, and a symbolic constant must not be shared between enumerations.

Enumerations must be declared explicitly, although a variable of an enumeration type can be declared implicitly by assigning one of the symbolic constants to it. By convention, each of the symbolic constants starts with the name of the enumeration followed by an underscore. The name of the enumeration is its *type name*, or *type*, and the symbolic constants are its possible *constants*.

Note

A Boolean is a pre-declared enumeration that does not follow the normal naming convention and it has a special role in some pseudocode constructs, such as `if` statements, for example:

```
enumeration boolean {FALSE, TRUE};
```

H.2.7 Lists

A list is an ordered set of other data items, separated by commas and enclosed in parentheses, for example:

```
(bits(32) shifter_result, bit shifter_carry_out)
```

A list always contains at least one data item.

Lists are often used as the return type for a function that returns multiple results. For example, this list at the start of this section is the return type of the function `Shift_C()` that performs a standard Arm shift or rotation, when its first operand is of type `bits(32)`.

Some specific pseudocode operators use lists surrounded by other forms of bracketing than the `(…)` parentheses. These are:

- Bitstring extraction operators, that use lists of bit numbers or ranges of bit numbers surrounded by angle brackets `<…>`.
- Array indexing, that uses lists of array indexes surrounded by square brackets `[…]`.
- Array-like function argument passing, that uses lists of function arguments surrounded by square brackets `[…]`.

Each combination of data types in a list is a separate type, with type name given by listing the data types. This means that the example list at the start of this section is of type `(bits(32), bit)`. The general principle that types can be declared by assignment extends to the types of the individual list items in a list. For example:

```
(shift_t, shift_n) = ('00', 0);
```

implicitly declares `shift_t`, `shift_n`, and `(shift_t, shift_n)` to be of types `bits(2)`, `integer`, and `(bits(2), integer)`, respectively.

A list type can also be explicitly named, with explicitly named elements in the list. For example:

```
type ShiftSpec is (bits(2) shift, integer amount);
```

After this definition and the declaration:

```
ShiftSpec abc;
```

the elements of the resulting list can then be referred to as `abc.shift`, and `abc.amount`. This qualified naming of list elements is only permitted for variables that have been explicitly declared, not for those that have been declared by assignment only.

Explicitly naming a type does not alter what type it is. For example, after the above definition of `ShiftSpec`, `ShiftSpec`, and `(bits(2), integer)` are two different names for the same type, not the names of two different types. To avoid ambiguity in references to list elements, it is an error to declare a list variable multiple times using different names of its type or to qualify it with list element names not associated with the name by which it was declared.

An item in a list that is being assigned to can be written as `"-"` to indicate that the corresponding item of the assigned list value is discarded. For example:

```
(shifted, -) = LSL_C(operand, amount);
```

List constants are written as a list of constants of the appropriate types, for example the `('00', 0)` in the earlier example.

H.2.8 Arrays

Pseudocode arrays are indexed by either enumerations or integer ranges. An integer range is represented by the lower inclusive end of the range, then `..` followed by the upper inclusive end of the range.

For example:

```
// The names of the Banked core registers.

enumeration RName {RName_0usr, RName_1usr, RName_2usr, RName_3usr, RName_4usr, RName_5usr,
                   RName_6usr, RName_7usr, RName_8usr, RName_8fiq, RName_9usr, RName_9fiq,
                   RName_10usr, RName_10fiq, RName_11usr, RName_11fiq, RName_12usr, RName_12fiq,
                   RName_SPusr, RName_SPfiq, RName_SPirq, RName_SPsvc,
                   RName_SPabt, RName_SPund, RName_SPmon, RName_SPhyp,
                   RName_LRusr, RName_LRfiq, RName_LRirq, RName_LRsvc,
                   RName_LRabt, RName_LRund, RName_LRmon,
                   RName_PC};

array bits(8) _Memory[0..0xFFFFFFFF];
```

Arrays are always explicitly declared, and there is no notation for a constant array. Arrays always contain at least one element, because:

- Enumerations always contain at least one symbolic constant.
- Integer ranges always contain at least one integer.

Arrays do not usually appear directly in pseudocode. The items that syntactically look like arrays in pseudocode are usually array-like functions such as `R[i]`, `MemU[address, size]` or `Elem[vector, i, size]`. These functions package up and abstract additional operations normally performed on accesses to the underlying arrays, such as register banking, memory protection, endian-dependent byte ordering, exclusive-access housekeeping and Advanced SIMD element processing.

H.3 Expressions

This section describes:

- *General expression syntax*.
- *Operators and functions - polymorphism and prototypes* on page H-492.
- *Precedence rules* on page H-492.

H.3.1 General expression syntax

An expression is one of the following:

- A constant.
- A variable, optionally preceded by a data type name to declare its type.
- The word `UNKNOWN` preceded by a data type name to declare its type.
- The result of applying a language-defined operator to other expressions.
- The result of applying a function to other expressions.

Variable names normally consist of alphanumeric and underscore characters, starting with an alphabetic or underscore character.

Each register described in the text is to be regarded as declaring a correspondingly named bitstring variable, and that variable has the stated behavior of the register. For example, if a bit of a register is defined as RAZ/WI, then the corresponding bit of its variable reads as 0 and ignore writes.

An expression like `bits(32) UNKNOWN` indicates that the result of the expression is a value of the given type, but the architecture does not specify what value it is and software must not rely on such values. The value produced must not constitute a security hole and must not be promoted as providing any useful information to software.

Note

Some earlier documentation describes this as an UNPREDICTABLE value. UNKNOWN values are similar to the definition of UNPREDICTABLE, but do not indicate that the entire architectural state becomes unspecified.

Only the following expressions are assignable. This means that these are the only expressions that can be placed on the left-hand side of an assignment.

- Variables.
- The results of applying some operators to other expressions.

  The description of each language-defined operator that can generate an assignable expression specifies the circumstances under which it does so. For example, those circumstances might require that one or more of the expressions the operator operates is an assignable expression.
- The results of applying array-like functions to other expressions. The description of an array-like function specifies the circumstances under which it can generate an assignable expression.

Every expression has a data type:

- For a constant, this data type is determined by the syntax of the constant.
- For a variable, there are the following possible sources for the data type:
  - An optional preceding data type name.
  - A data type the variable was given earlier in the pseudocode by recursive application of this rule.
  - A data type the variable is being given by assignment, either by direct assignment to the variable, or by assignment to a list of which the variable is a member).

  It is a pseudocode error if none of these data type sources exists for a variable, or if more than one of them exists and they do not agree about the type.
- For a language-defined operator, the definition of the operator determines the data type.
- For a function, the definition of the function determines the data type.

H.3.2 Operators and functions - polymorphism and prototypes

Operators and functions in pseudocode can be polymorphic, producing different functionality when applied to different data types. Each resulting form of an operator or function has a different prototype definition. For example, the operator + has forms that act on various combinations of integers, reals, and bitstrings.

One particularly common form of polymorphism is between bitstrings of different lengths. This is represented by using `bits(N)`, `bits(M)`, or similar, in the prototype definition.

H.3.3 Precedence rules

The precedence rules for expressions are:

1. Constants, variables, and function invocations are evaluated with higher priority than any operators using their results.

2. Expressions on integers follow the normal operator precedence rules of *exponentiation before multiply/divide before add/subtract*, with sequences of multiply/divides or add/subtracts evaluated left-to-right.

3. Other expressions must be parenthesized to indicate operator precedence if ambiguity is possible, but need not be if all permitted precedence orders under the type rules necessarily lead to the same result. For example, if `i`, `j`, and `k` are integer variables, `i > 0 && j > 0 && k > 0` is acceptable, but `i > 0 && j > 0 || k > 0` is not.

H.4 Operators and built-in functions

This section describes:

- *Operations on generic types*.
- *Operations on Booleans*.
- *Bitstring manipulation*.
- *Arithmetic* on page H-496.

H.4.1 Operations on generic types

The following operations are defined for all types.

Equality and non-equality testing

Any two values `x` and `y` of the same type can be tested for equality by the expression `x == y` and for non-equality by the expression `x != y`. In both cases, the result is of type `boolean`.

A special form of comparison is defined with a bitstring constant that includes `'x'` bits as well as `'0'` and `'1'` bits. The bits corresponding to the `'x'` bits are ignored in determining the result of the comparison. For example, if opcode is a 4-bit bitstring, `opcode == '1x0x'` is equivalent to `opcode<3> == '1' && opcode<1> == '0'`.

Note

This special form is permitted in the implied equality comparisons in `when` parts of `case … of …` structures.

Conditional selection

If `x` and `y` are two values of the same type and `t` is a value of type `boolean`, then `if t then x else y` is an expression of the same type as `x` and `y` that produces `x` if `t` is `TRUE` and `y` if `t` is `FALSE`.

H.4.2 Operations on Booleans

If `x` is a Boolean, then `!x` is its logical inverse.

If `x` and `y` are Booleans, then `x && y` is the result of ANDing them together. As in the C language, if `x` is `FALSE`, the result is determined to be `FALSE` without evaluating `y`.

If `x` and `y` are Booleans, then `x || y` is the result of ORing them together. As in the C language, if `x` is `TRUE`, the result is determined to be `TRUE` without evaluating `y`.

If `x` and `y` are Booleans, then `x ^ y` is the result of exclusive-ORing them together.

H.4.3 Bitstring manipulation

The following bitstring manipulation functions are defined:

Bitstring length and most significant bit

If `x` is a bitstring:

- The bitstring length function `Len(x)` returns the length of `x` as an integer.
- `TopBit(x)` is the leftmost bit of `x`. Using bitstring extraction, this means:

  `TopBit(x)= x<Len(x)-1>`.

Bitstring concatenation and replication

If `x` and `y` are bitstrings of lengths `N` and `M` respectively, then `x:y` is the bitstring of length `N+M` constructed by concatenating `x` and `y` in left-to-right order.

If x is a bitstring and n is an integer with `n > 0`:

- `Replicate(x, n)` is the bitstring of length `n*Len(x)` consisting of n copies of x concatenated together
- `Zeros(n)` = `Replicate('0', n)`, `Ones(n)` = `Replicate('1', n)`.

Bitstring extraction

The bitstring extraction operator extracts a bitstring from either another bitstring or an integer. Its syntax is `x<integer_list>`, where x is the integer or bitstring being extracted from, and `<integer_list>` is a list of integers enclosed in angle brackets rather than the usual parentheses. The length of the resulting bitstring is equal to the number of integers in `<integer_list>`. In `x<integer_list>`, each of the integers in `<integer_list>` must be:

- `>= 0`
- `< Len(x)` if x is a bitstring.

The definition of `x<integer_list>` depends on whether `integer_list` contains more than one integer:

- If `integer_list` contains more than one integer, `x<i, j, k,…, n>` is defined to be the concatenation:

  `x<i> : x<j> : x<k> : … : x<n>`.

- If `integer_list` consists of just one integer `i`, `x<i>` is defined to be:
  - If x is a bitstring, `'0'` if bit `i` of x is a zero and `'1'` if bit `i` of x is a one.
  - If x is an integer, let y be the unique integer in the range `0` to `2^(i+1)-1` that is congruent to x modulo `2^(i+1)`. Then `x<i>` is `'0'` if `y < 2^i` and `'1'` if `y >= 2^i`.

    Loosely, this definition treats an integer as equivalent to a sufficiently long two's complement representation of it as a bitstring.

In `<integer_list>`, the notation `i:j` with `i >= j` is shorthand for the integers in order from `i` down to `j`, with both end values included. For example, `instr<31:28>` is shorthand for `instr<31, 30, 29, 28>`.

The expression `x<integer_list>` is assignable provided x is an assignable bitstring and no integer appears more than once in `<integer_list>`. In particular, `x<i>` is assignable if x is an assignable bitstring and `0 <= i < Len(x)`.

