Embedded Trace Macrocell™

ETMv1.0 to ETMv3.4

Architecture Specification

ARM®

ARM IHI 0014O

Embedded Trace Macrocell
Architecture Specification

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Confidentiality	Change
30 March 1999	A	Limited Confidential	First release for ETMv1.0 and ETMv1.1.
12 July 1999	B	Limited Confidential	Errata 01 corrections incorporated for ETMv1.1 and ETMv1.0.
03 December 1999	C	Non-Confidential	Protocol enhancements and modified trace port connector pinout added. ETMv1.0 and ETMv1.1 release.
18 May 2000	D	Confidential	Protocol version 2 enhancements added. ETMv1.2 release.
06 September 2000	E	Non-Confidential	Minor corrections to Issue D incorporated. ETMv1.2 release.
15 January 2001	F	Confidential	Protocol version 3 enhancements added to support the tracing of Java instructions. ETMv1.3 release.
08 May 2001	G	Non-Confidential	Description of protocol versions and variants included. Released in conjunction with fixes to errata in ETMv1.2 and ETMv1.3.
25 July 2001	H	Non-Confidential	Description of ETMv2.0 enhancements included.
17 December 2002	I	Non-Confidential	Incorporation of ETMv2.1, ETMv3.0, and ETMv3.1 architectures.
16 July 2004	J	Non-Confidential	Incorporation of ETMv3.2 architecture.
17 March 2005	K	Non-Confidential	Minor corrections and updates.
04 November 2005	L	Confidential	Incorporates ETMv3.3 architecture, re-organizes descriptions of address comparators, and has minor enhancements elsewhere.
14 December 2005	M	Confidential	Final draft of ETMv3.4 issue.
08 February 2006	N	Non-Confidential	Non-confidential release of ETMv3.4 issue. No change to content.
20 July 2007	O	Non-Confidential	Various enhancements, updates and corrections, incorporating all errata to Issue N. Updated Implementer codes list. Added summary of IMPLEMENTATION DEFINED ETM features to Appendix A.

Proprietary Notice

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith.

1. Subject to the provisions set out below, ARM hereby grants to you a perpetual, non-exclusive, nontransferable, royalty free, worldwide licence to use this ARM Embedded Trace Macrocell Architecture Specification for the purposes of developing; (i) software applications or operating systems which are targeted to run on microprocessor cores distributed under licence from ARM; (ii) tools which are designed to develop software programs which are targeted to run on microprocessor cores distributed under licence from ARM; (iii) integrated circuits which incorporate a microprocessor core manufactured under licence from ARM.

2. Except as expressly licensed in Clause 1 you acquire no right, title or interest in the ARM Embedded Trace Macrocell Architecture Specification, or any Intellectual Property therein. In no event shall the licences granted in Clause 1, be construed as granting you expressly or by implication, estoppel or otherwise, licences to any ARM technology other than the ARM Embedded Trace Macrocell Architecture Specification. The licence grant in Clause 1 expressly excludes any rights for you to use or take into use any ARM patents. No right is granted to you under the provisions of Clause 1 to; (i) use the ARM Embedded Trace Macrocell Architecture Specification for the purposes of developing or having developed microprocessor cores or models thereof which are compatible in whole or part with either or both the instructions or programmer's models described in this ARM Embedded Trace Macrocell Architecture Specification; or (ii) develop or have developed models of any microprocessor cores designed by or for ARM; or (iii) distribute in whole or in part this ARM Embedded Trace Macrocell Architecture Specification to third parties without the express written permission of ARM; or (iv) translate or have translated this ARM Embedded Trace Macrocell Architecture Specification into any other languages.

3.THE ARM EMBEDDED TRACE MACROCELL ARCHITECTURE SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE.

4. No licence, express, implied or otherwise, is granted to LICENSEE, under the provisions of Clause 1, to use the ARM tradename, in connection with the use of the ARM Embedded Trace Macrocell Architecture Specification or any products based thereon. Nothing in Clause 1 shall be construed as authority for you to make any representations on behalf of ARM in respect of the ARM Embedded Trace Macrocell Architecture Specification or any products based thereon.