Encoding diagrams for registers frequently show named bits or multi-bit fields. For example, the encoding diagram for the TRCACATRn shows its bit<21> as DTBM. In such cases, the syntax `TRCACATR.DTBM` is used as a more readable synonym for `TRCACATR<21>`.

Logical operations on bitstrings

If x is a bitstring, `NOT(x)` is the bitstring of the same length obtained by logically inverting every bit of x.

If x and y are bitstrings of the same length, `x AND y`, `x OR y`, and `x EOR y` are the bitstrings of that same length obtained by logically ANDing, ORing, and exclusive-ORing corresponding bits of x and y together.

Bitstring count

If x is a bitstring, `BitCount(x)` produces an integer result equal to the number of bits of x that are ones.

Testing a bitstring for being all zero or all ones

If x is a bitstring:

- `IsZero(x)` produces `TRUE` if all of the bits of x are zeros and `FALSE` if any of them are ones.
- `IsZeroBit(x)` produces `'1'` if all of the bits of x are zeros and `'0'` if any of them are ones.

`IsOnes(x)` and `IsOnesBit(x)` work in the corresponding ways. This means:

```
IsZero(x)    = (BitCount(x) == 0)
IsOnes(x)    = (BitCount(x) == Len(x))
IsZeroBit(x) = if IsZero(x) then '1' else '0'
IsOnesBit(x) = if IsOnes(x) then '1' else '0'
```

Lowest and highest set bits of a bitstring

If x is a bitstring, and `N = Len(x)`:

- `LowestSetBit(x)` is the minimum bit number of any of its bits that are ones. If all of its bits are zeros, `LowestSetBit(x) = N`.
- `HighestSetBit(x)` is the maximum bit number of any of its bits that are ones. If all of its bits are zeros, `HighestSetBit(x) = -1`.
- `CountLeadingZeroBits(x)` is the number of zero bits at the left end of x, in the range 0 to N. This means:

  `CountLeadingZeroBits(x) = N - 1 - HighestSetBit(x)`.
- `CountLeadingSignBits(x)` is the number of copies of the sign bit of x at the left end of x, excluding the sign bit itself, and is in the range 0 to N–1. This means:

  `CountLeadingSignBits(x) = CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>)`.

Zero-extension and sign-extension of bitstrings

If x is a bitstring and i is an integer, then `ZeroExtend(x, i)` is x extended to a length of i bits, by adding sufficient zero bits to its left. That is, if `i == Len(x)`, then `ZeroExtend(x, i) = x`, and if `i > Len(x)`, then:

```
ZeroExtend(x, i) = Replicate('0', i-Len(x)) : x
```

If x is a bitstring and i is an integer, then `SignExtend(x, i)` is x extended to a length of i bits, by adding sufficient copies of its leftmost bit to its left. That is, if `i == Len(x)`, then `SignExtend(x, i) = x`, and if `i > Len(x)`, then:

```
SignExtend(x, i) = Replicate(TopBit(x), i-Len(x)) : x
```

It is a pseudocode error to use either `ZeroExtend(x, i)` or `SignExtend(x, i)` in a context where it is possible that `i < Len(x)`.

Converting bitstrings to integers

If x is a bitstring, `SInt(x)` is the integer whose two's complement representation is x:

```
// SInt()
// ======

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

`UInt(x)` is the integer whose unsigned representation is x:

```
// UInt()
// ======
```

```
integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

`Int(x, unsigned)` returns either `SInt(x)` or `UInt(x)` depending on the value of its second argument:

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

H.4.4 Arithmetic

Most pseudocode arithmetic is performed on integer or real values, with operands being obtained by conversions from bitstrings and results converted back to bitstrings afterwards. As these data types are the unbounded mathematical types, no issues arise about overflow or similar errors.

Unary plus, minus, and absolute value

If `x` is an integer or real, then `+x` is `x` unchanged, `-x` is `x` with its sign reversed, and `Abs(x)` is the absolute value of `x`. All three are of the same type as `x`.

Addition and subtraction

If `x` and `y` are integers or reals, `x+y` and `x-y` are their sum and difference. Both are of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

Addition and subtraction are particularly common arithmetic operations in pseudocode, and so it is also convenient to have definitions of addition and subtraction acting directly on bitstring operands.

If `x` and `y` are bitstrings of the same length `N`, so that `N = Len(x) = Len(y)`, then `x+y` and `x-y` are the least significant `N` bits of the results of converting them to integers and adding or subtracting them. Signed and unsigned conversions produce the same result:

```
x+y = (SInt(x) + SInt(y))<N-1:0>
    = (UInt(x) + UInt(y))<N-1:0>
x-y = (SInt(x) - SInt(y))<N-1:0>
    = (UInt(x) - UInt(y))<N-1:0>
```

If `x` is a bitstring of length `N` and `y` is an integer, `x+y` and `x-y` are the bitstrings of length `N` defined by `x+y = x + y<N-1:0>` and `x-y = x - y<N-1:0>`. Similarly, if `x` is an integer and `y` is a bitstring of length `M`, `x+y` and `x-y` are the bitstrings of length `M` defined by `x+y = x<M-1:0> + y` and `x-y = x<M-1:0> - y`.

Comparisons

If `x` and `y` are integers or reals, then `x == y`, `x != y`, `x < y`, `x <= y`, `x > y`, and `x >= y` are equal, not equal, less than, less than or equal, greater than, and greater than or equal comparisons between them, producing Boolean results. In the case of `==` and `!=`, this extends the generic definition applying to any two values of the same type to also act between integers and reals.

Multiplication

If `x` and `y` are integers or reals, then `x * y` is the product of `x` and `y`. It is of type `integer` if `x` and `y` are both of type `integer`, and `real` otherwise.

Division and modulo

If `x` and `y` are integers or reals, then `x/y` is the result of dividing `x` by `y`, and is always of type `real`.

If x and y are integers, then `x DIV y` and `x MOD y` are defined by:

```
x DIV y = RoundDown(x/y)
x MOD y = x - y * (x DIV y)
```

It is a pseudocode error to use any of x/y, `x MOD y`, or `x DIV y` in any context where y can be zero.

Square root

If x is an integer or a real, `Sqrt(x)` is its square root, and is always of type `real`.

Rounding and aligning

If x is a real:

- `RoundDown(x)` produces the largest integer n such that `n <= x`.
- `RoundUp(x)` produces the smallest integer n such that `n >= x`.
- `RoundTowardsZero(x)` produces `RoundDown(x)` if `x > 0.0`, `0 if x == 0.0`, and `RoundUp(x) if x < 0.0`.

If x and y are both of type `integer`, `Align(x, y) = y * (x DIV y)` is of type `integer`.

If x is of type `bitstring` and y is of type `integer`, `Align(x, y) = (Align(UInt(x), y))<Len(x)-1:0>` is a `bitstring` of the same length as x.

It is a pseudocode error to use either form of `Align(x, y)` in any context where y can be 0. In practice, `Align(x, y)` is only used with y a constant power of two, and the bitstring form used with `y = 2^n` has the effect of producing its argument with its n low-order bits forced to zero.

Scaling

If n is an integer, `2^n` is the result of raising `2` to the power n and is of type `real`.

If x and n are of type `integer`, then:

- `x << n = RoundDown(x * 2^n)`
- `x >> n = RoundDown(x * 2^(-n))`.

Maximum and minimum

If x and y are integers or reals, then `Max(x, y)` and `Min(x, y)` are their maximum and minimum respectively. Both are of type `integer` if x and y are both of type `integer`, and `real` otherwise.

H.5 Statements and program structure

The following sections describe the control statements used in the pseudocode:

- *Simple statements*.
- *Compound statements* on page H-499.
- *Comments* on page H-502.

H.5.1 Simple statements

Each of the following simple statements must be terminated with a semicolon, as shown.

Assignments

An assignment statement takes the form:

```
<assignable_expression> = <expression>;
```

Procedure calls

A procedure call takes the form:

```
<procedure_name>(<arguments>);
```

Return statements

A procedure return takes the form:

```
return;
```

and a function return takes the form:

```
return <expression>;
```

where `<expression>` is of the type declared in the function prototype line.

UNDEFINED

This subsection describes the statement:

```
UNDEFINED;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that the Undefined Instruction exception is taken.

UNPREDICTABLE

This subsection describes the statement:

```
UNPREDICTABLE;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is UNPREDICTABLE.

SEE…

This subsection describes the statement:

```
SEE <reference>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is that nothing occurs as a result of the current pseudocode because some other piece of pseudocode defines the required behavior. The `<reference>` indicates where that other pseudocode can be found.

It usually refers to another instruction but can also refer to another encoding or note of the same instruction.

IMPLEMENTATION_DEFINED

This subsection describes the statement:

```
IMPLEMENTATION_DEFINED {<text>};
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is IMPLEMENTATION DEFINED. An optional `<text>` field can give more information.

SUBARCHITECTURE_DEFINED

This subsection describes the statement:

```
SUBARCHITECTURE_DEFINED <text>;
```

This statement indicates a special case that replaces the behavior defined by the current pseudocode, apart from behavior required to determine that the special case applies. The replacement behavior is SUBARCHITECTURE DEFINED. An optional `<text>` field can give more information.

H.5.2 Compound statements

Indentation normally indicates the structure in compound statements. The statements contained in structures such as `if … then … else …` or procedure and function definitions are indented more deeply than the statement itself, and their end is indicated by returning to the original indentation level or less.

Indentation is normally done by four spaces for each level.

if … then … else …

A multi-line `if … then … else …` structure takes the form:

```
if <boolean_expression> then
    <statement 1>
    <statement 2>
    …
    <statement n>
elsif <boolean_expression> then
    <statement a>
    <statement b>
    …
    <statement z>
else
    <statement A>
    <statement B>
    …
    <statement Z>
```

The block of lines consisting of `elsif` and its indented statements is optional, and multiple such blocks can be used.

The block of lines consisting of `else` and its indented statements is optional.

Abbreviated one-line forms can be used when there are only simple statements in the `then` part and in the `else` part, if it is present, such as:

```
if <boolean_expression> then <statement 1>
if <boolean_expression> then <statement 1> else <statement A>
if <boolean_expression> then <statement 1> <statement 2> else <statement A>
```

Note

In these forms, `<statement 1>`, `<statement 2>` and `<statement A>` must be terminated by semicolons. This and the fact that the `else` part is optional are differences from the `if … then … else …` expression.

repeat … until …

A `repeat … until …` structure takes the form:

```
repeat
    <statement 1>
    <statement 2>
    …
    <statement n>
until <boolean_expression>;
```

while … do

A `while … do` structure takes the form:

```
while <boolean_expression> do
    <statement 1>
    <statement 2>
    …
    <statement n>
```

for …

A `for …` structure takes the form:

```
for <assignable_expression> = <integer_expr1> to <integer_expr2>
    <statement 1>
    <statement 2>
    …
    <statement n>
```

case … of …

A `case … of …` structure takes the form:

```
case <expression> of
    when <constant values>
        <statement 1>
        <statement 2>
        …
        <statement n>
        … more "when" groups …
    otherwise
        <statement A>
        <statement B>
        …
        <statement Z>
```

In this structure, `<constant values>` consists of one or more constant values of the same type as `<expression>`, separated by commas. Abbreviated one line forms of `when` and `otherwise` parts can be used when they contain only simple statements.

If `<expression>` has a bitstring type, `<constant values>` can also include bitstring constants containing `'x'` bits. For details see *Equality and non-equality testing* on page H-493.

Procedure and function definitions

A procedure definition takes the form:

```
<procedure name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    …
    <statement n>
```

where `<argument prototypes>` consists of zero or more argument definitions, separated by commas. Each argument definition consists of a type name followed by the name of the argument.

Note

This first prototype line is not terminated by a semicolon. This helps to distinguish it from a procedure call.

A function definition is similar but also declares the return type of the function:

```
<return type> <function name>(<argument prototypes>)
    <statement 1>
    <statement 2>
    …
    <statement n>
```

An array-like function is similar but with square brackets:

```
<return type> <function name>[<argument prototypes>]
    <statement 1>
    <statement 2>
    …
    <statement n>
```

An array-like function also usually has an assignment prototype:

```
<function name>[<argument prototypes>] = <value prototypes>
    <statement 1>
    <statement 2>
    …
    <statement n>
```

H.5.3 Comments

Two styles of pseudocode comment exist:

- // starts a comment that is terminated by the end of the line.
- /* starts a comment that is terminated by */.

Appendix I
Architecture Revisions

This appendix describes the architectural revisions in this book.

Table I-1 Architectural Changes for ETMv4.1

Change	Location
Added new values for TRCDEVARCH.REVISION and TRCIDR1.TRCARCHMIN.	• *TRCDEVARCH, Device Architecture register* on page 7-365. • *TRCIDR1, ID Register 1* on page 7-373.
Added support for a Virtual context identifier up to 32 bits, for tracing a PE based on the Armv8.1-A architecture and the Virtualization Host Extensions.	*Virtual context identifier tracing* on page 2-81, and throughout the document.
Added tracing of CONTEXTIDR_EL2.	*Context ID tracing* on page 2-81, and throughout the document.
Added TRCCONFIGR.VMIDOPT.	*TRCCONFIGR, Trace Configuration Register* on page 7-361.
Added TRCIDR2.VMIDOPT.	*TRCIDR2, ID Register 2* on page 7-374.
Updated the size of the Virtual context identifier field in Address and Address with Context packets.	*Address and Context tracing packets* on page 6-285, and throughout the document.
Updated the behavior of the Virtual context identifier comparators is updated.	*Virtual context identifier comparators* on page 4-158, and throughout the document.

Table I-2 Architectural Changes for ETMv4.2

Change	Location
Updated VMID location for Armv8-R.	*Context ID tracing* on page 2-81.
Added a new exception type of SecureFault.	*Exception types for the Armv6-M, Armv7-M and Armv8-M architectures* on page 5-199.
Added clarification to footnote about Lazy FP push exception tracing and non-invasive debug.	*Exception types for the Armv6-M, Armv7-M and Armv8-M architectures* on page 5-199.
Added clarification about the change in the definition of a lockup address between Armv7-M and Armv8-M.	*Further information for tracing exceptions on Armv8-M* on page 5-205.
Added Function Return instruction trace element.	*Function Return instruction trace element* on page 5-214.
Added support for virtual addresses larger than 48 bits.	• *TRCACVRn, Address Comparator Value Registers, n=0-15* on page 7-348. • *Address instruction trace element* on page 5-209. • *Exception instruction trace element* on page 5-196.
Added TRCAUTHSTATUS.HNID and TRCAUTHSTATUS.HID to indicate whether a separate debug enable for EL2 is implemented, and whether debug at EL2 is permitted.	*TRCAUTHSTATUS, Authentication Status register* on page 7-349.
Added new values for TRCDEVARCH.REVISION and TRCIDR1.TRCARCHMIN.	• *TRCDEVARCH, Device Architecture register* on page 7-365. • *TRCIDR1, ID Register 1* on page 7-373.
Added clarification about distinguishing between Secure and Non-secure states for the Armv8-M Security Extension.	• *TRCIDR3, ID Register 3* on page 7-376. • *TRCVICTLR, ViewInst Main Control Register* on page 7-413. • *TRCACATRn, Address Comparator Access Type Registers, n=0-15* on page 7-345. • *TRCAUTHSTATUS, Authentication Status register* on page 7-349. • *Context instruction trace packet* on page 6-292. • *Address with Context instruction trace packets* on page 6-294.
Added two new instructions, BXNS and BLXNS.	• *T32 instruction set* on page F-467. • *Armv7, Armv8-R, and Armv8-M A32 and T32 instruction sets* on page F-470.

Table I-3 Architectural Changes for ETMv4.3

Change	Location
Added a permitted minor architecture revision for ETMv4.3.	• *TRCIDR1, ID Register 1* on page 7-373. • *TRCDEVARCH, Device Architecture register* on page 7-365.
Added new instructions as indirect branch instructions and included full 64-bit address tracing to support the new instructions that were added to Armv8.3-A for pointer authentication.	• *Branch instructions* on page F-466. • *Address instruction trace element* on page 5-209.
Introduced a new minimal resource implementation that has none of the following resources: • Data value comparators. • External inputs. • External input selectors. • Counters. • Sequencer states.	• *TRCCNTCTLRn, Counter Control Registers, n=0-3* on page 7-359. • *TRCIDR4, ID Register 4* on page 7-378. • *TRCIDR5, ID Register 5* on page 7-381. • *TRCEVENTCTL0R, Event Control 0 Register* on page 7-368. • *TRCEVENTCTL1R, Event Control 1 Register* on page 7-368. • *TRCSEQEVRn, Sequencer State Transition Control Registers, n=0-2* on page 7-400. • *TRCTSCTLR, Global Timestamp Control Register* on page 7-409. • *TRCVDCTLR, ViewData Main Control Register* on page 7-411. • *TRCVICTLR, ViewInst Main Control Register* on page 7-413.
Introduced the option for WFI and WFE instructions to be classified as branch instructions.	• *TRCIDR2, ID Register 2* on page 7-374. • *Atom instruction trace element* on page 5-192. • *Trace unit behavior on a PE low-power state* on page 3-105.
Added the Ignore packet to the instruction and data trace protocols.	• *Descriptions of instruction trace packets* on page 6-252. • *Descriptions of data trace packets* on page 6-307.

Table I-4 Architectural Changes for ETMv4.4

Change	Location
Added new values for TRCDEVARCH.REVISION and TRCIDR1.TRCARCHMIN.	• *TRCDEVARCH, Device Architecture register* on page 7-365. • *TRCIDR1, ID Register 1* on page 7-373.
Added rule to state that ETMv4.4 is required if the PE implements the Armv8.4-A architecture.	*Armv8-A architecture requirements* on page B-452.
Added the ability to trace and filter EL2 Exception Level in Secure state.	• *TRCACATRn, Address Comparator Access Type Registers, n=0-15* on page 7-345 • *TRCIDR3, ID Register 3* on page 7-376. • *TRCVICTLR, ViewInst Main Control Register* on page 7-413.
Constrained the configuration of trace unit if the PE implements Armv8.4-Trace.	*TRCIDR0, ID Register 0* on page 7-370.
Updated the Authentication Interface if the PE implements Armv8.4-Trace.	*TRCAUTHSTATUS, Authentication Status register* on page 7-349.

Table I-4 Architectural Changes for ETMv4.4 (continued)

Change	Location
Added rule about not sharing trace unit with multiple PEs if Armv8.4-Trace is implemented.	• *Sharing a trace unit between multiple PEs* on page 1-23. • *TRCIDR3, ID Register 3* on page 7-376.
Added note about behavior of trace unit when non-invasive debug is not permitted.	*Trace unit behavior when tracing is prohibited* on page 3-108.
Added new rule about tracing in a region where the context information is restricted if Armv8.4-Trace is implemented.	*Context instruction trace element* on page 5-211.
Added new rule about timestamp values if Armv8.4-Trace is implemented.	*Global timestamping* on page 6-259.

Table I-5 Architectural Changes for ETMv4.5

Change	Location
Added new rules to describe the new Unified Power Domain.	• *Trace unit power domains* on page 3-91. • *If a trace unit is implemented in a unified power domain* on page 3-103.
Added new rule to describe changes required for Armv8.5 extensions.	Table 5-3 on page 5-197.
Added new values for TRCIDR1.TRCARCHMIN and TRCDEVARCH.REVISION.	• *TRCIDR1, ID Register 1* on page 7-373. • *TRCDEVARCH, Device Architecture register* on page 7-365.
Added new rules to further define sequences of instructions.	• *About instruction trace P0 elements* on page 2-35. • *Behavior of the start/stop control during a trace run* on page 4-121.
Added new rules to further define P0 instructions.	Appendix F *Instruction Categories*.
The following new trace elements are added to the instruction trace stream: • Branch Future Flush element. • Resynchronization element.	• *Branch Future Flush element* on page 5-213. • *Resynchronization element* on page 5-213.
The following new packets are added to the instruction trace stream: • Branch Future Flush packet. • Resynchronization packet.	• *Branch Future Flush packet* on page 6-306. • *Packets associated with synchronization between the trace unit and a trace analyzer* on page 6-253.
Added new rules for trace unit resources.	• *Memory access resources* on page 4-147. • *The instruction-based filtering model* on page 4-118. • *Use of the return stack* on page 5-222.
The following trace elements are extended to accomodate the new architecture: • *Atom element*. • *Exception element*. • Address element.	• *Atom instruction trace element* on page 5-192. • *Exception instruction trace element* on page 5-196. • *Address instruction trace element* on page 5-209.

Table I-5 Architectural Changes for ETMv4.5 (continued)

Change	Location
The following trace packets are extended to accomodate the new architecture: • Exception packet. • Trace On packet. • Exception Return packet. • Function Return packet.	• *Exception instruction trace packet* on page 6-262. • *Trace On instruction trace packet* on page 6-257. • *Exception instruction trace packet* on page 6-262. • *Function Return packet* on page 6-261.
Registers have been updated to include new possible values to allow for changes in architecture.	• *TRCIDR0, ID Register 0* on page 7-370. • *TRCIDR1, ID Register 1* on page 7-373. • *TRCDEVARCH, Device Architecture register* on page 7-365.
Made new additions to instruction categorizations.	Appendix F *Instruction Categories*.

Appendix J
Revisions

This appendix lists the technical changes between releases of this specification.

Table J-1 Differences between issue A and issue B.a

Change	Location
Changed CPSR to APSR throughout the document.	Entire document.
Removed Exception without Execution elements from the instruction trace, and renamed Exception with Execution to Exception.	Entire document.
Added detail to describe how to synchronize the instruction and data trace streams.	*Synchronizing the instruction and data trace streams* on page 2-43.
Added detail about interrupt continuable instructions on Armv6-M and Armv7-M PEs.	• *The algorithm for tracing the APSR condition flag values* on page 2-73. • *Behavior of the start/stop control during a trace run* on page 4-121. • *Single-shot controls for address comparators* on page 4-159. • *Exception instruction trace element* on page 5-196.
Added section.	*Tracing of stack transfers on Armv6-M, Armv7-M, and Armv8-M PEs* on page 4-136.
Added effect of the use of tagged addresses in Armv8 on trace unit memory access resources.	• *Memory access resources* on page 4-147. • *TRCACVRn, Address Comparator Value Registers, n=0-15* on page 7-348.

Table J-1 Differences between issue A and issue B.a (continued)

Change	Location
Added restrictions for single-shot comparator controls, and information about how these restrictions apply to Armv6-M and Armv7-M.	*Single-shot controls for address comparators* on page 4-159.
Added the ability for an *Exception element* to indicate that there is a serious fault pending.	Table 5-1 on page 5-184.
Added Lockup to exception types for the Armv6-M and Armv7-M architectures.	*Architectural exceptions* on page 5-197.
Added section.	*Additional information for tracing exceptions on Armv6-M, Armv7-M, and Armv8-M* on page 5-200.
Clarified that for accesses to the PPB space on an Armv6-M or Armv7-M PE, the endianness traced is UNKNOWN. It must always be considered to be little-endian.	*Occasions when P1 elements are traced without the address or endianness of the data transfer* on page 5-229.
Added TRCSSPCICR register.	• Table 7-1 on page 7-336. • *TRCSSPCICRn, Single-shot Processing Element Comparator Input Control Register, n=0-7* on page 7-404.
Added TRCEXDATA field to TRCIDR0.	*TRCIDR0, ID Register 0* on page 7-370.
Added TRCVDCTLR.TRCEXDATA bit.	*TRCVDCTLR, ViewData Main Control Register* on page 7-411.

Table J-2 Differences between issue B.a and issue B.b

Change	Location
Changed the text *direct branch instructions* to *direct branch and ISB instructions*.	Entire document.
Added detail about low-power state.	*Trace unit behavior on a PE low-power state* on page 3-105.
Changed the word *inactive* to *active* in the first block of Figure 4-5.	*Behavior of the ViewInst start/stop control* on page 4-123.
Added detail to clarify the behavior of the trace unit when tracing becomes active for some exceptions.	*Forcing tracing of exceptions* on page 4-130.
Added detail about the PE comparator inputs that are used to perform comparison against data addresses or data values.	*Single-shot controls for address comparators* on page 4-159.
Added detail to describe the TBI field of Translation Control Registers in Armv8-A PEs.	*Address instruction trace element* on page 5-209.
Added text to clarify the operation of the trace unit return stack.	*Use of the return stack* on page 5-222.
Changed the Purpose field description for the packets Address with Context, Short Address, and Long Address.	*Instruction trace packet header encodings, in byte order* on page 6-246.
Removed the incorrect references to P0.	*Short Address instruction trace packets* on page 6-289.
Added text to describe retention states.	*Access permissions* on page 7-340 and *TRCPDSR[1:0] encodings* on page 7-393.
Changed the value of ATID to align with the definition of an ATB trigger in the AMBA4 ATB protocol specification.	*TRCEVENTCTL1R, Event Control 1 Register* on page 7-368.

Table J-2 Differences between issue B.a and issue B.b (continued)

Change	Location
Removed sentence as TRCPDSR is not affected by an external trace reset.	*TRCPDSR, PowerDown Status Register* on page 7-392.
Added text to the INSTPRIORITY, DSTALL, and ISTALL field descriptions.	*TRCSTALLCTLR, Stall Control Register* on page 7-405.
Added text to clarify the condition where TRCEXDATA is 1.	*TRCVDCTLR, ViewData Main Control Register* on page 7-411.
Changed the parameter to NUMSSCC to align with the field name in TRCIDR4	*Configuration parameters* on page C-455.
Changed the entry for the parameter SUPPDAC to read “Yes” and added entries to the table for missing parameters.	*Recommended configurations* on page C-455.
Added new footnote to Table E-6.	*T32 instruction set, 32-bit instructions, indirect branches* on page F-468.

Table J-3 Differences between issue B.b and issue C

Change	Location
Replaced all instances of VMID with *Virtual context identifier*, except in the following cases: • If VMID forms part of the register name or field. • If VMID is the name of a field in a packet, such as a Context packet. • If VMID references VTTBR.VMID.	Throughout the document.
Updated descriptions to use the new RES0H definition.	Throughout the document.
Changed *PE clock* and *PE cycle* to *processor clock* and *processor cycle*	Throughout the document.
Added extension to pseudocode to consider the case where an overflow occurs.	*Trace analyzer pseudocode* on page 2-46.
Added clarification for trace units with data trace.	*Synchronizing with the data trace stream* on page 2-68.
Added clarification about cycle counting.	*Cycle counting* on page 2-81
Added support for a Virtual context identifier up to 32 bits, for tracing a PE based on the Armv8.1-A architecture and the Virtualization Host Extensions.	*Virtual context identifier tracing* on page 2-81, and throughout the document.
Added tracing of CONTEXTIDR_EL2.	*Context ID tracing* on page 2-81, and throughout the document.
Corrected text to be consistent with rules in the register description for TRCBBCTLR, Branch Broadcast Control Register.	*Branch broadcasting* on page 2-83.

Table J-3 Differences between issue B.b and issue C (continued)

Change	Location
Added clarification to emphasize the fact that the Software Lock needs to be implemented to be locked.	• *Saving and restoring the trace unit state* on page 3-94. • *Behavior on an external trace reset* on page 3-99. • *If a trace unit is implemented in multiple power domains* on page 3-101. • *External debugger and memory-mapped access* on page 4-167. • *Behaviors when using memory-mapped access* on page 7-342. • *TRCLAR, Software Lock Access Register* on page 7-388. • *TRCPDSR, PowerDown Status Register* on page 7-392.
Added clarifications to indicate that the rule governs the architected external outputs.	• *Trace unit behavior when the trace unit is disabled* on page 3-100. • *Trace unit behavior on a PE low-power state* on page 3-105. • *Behavior when in a prohibited region* on page 3-109.
Relaxed the specification to make it UNKNOWN whether writes to certain trace registers are ignored when the trace unit is enabled or not idle.	*Trace unit behavior when the trace unit is enabled* on page 3-100.
Added clarification about output trace packets when a flush request is received.	*Trace unit behavior on a trace flush* on page 3-108.
Added relaxation about a PE leaving prohibited region.	*Behavior when in a prohibited region* on page 3-109.
Added a section.	*Trace Unit behavior on changes in Exception level or Security state* on page 3-110.
Corrected the value of Counter 1 CNTEVENT in Figure 4-14. The value is LOW for the duration of the diagram.	Figure 4-14 on page 4-141.
Added text to clarify information about provision of a reduced function counter.	*Provision of a reduced function counter* on page 4-143.
Added relaxation to the Context ID comparators.	*Context ID comparators* on page 4-157.
Updated the behavior of the Virtual context identifier comparators.	*Virtual context identifier comparators* on page 4-158, and throughout the document.
Added detail about programming the trace unit.	*Use of the trace unit main enable bit* on page 4-165.
Updated text to clarify that the trace unit registers must only be programmed when the trace unit is disabled or not idle, and besides the exceptions to the rule, when the trace unit is enabled and TRCSTATR.IDLE indicates that the unit is not idle, writes to all other trace unit trace registers might be ignored.	*Use of the trace unit main enable bit* on page 4-165, and throughout the document.
Added note about setting claim tags.	*Synchronization when using system instructions to program the trace unit* on page 4-169.
Corrected value for TRCRSCTLRn, n=2.	Table 4-17 on page 4-182.
Added relaxation to the Virtual context identifier comparators.	*Virtual context identifier comparators* on page 4-158.
Added text to clarify the generation of Trace On elements around PE reset exceptions.	*Trace On instruction trace element* on page 5-190.

Table J-3 Differences between issue B.b and issue C (continued)

Change	Location
Updated Table 5-3, Exception types for the Armv7-A, Armv7-R, and Armv8 architectures.	Table 5-3 on page 5-197.
Added footnote to Table 5-4 and changed wording in the right column. Updated *fault* to read *exception*.	Table 5-4 on page 5-198.
Added detail about tracing of data transfers around PE reset exceptions.	*Additional information for tracing exceptions on Armv6-M, Armv7-M, and Armv8-M* on page 5-200.
Added clarification for tracing exceptions on Armv6-M and Armv7-M PEs.	*Additional information for tracing exceptions on Armv6-M, Armv7-M, and Armv8-M* on page 5-200.
Added note to clarify the generation of Address elements after Exceptions.	*Address instruction trace element* on page 5-209.
Replaced text and relaxed requirement.	*Context instruction trace element* on page 5-211.
Replaced text about generation of Commit element and about requests to insert a Cycle Count element.	*Cycle Count instruction trace element* on page 5-216.
Changed interpretation of Mispredict element.	*Mispredict instruction trace element* on page 5-219.
Added clarification for tracing P1 elements when TRCCONFIGR.DA is `0b0.`	*Occasions when P1 elements are traced without the address or endianness of the data transfer* on page 5-229.
Added clarification for the value of the timestamp.	*Timestamp data trace element* on page 5-230.
Removed note.	*Address instruction trace element* on page 5-209.
Updated text to clarify when Context elements are required or optional.	*Context instruction trace element* on page 5-211.
Updated text to clarify that some out of order cores mispredict non-conditional branches.	*Packet types* on page 6-234.
Updated the size of the Virtual context identifier field in Address and Address with Context packets.	*Address and Context tracing packets* on page 6-285, and throughout the document.
Added clarification for tracing of P1 packets when TRCCONFIGR.DA is `0b0.`	*Compression techniques used when generating Address packets* on page 6-286.
Added footnote to the entry for Reserved registers.	Table 7-6 on page 7-344.
Updated field bit descriptions of TRCAUTHSTATUS.	*TRCAUTHSTATUS, Authentication Status register* on page 7-349.
Added TRCCONFIGR.VMIDOPT.	*TRCCONFIGR, Trace Configuration Register* on page 7-361.
Added TRCIDR2.VMIDOPT.	*TRCIDR2, ID Register 2* on page 7-374.
Added new values for TRCDEVARCH.REVISION and TRCIDR1.TRCARCHMIN.	• *TRCDEVARCH, Device Architecture register* on page 7-365. • *TRCIDR1, ID Register 1* on page 7-373.
Added relaxation to field description of ATB trigger enable bit of TRCEVENTCTL1R.	*TRCEVENTCTL1R, Event Control 1 Register* on page 7-368.
Added text to clarify that TRCIDR0.COMMOPT is RES0 when TRCIDR0.TRCCCI is zero and cycle counting is not implemented.	*TRCIDR0, ID Register 0* on page 7-370.

Table J-3 Differences between issue B.b and issue C (continued)

Change	Location
Updated field description of TRCIDR0.TRCBB to be consistent with the rules described in TRCBBCTLR.	*TRCIDR0, ID Register 0* on page 7-370.
Updated field description of TRCIDR2.CCSIZE to state that this field is relaxed to RES0 if cycle counting is not implemented, as indicated by TRCIDR0.TRCCCI.	*TRCIDR2, ID Register 2* on page 7-374.
Updated field description of TRCIDR3.CCITMIN to state that this field is relaxed to RES0 when cycle counting in the instruction trace is not supported.	*TRCIDR3, ID Register 3* on page 7-376.
Added text to state that Arm deprecates the implementation of the Software Lock.	• *TRCLAR, Software Lock Access Register* on page 7-388. • *TRCLSR, Software Lock Status Register* on page 7-389. • Table C-1 on page C-455.
Added text to the usage constraints of TRCQCTLR, Q Element Control Register.	*TRCQCTLR, Q Element Control Register* on page 7-398.
Added new text to the DSTALL and ISTALL fields to clarify stalling in a multi-threaded processor.	*TRCSTALLCTLR, Stall Control Register* on page 7-405.
Added text to the description of the IDLE bit of TRCSTATR.	*TRCSTATR, Trace Status Register* on page 7-406.
Clarified text about software achieving imprecise filtering.	*TRCVICTLR, ViewInst Main Control Register* on page 7-413.
Added missing Address element in example one of basic program trace with filtering when an exception occurs.	Table A-11 on page A-435.
Added PUSH and POP instructions added to the list of load and store instructions.	Table F-9 on page F-470.
Added clarification for tracing of SWP and SWPB instructions.	*Armv7, Armv8-R, and Armv8-M A32 and T32 instruction sets* on page F-470.
Added clarification for the tracing of loading and storing doubleword registers in big-endian mode. Also added text about Armv6-M and Armv7-M PEs following the Address order 32 transfer index scheme.	*P1 element transfer index meanings* on page F-477.
Updated several existing glossary definitions, notably *RES0* and *RES1*.	• RES0. • RES1.
Added several new glossary entries, notably definitions for *RES0H* and *RES1H*.	• RES0H. • RES1H.

Table J-4 Differences between issue C and issue D

Change	Location
Updated VMID location for Armv8-R.	*Context ID tracing* on page 2-81.
Added new section to clarify trace unit behavior with a multi-threaded processor.	*Trace unit behavior for a multi-threaded processor* on page 3-111.
Added new section detailing changes due to a relaxation in architecture to permit the trace unit core power domain to be powered down when switching between PEs.	*Sharing the trace unit between multiple PEs* on page 3-111.

Table J-4 Differences between issue C and issue D (continued)

Change	Location
Added clarification about programming the ViewInst logic to be sensitive to comparators that are not programmed for instruction address comparators.	*Overview of the ViewInst function* on page 4-119.
Relaxed rules about System error exception tracing.	*Forcing tracing of exceptions* on page 4-130.
Added clarification for PEs that have more than 8 DWT comparators.	*PE comparator inputs* on page 4-159.
Added support for virtual addresses larger than 48 bits.	• *Exception instruction trace element* on page 5-196. • *Address instruction trace element* on page 5-209. • *TRCACVRn, Address Comparator Value Registers, n=0-15* on page 7-348.
Added a new exception type of SecureFault.	*Exception types for the Armv6-M, Armv7-M and Armv8-M architectures* on page 5-199.
Added clarification to footnote about Lazy FP push exception tracing and non-invasive debug.	*Exception types for the Armv6-M, Armv7-M and Armv8-M architectures* on page 5-199.
Added clarification about the change in the definition of a lockup address between Armv7-M and Armv8-M.	*Further information for tracing exceptions on Armv8-M* on page 5-205.
Added Function Return instruction trace element.	*Function Return instruction trace element* on page 5-214.
Added TRCAUTHSTATUS.HNID and TRCAUTHSTATUS.HID to indicate whether a separate debug enable for EL2 is implemented, and whether debug at EL2 is permitted.	*TRCAUTHSTATUS, Authentication Status register* on page 7-349.
Added two new signals to the Authentication interface to control debug in EL2, HIDEN and HNIDEN.	*TRCAUTHSTATUS, Authentication Status register* on page 7-349.
Added clarification for Armv6-M, Armv7-M, and Armv8-M PEs in Device Affinity register 0, and Device Affinity register 1.	• *TRCDEVAFF0, Device Affinity register 0* on page 7-364. • *TRCDEVAFF1, Device Affinity register 1* on page 7-364.
Added clarification about distinguishing between Secure and Non-secure states for the Armv8-M Security Extension.	• *TRCIDR3, ID Register 3* on page 7-376. • *TRCVICTLR, ViewInst Main Control Register* on page 7-413. • *TRCACATRn, Address Comparator Access Type Registers, n=0-15* on page 7-345. • *TRCAUTHSTATUS, Authentication Status register* on page 7-349. • *Context instruction trace packet* on page 6-292. • *Address with Context instruction trace packets* on page 6-294.
Added new values for TRCDEVARCH.REVISION and TRCIDR1.TRCARCHMIN.	• *TRCDEVARCH, Device Architecture register* on page 7-365. • *TRCIDR1, ID Register 1* on page 7-373.
Added support for sharing a trace unit between more than 8 PEs, up to 32 PEs.	• *TRCPROCSELR, Processing Element Select Control Register* on page 7-397. • *TRCIDR3, ID Register 3* on page 7-376.

Table J-4 Differences between issue C and issue D (continued)

Change	Location
Added recommendation about not implementing OS Lock on trace units for Armv6-M, Armv7-M, and Armv8-M PEs.	• *TRCOSLSR, OS Lock Status Register* on page 7-390. • Table C-1 on page C-456.
Updated definitions of A32 instruction set and T32 instruction set to include the VSEL instruction.	• *A32 instruction set* on page F-480. • *T32 instruction set* on page F-480.
Added two new instructions, BXNS and BLXNS.	• *T32 instruction set* on page F-467. • *Armv7, Armv8-R, and Armv8-M A32 and T32 instruction sets* on page F-470.
Added new sections in Instruction Categories appendix.	• *General-purpose register loads and stores* on page F-472. • *Extension register loads and stores* on page F-473. • *Multiple element loads and stores* on page F-474. • *Single element structure loads and stores to and from a single lane* on page F-476. • *Single element structure loads to multiple lanes* on page F-476.
Added new table showing the transfer groups and transfer indexes for exception stack push and pop data transfers.	• *Exception stack push and pop data transfers* on page F-479.
Added new table showing the transfer groups and transfer indexes for Lazy FP context save operation.	• *Lazy FP context save operation* on page F-479.

Table J-5 Differences between issue D and issue E

Change	Location
Updated terminology for a Context Synchronization Operation. It is now a Context synchronization event.	Throughout the document.
Removed ambiguous usage of ‘configuration’.	Throughout the document.
Replaced note about explicit tracing with a more constrained rule.	*Explicit tracing of data load and store instructions* on page 2-84.
Clarified entry to a prohibited region in Armv8-M.	*Trace unit behavior when tracing is prohibited* on page 3-108.
Corrected rule about exception tracing.	*Rules for tracing exceptions* on page 4-129
Added relaxation for PE Reset exceptions,	*Exception instruction trace element* on page 5-196
Added ETMv4.3 as a permitted architecture version.	*TRCDEVARCH, Device Architecture register* on page 7-365 and *TRCIDR1, ID Register 1* on page 7-373.
Added Armv8.3-A branch instructions.	*A64 instruction set* on page F-466.
Extended the address instruction trace element.	*Address instruction trace element* on page 5-209.

Table J-5 Differences between issue D and issue E (continued)

Change	Location
Included the minimal resource implementation.	Updated the following register descriptions: • *TRCCNTCTLRn, Counter Control Registers, n=0-3* on page 7-359. • *TRCIDR4, ID Register 4* on page 7-378. • *TRCIDR5, ID Register 5* on page 7-381. • *TRCEVENTCTL0R, Event Control 0 Register* on page 7-368. • *TRCEVENTCTL1R, Event Control 1 Register* on page 7-368. • *TRCSEQEVRn, Sequencer State Transition Control Registers, n=0-2* on page 7-400. • *TRCTSCTLR, Global Timestamp Control Register* on page 7-409. • *TRCVDCTLR, ViewData Main Control Register* on page 7-411. • *TRCVICTLR, ViewInst Main Control Register* on page 7-413.
Added support for WFI and WFE instructions.	Updated the following sections: • *Trace unit behavior on a PE low-power state* on page 3-105, • *Atom instruction trace element* on page 5-192. • *TRCIDR2, ID Register 2* on page 7-374.
Added the Ignore packets.	*Descriptions of instruction trace packets* on page 6-252 and *Descriptions of data trace packets* on page 6-307.

Table J-6 Differences between issue E and issue F

Change	Location
Rebranded company name from *ARM* to *Arm*.	Throughout the document.
Added two new rules to restrict the tracing of certain types of speculation.	*Trace behavior on speculative execution* on page 2-70.
Added note to clarify the need for explicit synchronization.	• *Saving and restoring the trace unit state* on page 3-94. • *Use of the trace unit main enable bit* on page 4-165. • *Synchronization when using system instructions to program the trace unit* on page 4-169.
Replaced text that incorrectly linked the operation of TRCVICTLR.TRCERR and TRCVICTLR.TRCRESET.	*Forcing tracing of exceptions* on page 4-130.
Added clarification about start/stop logic and incorrect speculation.	*Tracing one or more processes or threads of execution by using the ViewInst start/stop control* on page 4-120.
Added note to describe the behavior of the return stack when an N atom is mispredicted.	*Operation of the trace analyzer return stack* on page 5-224.
Corrected payload value for a Context packet from 0-6 bytes in issue E, to 0-9 bytes.	• Table 6-8 on page 6-246. • *Context instruction trace packet* on page 6-292.
Added text to clarify that the number of initial E Atoms.	*Atom Format 6 instruction trace packet* on page 6-303.
Corrected title of Figure 6-36 to read 64 bits.	*Address with Context Instruction trace packets that can indicate up to 64 bits of the address* on page 6-296.
Added new row to Table 7-3 to describe Unimplemented trace registers.	Table 7-3 on page 7-341.

Table J-6 Differences between issue E and issue F (continued)

Change	Location
Updated *Reserved trace* to read *Reserved trace and unimplemented trace*, and updated *Reserved management* to read *Reserved management and unimplemented management*.	• Table 7-4 on page 7-342. • Table 7-5 on page 7-343.
Added footnote c to Table 7-6 to relax specification for Unimplemented Trace registers,	Table 7-6 on page 7-344.
The requirement to trace the full invalid address is relaxed.	• *Exception instruction trace element* on page 5-196 • *Address instruction trace element* on page 5-209
Relaxed the rules about the inclusion of the Virtual context identifier field and the Context ID field in the tracing of the Context instruction trace packet.	*Context instruction trace packet* on page 6-292.
Relaxed the rules about the inclusion of the Virtual context identifier field and the Context ID field in the tracing of the Address with Context instruction trace packet.	*Address with Context instruction trace packets* on page 6-294.

Table J-7 Differences between issue F and issue G.a

Change	Location
Added new rule to clarify timestamp size.	*Global timestamping* on page 2-82.
Updated figures and text to account for management registers which reside in both power domains.	• *An example where the trace unit core power domain is connected to the PE core power domain* on page 3-92. • *Trace unit powerdown support* on page 3-94. • *A combined powerup request signal* on page 3-102. • *Register map* on page 4-163. • *About accesses to registers in different trace unit power domains* on page 4-164.
Extended text to include ignore packets.	*Trace unit behavior on a trace buffer overflow* on page 3-106.
Added text to indicate that exceptions taken from prohibited regions are not traced.	*Forcing tracing of exceptions* on page 4-130.
Clarified text relating to examples.	*Counters* on page 4-139.
Added notes to relax comparator behavior around MOVPRFX instructions.	• *Single address comparators* on page 4-149. • *Address range comparators* on page 4-151.
Added text to clarify support for 64-bit and system instructions	*64-bit support* on page 4-168.
Clarified potentially misleading text.	*Resource grouping* on page 4-173.
Clarified text which conflicts with function definitions.	*Activating a trace unit event with a selected trace unit resource or pair of trace unit resources* on page 4-177.
Added new text to define behavior for when an imprecise exception occurs.	*Architectural exceptions* on page 5-197.
Corrected a value which was incorrect for Armv8-M.	*Exception Return instruction trace element* on page 5-208.
Updated text to prohibit timestamp requests when certain operations happen in prohibited regions.	*Timestamp instruction trace element* on page 5-215.

Table J-7 Differences between issue F and issue G.a (continued)

Change	Location
Added missing entries to tables.	• *Behaviors when using an external debugger* on page 7-341. • *Behaviors when using memory-mapped access* on page 7-342. • *Behaviors when using system instructions* on page 7-343.
Added additional restriction to TRCLSR.SLI.	*TRCLSR, Software Lock Status Register* on page 7-389.
Corrected text in TRCSSCSRn.STATUS	*TRCSSCSRn, Single-shot Comparator Status Registers, n=0-7* on page 7-403.
Corrected wrong value in table entry.	*Program trace containing a context changing operation (exception immediately after ISB)* on page A-441.

Table J-8 Differences between issue G.a and issue G.b

Change	Location
Added new examples of how new elements can be used to trace typical sequences of code.	Appendix A *Examples of Trace*.

Glossary

A32 instruction An instruction that is executed by a PE that is in AArch32 Execution state and A32 Instruction set state. A32 is a fixed-width instruction set that uses 32-bit instruction encodings. Previously, this instruction set was called the Arm instruction set.

See also AArch32 State, A32 state, A64 instruction, and T32 instruction.

A32 state When a core is in the AArch32 Execution state, if it is in the A32 Instruction set state then it executes A32 instructions.

See also AArch32 State, T32 instruction, T32 state.

A64 instruction The instruction set used by an Armv8-A core that is in AArch64 Execution state. A64 is a fixed-width instruction set that uses 32-bit instruction encodings.

See also A32 instruction and T32 instruction.

AArch32 State The Arm 32-bit Execution state that uses 32-bit general purpose registers, and a 32-bit program counter (PC), stack pointer (SP), and link register (LR). AArch32 Execution state provides a choice of two instruction sets, A32 and T32.

In implementations of versions of the Arm architecture before Armv8, and in the Arm R and M architecture profiles, execution is always in AArch32 state.

See also AArch64 State, A32 instruction and T32 instruction.

AArch64 State The Arm 64-bit Execution state that uses 64-bit general purpose registers, and a 64-bit program counter (PC), stack pointer (SP), and exception link registers (ELR). AArch64 Execution state provides a single instruction set, A64.

AArch64 state is supported only in the Armv8-A architecture profile.

See also AArch32 State, A64 instruction.

Abort
An abort occurs when an illegal memory access causes an exception. An abort can be generated by the hardware that manages memory accesses, or by the external memory system. The hardware that generates the abort might be a *Memory Management Unit* (MMU) or a *Memory Protection Unit* (MPU).

See also Data Abort and Prefetch Abort.

Aligned
A data item that is stored at an address that is exactly divisible by the number of bytes that defines its data size. Aligned doublewords, words, and halfwords have addresses that are divisible by eight, four, and two respectively. An aligned access is one where the address of the access is aligned to the size of each element of the access.

Application Program Status Register (APSR)
In AArch32 state, a view of the CPSR that is accessible by unprivileged software.

See also *Current Program Status Register (CPSR) on page Glossary-525*.

APSR
See *Application Program Status Register (APSR)*

Architecturally executed
An instruction is architecturally executed only if it would be executed in a simple sequential execution of the program. When such an instruction has been executed and retired is has been *architecturally executed*. Any instruction that, in a simple sequential execution of a program, is treated as a `NOP` because it fails its condition code check, is an architecturally-executed instruction.

In a PE that performs speculative execution, an instruction is not architecturally executed if the PE discards the results of a speculative execution.

Architecture tick
An atomic unit of execution. In the ARMv8-M architecture most instructions are considered atomic units of execution (they are either performed or not). The deviations from this behavior are instructions that support ICI behavior.

Arm instruction
See A32 instruction.

ARM state
See AArch32 State.

ATB
An AMBA bus protocol for trace data. A trace device can use an ATB to share CoreSight capture resources.

Banked register
A register that has multiple instances. A property of the state of the device determines which instance is in use. For example, the PE mode or Security state might determine which instance is in use.

Base register
A register that is specified by a load/store instruction that is used as the base value for the address calculation for the instruction. Depending on the instruction, an offset can be added to or subtracted from the base register value to form the address that is used for the memory access.

Beat
The execution of one quarter of a vector operation. Due to the vector length being128 bits, one beat of a vector add instruction equates to computing 32 bits of result data.

Note

This is independent of lane width; for example, if the lane width is 8 bits, then a single beat of a vector add instruction would perform four 8 bit additions.

BF branch point
The BF branch point is the interstice between two instructions, that is the gap between the instruction that immediately precedes the instruction at the branch label and the instruction that is identified by the branch label.

Big-endian
In the context of the Arm architecture, big-endian is defined as the memory organization in which, for example:

- A byte or halfword at a word-aligned address is the most significant byte or halfword at that address.
- A byte at a halfword-aligned address is the most significant byte in the halfword at that address.

See also Little-endian.

Branch Future (BF)
The Armv8.1-M architecture supports branch future instructions. The BF instruction and its variants notify the PE that a branch will be taken in the future.

Breakpoint A debug event that is triggered by the execution of a particular instruction. It is specified by one or both of the address of the instruction and the state of the PE when the instruction is executed.

See also Watchpoint.

Byte An 8-bit data item.

Condition code check
The process of determining whether a conditional instruction executes normally or is treated as a `NOP`. For an instruction that includes a condition code field, that field is compared with the condition flags to determine whether the instruction is executed normally.

In architecture predating Armv8 architecture, for a T32 instruction in an IT block, the value of the ITSTATE register determines whether the instruction is executed normally.

In Armv8, for a T32 instruction in an IT block, the value of PSTATE.IT determines whether the instruction is executed normally.

See also Condition code field, Condition flags, Conditional execution.

Condition code field
A four-bit field in an instruction that specifies the condition under which the instruction executes.

See also Condition code check.

Condition flags The N, Z, C, and V bits of a *Program Status Register* (PSR) or process state (PSTATE). In addition, in Armv8 architecture, the N, Z, C, and V bits of a *Saved Program Status Register* (SPSR), or *Floating-Point Status and Control Register* (FPSCR).

See also Condition code check, Current Program Status Register (CPSR), Saved Program Status Register (SPSR), and Process State (PSTATE).

Conditional execution
When the conditional instruction starts executing, if the condition code check returns TRUE, the instruction executes normally. Otherwise, it is treated as a `NOP`.

See also Condition code check.

Context synchronization event
A Context synchronization event is one of:

- Performing an ISB operation. An ISB operation is performed when an `ISB` instruction is executed and does not fail its condition code check.
- Taking an exception.
- Returning from an exception.

On an Armv8-A PE, the following events are also Context synchronization events:

- Exit from Debug state.
- Executing a `DCPS` instruction.
- Executing a `DRPS` instruction.

The effects of a Context synchronization event are:

- All unmasked interrupts that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- If halting is allowed, all Halting debug events that are pending at the time of the Context synchronization event are taken before the first instruction after the Context synchronization event.
- No instructions appearing in program order after an instruction that causes a Context synchronization event will have performed any part of their functionality until the Context synchronization event has occurred.
- All direct and indirect writes to System registers that are made before the Context synchronization event affect any instruction, including a direct read, that appears in program order after the instruction causing the Context synchronization event.

- All invalidations of TLBs, instruction caches, and, in AArch32 state, branch predictors, that are completed before the Context synchronization event, affect all instructions that appear in program order after an instruction causing a Context synchronization event.
- In AArch32 state, all Non-cacheable writes that are completed before the Context synchronization event affect all instructions that appear in program order after an instruction causing a Context synchronization event.

Note

- The architecture requires that instructions that generate instruction synchronization events do not appear to be executed speculatively other than it is permissible that the performance counters can reveal such speculation.
- *Context synchronization events* were previously described as *Context synchronization operations*.

See also Condition code check.

Coprocessor
A processor, or conceptual processor, that supplements the main processor to carry out additional functions. Before Armv8 the Arm architecture defines an interface to up to 16 coprocessors, CP0-CP15, where coprocessors CP8-CP15 are reserved for use by Arm. CP15, CP14, CP10, and CP11 are used as follows:

- CP15 instructions access the System Control coprocessor.
- CP14 instructions access control registers for debug, trace, and execution environment features.
- CP10 and CP11 instruction space is for floating-point and Advanced SIMD instructions, if supported.

In Armv8, AArch32 state retains this use of CP10, CP11, CP14, and CP15, but does not otherwise support the coprocessor interface.

Core register
Processing registers used in AArch32 Execution state, comprising:

- 13 general-purpose registers, R0 to R12, that software can use for processing.
- SP, the Stack Pointer, that can also be referred to as R13.
- LR, the Link Register, that can also be referred to as R14.
- PC, the Program Counter, that can also be referred to as R15.

In AArch32 state, in some situations, software can use SP and LR for processing. The instruction descriptions include any constraints on the use of SP, LR, and PC.

See also Program Counter (PC), Stack Pointer (SP), and Link Register (LR).

CoreSight
Arm on-chip debug and trace components, that provide the infrastructure for monitoring, tracing, and debugging a complete system on chip.

See also CoreSight ECT and CoreSight ETM.

CoreSight ECT
See Embedded Cross Trigger (ECT).

CoreSight ETB
See Embedded Trace Buffer (ETB).

CoreSight ETM
See Embedded Trace Macrocell (ETM).

CoreSight Granular Power Requestor
See Arm® CoreSight™ Architecture Specification.

CPSR
See Current Program Status Register (CPSR).

Cross Trigger Interface (CTI)
Part of an *Embedded Cross Trigger* (ECT) device. In an ECT, the CTI provides the interface between a PE or ETM and the CTM.

See also Embedded Cross Trigger (ECT).

Cross Trigger Matrix (CTM)
Part of an Embedded Cross Trigger (ECT) device. In an ECT, the CTM combines the trigger requests generated by CTIs and broadcasts them to all CTIs as channel triggers.

See also Embedded Cross Trigger (ECT).

CTI *See* Cross Trigger Interface (CTI).

Current Program Status Register (CPSR)
In AArch32 state, the register that holds the current PE status.

See also Program Status Register (PSR), Saved Program Status Register (SPSR), Application Program Status Register (APSR) and Process State (PSTATE).

D_BRANCH_INFO
New data structure that a trace analyzer requires when analyzing trace that contains BF instructions.

Data Abort An indication to the PE of an attempted data access that is not permitted. The Data Abort might be generated by access permission checks performed by the memory system on the PE, or might be signaled by the memory system.

See also Abort and Prefetch Abort.

Debugger A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Deprecated Something that is present in the Arm architecture for backwards compatibility. Whenever possible, software must avoid using deprecated features. Features that are deprecated but not optional are present in current implementations of the Arm architecture, but might not be present, or might be deprecated and OPTIONAL, in future versions of the Arm architecture.

See also OPTIONAL.

Device In the context of an Arm debugger, a component on a target containing the application that you want to debug.

See also Target.

Doubleword A 64-bit data item. Doublewords are normally at least word-aligned in Arm systems.

Doubleword-aligned
A data item having a memory address that is divisible by eight.

Embedded Cross Trigger (ECT)
A modular system that supports the interaction and synchronization of multiple triggering events with an SoC. It comprises:

- *Cross Trigger Interface* (CTI).
- *Cross Trigger Matrix* (CTM).

Embedded Trace Buffer (ETB)
A Logic block that extends the information capture functionality of a trace macrocell.

Embedded Trace Macrocell (ETM)
A hardware macrocell that, when connected to a PE, outputs trace information on a trace port. The ETM provides PE driven trace through a trace port compliant to the ATB protocol. An ETM always supports instruction trace, and might support data trace.

End address The address of the instruction referenced by a branch future instruction. This is the address after the BF branch point.

End address size
The size of the instruction at the End address.

Endianness The scheme that determines the order of successive bytes of data in a larger data structure when that structure is stored in memory.

See also Big-endian and Little-endian.

ETB *See* Embedded Trace Buffer (ETB).

ETM *See* Embedded Trace Macrocell (ETM).

Event In an Arm trace macrocell:

Simple An observable condition that a trace macrocell can use to control aspects of a trace.

Complex A boolean combination of simple events that a trace macrocell can use to control aspects of a trace.

Event is used with different meanings in other contexts. For example, the Arm Performance Monitors define a set of events, and can count the occurrences of those events.

Exception A mechanism to handle a fault, error event, or external notification. For example, exceptions handle external interrupts and undefined instructions.

Exception continuable
Any instruction that can cause the PE to set the EPSR.ECI fields.

See also the *Arm®v8-M Architecture Reference Manual*.

Fault An abort that is generated by the memory system, for example by the *Memory Management Unit* (MMU) or *Memory Protection Unit* (MPU).

FIQ FIQ interrupt. **nFIQ** is one of two interrupt signals on the A-profile and R-profile Arm PEs and on earlier Arm PEs.

See also IRQ.

GPR *See* CoreSight Granular Power Requestor.

Halfword A 16-bit data item. Halfwords are normally halfword-aligned in Arm systems.

Halfword-aligned
A data item having a memory address that is divisible by 2.

IMPLEMENTATION DEFINED
Behavior that is not defined by the architecture, but must be defined and documented by individual implementations.

When IMPLEMENTATION DEFINED appears in body text, it is always shown in SMALL CAPITALS.

IMPLEMENTATION SPECIFIC
In Arm trace architecture specifications, behavior that is not architecturally defined, and might not be documented by an individual implementation. Used when there are several implementation options available and the option that is chosen does not affect software compatibility.

When IMPLEMENTATION SPECIFIC appears in body text, it is always shown in SMALL CAPITALS.

See also IMPLEMENTATION DEFINED.

Implicit branch A change in the program flow caused by a branch future instruction and the LO_BRANCH_INFO operation in the PE.

Imprecise tracing

In an Arm trace macrocell, a filtering scenario where instruction or data tracing can start or finish earlier or later than expected. Most imprecise cases cause tracing to start or finish later than expected.

For example, if **TraceEnable** logic is programmed to use a counter so that tracing begins after the fourth write to a location in memory, the instruction that caused the fourth write is not traced, although subsequent instructions are. This is because the use of a counter in the **TraceEnable** programming always results in imprecise tracing.

Instruction abort

An indication to the core of an attempted instruction fetch that is not permitted. The Instruction Abort might be generated by access permission checks performed by the memory system on the PE, or might be signaled by the memory system. An exception is taken only if the PE attempts to execute the instruction. No exception is taken if the PE does not execute an instruction that it attempted to fetch or prefetch from a faulting memory location.

The AArch64 architecture definitions introduce the term *Instruction Abort*. Descriptions of AArch32 state use the term *Prefetch Abort*.

Instruction Synchronization Barrier (ISB)
An operation to ensure that any instruction that comes after the ISB operation is fetched only after the ISB has completed.

Intermediate Physical Address (IPA)
An implementation of virtualization, the address to which a Guest OS maps a virtual address (VA). A hypervisor might then map the IPA to a PA. Typically, the Guest OS is unaware of the translation from IPA to PA.

See also Physical Address (PA), Virtual Address (VA).

Interrupt continuable
Multicycle load or store instructions that can be interrupted part way through their execution. After the interrupt service routine has completed, execution of the partially executed instruction can be resumed and the instruction is not required to be restarted from the beginning. An interruption of a multicycle load or store instructions will cause the PE to set the EPSR.ICI fields.

See also the *Arm®v8-M Architecture Reference Manual*.

IRQ IRQ interrupt. **nIRQ** is one of two interrupt signals on the A-profile and R-profile Arm PEs, and on earlier Arm PEs.

See also FIQ.

ISB *See* Instruction Synchronization Barrier (ISB).

IT block In T32 state, a block of up to four instructions following a T32 If Then (`IT`) instruction. Each instruction in the block is conditional. The condition for each instruction is either the same as or the inverse of the condition that is specified by the `IT` instruction. From the introduction of Armv8, Arm deprecates having more than one instruction in any IT block.

Jazelle state In AArch32 state, in the Jazelle Instruction set state the core executes Java bytecodes as part of a *Java Virtual Machine* (JVM).

From Armv8, Jazelle state is not supported.

See also A32 state, T32 state.

JTAG *See* Joint Test Action Group (JTAG).

Joint Test Action Group (JTAG)
An IEEE group that is focused on silicon chip testing methods. Many debug and programming tools use a *Joint Test Action Group* (JTAG) interface port to communicate with PEs.

See *IEEE Std 1149.1-1990 IEEE Standard Test Access Port and Boundary-Scan Architecture* specification available from the IEEE Standards Association `http://standards.ieee.org`.

JTAG Access Port (JTAG-AP)
An optional component of the DAP that provides debugger access to on-chip scan chains.

Lane A section of a vector register or operation.

LE/LETP Loop End instruction with and without tail prediction.

Link Register (LR)
On Arm PEs, LR refers to the Link Register for exception returns, and:

- In AArch32 state, the LR is register R14 in the general-purpose register file.
- In AArch64 state, there is a dedicated LR for each implemented Exception level, called the Exception Link Register (ELR) for that Exception level, for example, ELR_EL1.

See also Core register, Program Counter (PC), and Stack Pointer (SP).

Little-endian In the context of the Arm architecture, little-endian is defined as the memory organization in which the most significant byte of a word is at a higher address than the least significant byte.

See also Big-endian.

LO_BRANCH_INFO
New PE register for storing BF and low overhead loop information as defined in the *Arm®v8-M Architecture Reference Manual*.

LOB M-profile Low Overhead loops and Branch future Extension.

MVE M-profile Vector Extension.

OPTIONAL When applied to a feature of the architecture, OPTIONAL indicates a feature that is not required in an implementation of the Arm architecture:

- If a feature is OPTIONAL and deprecated, this indicates that the feature is being phased out of the architecture.

  Arm expects such features to be included in a new implementation only if there is a known backwards-compatibility reason for the inclusion of the feature.

  A feature that is OPTIONAL and deprecated might not be present in future versions of the architecture.

- A feature that is OPTIONAL but not deprecated is, typically, a feature added to a version of the Arm architecture after the initial release of that version of the architecture. Arm recommends that such features are included in all new implementations of the architecture.

In body text, these meanings of the term OPTIONAL are shown in SMALL CAPITALS.

Do not confuse these Arm-specific uses of OPTIONAL with other uses of optional, where optional it has its usual meaning. These include:

- Optional arguments in the syntax of many instructions.
- Behavior that is determined by an implementation choice, for example the optional byte order reversal in an Armv7-R implementation, where the SCTLR.IE bit indicates the implemented option.

See also Deprecated.

P0 instruction Instructions that generate an *Atom element* when traced.

PE *See* Processing element (PE).

Physical Address (PA)
An address that identifies a location in the physical memory map.

See also Virtual Address (VA).

Predicate Defines which lanes are active in the vector operation.

Preferred exception return address
The address included in an *Exception element* which indicates where execution had reached before the exception was taken.

Prefetch Abort The AArch32 name for an *Instruction abort*.

See also Abort, Data Abort, and Instruction abort.

Processing element (PE)
The abstract machine that is defined in the Arm architecture, as documented in an Arm Architecture Reference Manual. A PE implementation compliant with the Arm architecture must conform with the behaviors described in the corresponding Arm Architecture Reference Manual.

Profiling In the context of RealView Trace, the accumulation of statistics during execution of a program to measure performance or to determine critical areas of code.

Program Counter (PC)
The PC holds the virtual address of the next instruction to be executed, and:

- In AArch32 state, the PC is a 32-bit register, and is register R15 in the general-purpose register file.

- In AArch64 state, the PC is a 64-bit register, that is independent of the general-purpose registers X0-X30. In AArch64 state, software cannot write directly to the PC.

See also Core register, Link Register (LR), and Stack Pointer (SP).

Program Status Register (PSR)

Holds PE status and control information for a core, or for a program running on the core. When executing in AArch32 state, the *Current Program Status Register* (CPSR) is the active PSR that affects PE operation, and the *Application Program Status Register* (APSR) is the unprivileged view of that state. When executing in AArch64 state, PSTATE holds equivalent status information, but is not accessible as a single register. For more information, see Process State (PSTATE).

The Saved *Program Status Register* (SPSR) is a copy of the current state of the cores that are saved by the hardware on taking an exception. At the point where a core recognizes an exception:

- If the exception is taken to AArch32 state, it copies the CPSR into the SPSR.
- If the exception is taken to AArch64 state, it saves the current PSTATE to the SPSR.

See the appropriate *Arm Architecture Reference Manual* for more information.

See also Application Program Status Register (APSR), Current Program Status Register (CPSR), and Saved Program Status Register (SPSR).

PSR *See* Program Status Register (PSR).

Process State (PSTATE)

From Armv8, Process State is an abstraction of the PE state that must be saved on taking an exception. After handling the exception, the Process State can be restored, so the PE can resume execution from the point where it took the exception. On taking an exception, PSTATE is saved in the SPSR for the Exception level and Execution state for which the exception is taken.

All instruction sets include instructions that operate on elements of Process State. In AArch32 state, the CPSR holds the applicable elements of Process State.

See also Program Status Register (PSR), Current Program Status Register (CPSR), Saved Program Status Register (SPSR), and Application Program Status Register (APSR).

PSTATE *See* Process State (PSTATE).

RAO *See* Read-As-One (RAO).

RAZ *See* Read-As-Zero (RAZ).

RAO/SBOP Read-As-One, Should-Be-One-or-Preserved on writes.

Hardware must implement the field as Read-as-One, and must ignore writes to the field. Software can rely on the field reading as all 1s, but must use an SBOP policy to write to the field. This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also UNK/SBOP, Read-As-One (RAO), RES1, Should-Be-One-or-Preserved (SBOP).

RAO/WI Read-As-One, Writes Ignored.

Hardware must implement the field as Read-As-One, and must ignore writes to the field. Software can rely on the field reading as all 1s, and on writes being ignored. This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also Read-As-One (RAO).

RAZ/SBZP In versions of the Arm architecture before Armv8, Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In any implementation, hardware must implement the field as Read-as-Zero, and must ignore writes to the field. Software can rely on the bit reading as 0, or all 0s for a bit field, but must use an SBZP policy to write to the field. This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also UNK/SBZP, Read-As-Zero (RAZ), RES0, Should-Be-Zero-or-Preserved (SBZP).

RAZ/WI Read-As-Zero, Writes Ignored.

Hardware must implement the field as Read-As-One, and must ignore writes to the field. Software can rely on the bit reading as 0, or all 0s for a bit field, and on writes being ignored. This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

See also Read-As-Zero (RAZ).

Read-As-One (RAO)

Hardware must implement the field as reading as all 1s. Software can rely on the field reading as all 1s. This description can apply to a single bit that reads as 1, or to a field that reads as all 1s.

See also RAO/SBOP, RAO/WI, RES1.

Read-As-Zero (RAZ)

Hardware must implement the field as reading as all 0s. Software can rely on the field reading as all 0s. This description can apply to a single bit that reads as 0, or to a field that reads as all 0s.

RES0 A reserved bit. Used for fields in register descriptions, and for fields in architecturally defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES0 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES0 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is *Architecturally executed* to an AArch64 System register, and a bit or field in that register is RES0 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES0 for fields in read/write registers is:

If a bit is RES0 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 0. In this case:
   - Reads of the bit always return 0.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 0.
   - A read of the bit returns the last value that is successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
   - A direct write to the bit must update a storage location that is associated with the bit.
   - The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES0 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

If a bit is RES0 only in some contexts

For a bit in a read/write register, when the bit is described as RES0:

- An indirect write to the register sets the bit to 0.
- A read of the bit must return the value last successfully written to the bit, by either a direct or an indirect write, regardless of the use of the register when the bit was written.

If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location that is associated with the bit.
- While the use of the register is such that the bit is described as RES0, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value that was written to the bit.

The RES0 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES0 indicates that the bit reads as 0, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES0 indicates that software must treat the bit as SBZ.

A bit that is RES0 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 0.
- Must use an SBZP policy to write to the bit.

This RES0 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES0.

In body text, the term RES0 is shown in SMALL CAPITALS.

See also Read-As-Zero (RAZ), RES1, Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

RES0H A reserved bit or field with *Should-Be-Zero-or-Preserved (SBZP)*. This behavior uses the *Hardwired to 0* subset of the RES0 definition.

RES1 A reserved bit. Used for fields in register descriptions, and for fields in architecturally defined data structures that are held in memory, for example in translation table descriptors.

Within the architecture, there are some cases where a register bit or field:

- Is RES1 in some defined architectural context.
- Has different defined behavior in a different architectural context.

Note

- RES1 is not used in descriptions of instruction encodings.
- Where an AArch32 System register is *Architecturally executed* to an AArch64 System register, and a bit or field in that register is RES1 in one Execution state and has defined behavior in the other Execution state, this is an example of a bit or field with behavior that depends on the architectural context.

This means the definition of RES1 for fields in read/write registers is:

If a bit is RES1 in all contexts

For a bit in a read/write register, it is IMPLEMENTATION DEFINED whether:

1. The bit is hardwired to 1. In this case:
   - Reads of the bit always return 1.
   - Writes to the bit are ignored.
2. The bit can be written. In this case:
   - An indirect write to the register sets the bit to 1.
   - A read of the bit returns the last value that was successfully written, by either a direct or an indirect write, to the bit.

     If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.

- A direct write to the bit must update a storage location that is associated with the bit.
- The value of the bit must have no effect on the operation of the PE, other than determining the value read back from the bit, unless this Manual explicitly defines additional properties for the bit.

Whether RES1 bits or fields follow behavior 1 or behavior 2 is IMPLEMENTATION DEFINED on a field-by-field basis.

If a bit is RES1 only in some contexts

For a bit in a read/write register, when the bit is described as RES1:

- An indirect write to the register sets the bit to 1.
- A read of the bit must return the value last successfully written to the bit, regardless of the use of the register when the bit was written.

  Note

  As indicated in this list, this value might be written by an indirect write to the register.

  If the bit has not been successfully written since reset, then the read of the bit returns the reset value if there is one, or otherwise returns an UNKNOWN value.
- A direct write to the bit must update a storage location that is associated with the bit.
- While the use of the register is such that the bit is described as RES1, the value of the bit must have no effect on the operation of the PE, other than determining the value read back from that bit, unless this Manual explicitly defines additional properties for the bit.

Considering only contexts that apply to a particular implementation, if there is a context in which a bit is defined as RES0, another context in which the same bit is defined as RES1, and no context in which the bit is defined as a functional bit, then it is IMPLEMENTATION DEFINED whether:

- Writes to the bit are ignored, and reads of the bit return an UNKNOWN value.
- The value of the bit can be written, and a read returns the last value that was written to the bit.

The RES1 description can apply to bits or fields that are read-only, or are write-only:

- For a read-only bit, RES1 indicates that the bit reads as 1, but software must treat the bit as UNKNOWN.
- For a write-only bit, RES1 indicates that software must treat the bit as SBO.

A bit that is RES1 in a context is reserved for possible future use in that context. To preserve forward compatibility, software:

- Must not rely on the bit reading as 1.
- Must use an SBOP policy to write to the bit.

This RES1 description can apply to a single bit, or to a field for which each bit of the field must be treated as RES1.

In body text, the term RES1 is shown in SMALL CAPITALS.

See also Read-As-One (RAO), RES0, Should-Be-One-or-Preserved (SBOP), UNKNOWN.

RES1H A reserved bit or field with *Should-Be-One-or-Preserved (SBOP)* behavior. This behavior uses the *Hardwired to 1* subset of the RES1 definition.

Reserved Unless otherwise stated in the architecture or product documentation:

- Reserved instruction and 32-bit system control register encodings are UNPREDICTABLE.
- Reserved 64-bit system control register encodings are undefined.
- Reserved register bit fields are UNK/SBZP.

Resynchronization element

A new element to cause the trace analyzer to perform a resynchronization routine.

RETPSR Combined Exception Return Program Status Registers.

Saved Program Status Register (SPSR)
A register that is used to save the PE state on taking an exception. For more information about the use of the SPSR see *Program Status Register (PSR)*.

See also Current Program Status Register (CPSR) and Process State (PSTATE).

SBO *See* Should-Be-One (SBO).

SBOP *See* Should-Be-One-or-Preserved (SBOP).

SBZ *See* Should-Be-Zero (SBZ).

SBZP *See* Should-Be-Zero-or-Preserved (SBZP).

Serial Wire Debug (SWD)
A debug implementation that uses a serial connection between the SoC and a debugger. This connection normally requires a bidirectional data signal and a separate clock signal, rather than the four to six signals that are required for a JTAG connection.

Serial Wire Debug Port (SW-DP)
The interface for Serial Wire Debug.

Should-Be-One (SBO)
Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 1s. If software writes a value that is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as 1, or to a field that should be written as all 1s.

Should-Be-One-or-Preserved (SBOP)
From the introduction of the Armv8 architecture, the description *Should-Be-One-or-Preserved (SBOP)* is superseded by *RES1*.

Hardware must ignore writes to the field.

Note

The Armv7 Large Physical Address Extension modified the definition of SBOP for register bits that are SBOP in some but not all contexts. The behavior of these bits is covered by the *RES1* definition, but not by the generic definition of SBOP given here.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 1s.

If software writes a value to the field that is not a value that is previously read for the field and is not all 1s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Should-Be-Zero-or-Preserved (SBZP) and Should-Be-One (SBO).

Should-Be-Zero (SBZ)
Hardware must ignore writes to the field.

Arm strongly recommends that software writes the field as all 0s. If software writes a value that is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as 0, or to a field that should be written as all 0s.

Should-Be-Zero-or-Preserved (SBZP)
From the introduction of the Armv8 architecture, the description *Should-Be-Zero-or-Preserved (SBZP)* is superseded by *RES0*.

Note

The Armv7 Large Physical Address Extension modified the definition of SBZP for register bits that are SBZP in some but not all contexts. The behavior of these bits is covered by the *RES0* definition, but not by the generic definition of SBZP given here.

Hardware must ignore writes to the field.

If software has read the field since the PE implementing the field was last reset and initialized, it must preserve the value of the field by writing the value that it previously read from the field. Otherwise, it must write the field as all 0s.

If software writes a value to the field that is not a value that is previously read for the field and is not all 0s, it must expect an UNPREDICTABLE result.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Should-Be-One-or-Preserved (SBOP) and Should-Be-Zero (SBZ).

SIMD Single-Instruction, Multiple-Data.

In the Arm instruction sets, supported SIMD instructions can comprise:

- Instructions that perform parallel operations on the bytes or halfwords of the Arm core registers.
- Instructions that perform vector operations. That is, they perform parallel operations on vectors that are held in multiword registers.

Different versions of the Arm architecture support and recommend different instructions for vector operations. See the appropriate version of the *Arm Architecture Reference Manual* for more information.

SPSR *See* Saved Program Status Register (SPSR).

Stack Pointer (SP)

On Arm PEs, SP refers to the stack pointer for the hardware-managed stack, and:

- In AArch32 state, the SP is register R13 in the general-purpose register file.
- In AArch64 state, there is a dedicated SP for each implemented Exception level.

See also Program Counter (PC) and Link Register (LR).

Supervisor Call (SVC)

An instruction that causes the PE to take a Supervisor Call exception.

Used by the Arm standard C library to handle semihosting. The Supervisor Call was previously called SoftWare Interrupt (SWI).

SVC *See* Supervisor Call (SVC).

SWD *See* Serial Wire Debug (SWD).

SWDP *See* Serial Wire Debug Port (SW-DP).

T32 instruction An instruction set that can be used by a PE that is in AArch32 execution state. T32 is a variable-length instruction set that uses both 16-bit and 32-bit instruction encodings. It is the only instruction set supported by Arm M-profile processors.

T32 instructions must be halfword-aligned.

The T32 instruction set was previously called the Thumb instruction set.

See also T32 state and T32EE state.

T32 state When a PE is executing in AArch32 state, if it is in T32 Instruction set state then it executes T32 instructions.

The T32 state was previously called Thumb state.

See also AArch32 State, Jazelle state, and T32EE state.

T32EE instruction

One or two halfwords that specify an operation for a PE in AArch32 state in T32EE Instruction set state to perform. T32EE is the T32 Execution Environment and the T32EE instruction set is based on the T32 instruction set, with some changes and additions to make it a better target for dynamically generated code, that is, code that is compiled on the device either shortly before or during execution.

In Armv8, support for T32EE is OPTIONAL and deprecated. ETMv4 does not support tracing of T32EE instructions.

See also A32 instruction, A64 instruction and T32 instruction.

T32EE state

In AArch32 state, in the T32EE Instruction set state the core executes the T32EE instruction set.

In Armv8, support for T32EE is OPTIONAL and deprecated. ETMv4 does not support tracing of the T32EE state.

See also A32 state,T32 state, and Jazelle state.

Target

In the context of an Arm debugger, the part of the development platform to which the debugger can connect, and on which debugging operations can be performed. A target can be:

- A runnable target, such as a PE that implements the Arm architecture. When connected to a runnable target, you can perform execution-related debugging operations on that target, such as stepping and tracing.
- A non-runnable CoreSight component. CoreSight components provide a system-wide solution to real-time debug and trace.

Thumb instruction

See T32 instruction.

Thumb state

See T32 state.

Trace port

A port on a device, such as a processor or ASIC, used to output trace information.

Translation table

A table, held in memory, that contains descriptors that define the properties of regions of memory, and the mapping between a supplied input address and the corresponding output address.

Note

In an Arm system without support for virtualization, the input address is a physical address and the output address is a virtual address.

See also Physical Address (PA) and Virtual Address (VA).

Trigger

In the context of tracing, a trigger is an event that instructs the debugger to stop collecting trace and display the trace information around the trigger position, without halting the PE. The exact information that is displayed depends on the position of the trigger within the buffer.

UNK

An abbreviation indicating that software must treat a field as containing an UNKNOWN value.

In any implementation, the bit must read as 0, or all 0s for a bit field. Software must not rely on the field reading as zero.

See also UNKNOWN.

UNK/SBOP

Hardware must implement the field as Read-As-One, and must ignore writes to the field.

Software must not rely on the field reading as all 1s, and except for writing back to the register it must treat the value as if it is UNKNOWN. Software must use an SBOP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 1, or to a field that should be written as its preserved value or as all 1s.

See also Read-As-One (RAO), Should-Be-One-or-Preserved (SBOP), UNKNOWN.

UNK/SBZP

Hardware must implement the bit as Read-As-Zero, and must ignore writes to the field.

Software must not rely on the field reading as all 0s, and except for writing back to the register must treat the value as if it is UNKNOWN. Software must use an SBZP policy to write to the field.

This description can apply to a single bit that should be written as its preserved value or as 0, or to a field that should be written as its preserved value or as all 0s.

See also Read-As-Zero (RAZ), Should-Be-Zero-or-Preserved (SBZP), UNKNOWN.

UNKNOWN

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole or documented or promoted as having a defined value or effect.

See also UNK.

UNP

See UNPREDICTABLE.

UNPREDICTABLE

For an Arm PE, UNPREDICTABLE means that the behavior cannot be relied on. UNPREDICTABLE behavior must not perform any function that cannot be performed at the current or a lower level of privilege using instructions that are not UNPREDICTABLE.

UNPREDICTABLE behavior must not be documented or promoted as having a defined effect. An instruction that is UNPREDICTABLE can be implemented as UNDEFINED.

In an implementation that supports virtualization, the Non-secure execution of UNPREDICTABLE instructions at a lower level of privilege can be trapped to the hypervisor, provided that at least one instruction that is not UNPREDICTABLE can be trapped to the hypervisor if executed at that lower level of privilege.

For an Arm trace macrocell, UNPREDICTABLE means that the behavior of the macrocell cannot be relied on. Such conditions have not been validated. When applied to the programming of an event resource, only the output of that event resource is UNPREDICTABLE. UNPREDICTABLE behavior can affect the behavior of the entire system, because the trace macrocell can cause the PE to enter Debug state, and external outputs can be used for other purposes.

When UNPREDICTABLE appears in body text, it is always in SMALL CAPITALS.

VA

See Virtual Address (VA).

Virtual Address (VA)

An address that is used in an instruction as a data or instruction address. The PC, LR, and SP always hold virtual addresses. For a Protected Memory System Architecture (PMSA) implementation, the virtual address is identical to the physical address.

See also Intermediate Physical Address (IPA) and Physical Address (PA).

Watch

In an Arm debugger, a watch is a variable or expression that the debugger must display at every step or breakpoint so that the user can see how its value changes.

Watchpoint

A debug event that is triggered by an access to memory, which is specified in terms of the address of the location in memory being accessed.

In DS-5, this is a hardware breakpoint.

See also Breakpoint.

WI

Writes Ignored. In a register that software can write to, a WI attribute applied to a bit or field indicates that the bit or field ignores the value written by software and retains the value it had before that write.

See also RAO/WI, RAZ/WI, RES0, RES1.

Word

A 32-bit data item. Words are normally word-aligned in Arm systems.

Word-aligned

A data item having a memory address that is divisible by four.