110 Fulbourn Road Cambridge, England CB1 9NJ

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
Embedded Trace Macrocell Architecture Specification

Glossary

List of Tables
Embedded Trace Macrocell Architecture Specification

List of Figures
Embedded Trace Macrocell Architecture Specification

Preface

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

About this specification

This specification describes the ARM *Embedded Trace Macrocell* (ETM) architecture. All ETMs conform to a version of this architecture that covers the following areas of functionality:

- The Programmer's Model, described in Chapter 2 and Chapter 3
- The Trace Port Protocol, described in Chapter 4, Chapter 5, Chapter 6, and Chapter 7
- The Physical Interface, described in Chapter 8.

Some parts of the ETM architecture are IMPLEMENTATION DEFINED. For details consult the relevant ETM *Technical Reference Manual* (TRM). See *Further reading* on page xxv.

Product revision status

The r*n*p*n* identifier indicates the revision status of some of the products described or referenced in this manual, where:

r*n* Identifies the major revision of the product.

p*n* Identifies the minor revision or modification status of the product.

Intended audience

This specification has been written for the following target audiences:

- Designers of development tools providing support for ETM functionality. All chapters in this specification are of interest to these users.
- Advanced users of development tools providing support for ETM functionality. Chapter 2 and Chapter 3 are particularly relevant to these users.
- Designers of Trace Port Analyzers. Chapter 8 and *Decoding required by trace capture devices* on page 4-7 are particularly relevant to these users.
- Designers of an ARM core based product that includes an ETM trace port. Chapter 8 is particularly relevant to these users.
- Engineers who want to specify, design or implement an Embedded Trace Macrocell to the ARM ETM Architecture.

Hardware engineers who want to incorporate an ARM ETM into their design must consult the relevant ETM *Technical Reference Manual* listed in *Further reading* on page xxv. ARM Limited recommends that all users of this specification have experience of the ARM architecture.

Using this specification

This specification is organized into the following chapters:

Chapter 1 ***Introduction***

Read this chapter for an introduction to the ETM.

Chapter 2 ***Controlling Tracing***

Read this chapter for information about how to control a trace run.

Chapter 3 ***Programmer's Model***

Read this chapter for information about the programmer's model for the ETM, including descriptions of the ETM registers. The chapter also describes the use of ETM Event Resources, and gives examples of the configuration of the **ViewData** and **TraceEnable** functions, that are used to filter the tracing.

Chapter 4 ***Signal Protocol Overview***

Read this chapter for a general description of the different types of information output by the ETM.

Chapter 5 ***ETMv1 Signal Protocol***

Read this chapter for information about the trace port protocol for ETMv1.

Chapter 6 ***ETMv2 Signal Protocol***

Read this chapter for information about the trace port protocol for ETMv2.

Chapter 7 ***ETMv3 Signal Protocol***

Read this chapter for information about the trace port protocol for ETMv3.

Chapter 8 ***Trace Port Physical Interface***

Read this chapter for information about the hardware interface requirements for the ETM.

Chapter 9 ***Tracing Dynamically Loaded Images***

Read this chapter for information about issues relating to tracing dynamically-loaded code issues. The chapter also tells you about the use of Context IDs.

Appendix A ***ETM Quick Reference information***

Read this appendix when you are configuring ETM events and require only quick-reference information.

Appendix B ***Architecture Version Information***

Read this appendix for a summary of information about the different architecture versions.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Numbering*.

Typographical

This manual uses the following typographical conventions:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes ARM processor signal names. Also used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
< **and** >	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. The replaceable terms appear in normal font in running text. For example: • `MRC p15, 0 <Rd>, <CRn>, <CRm>, <Opcode_2>` • The Opcode_2 value selects which register is accessed.

Numbering

A numbering convention used in this specification is:

<size in bits>'<base><number>

This is a Verilog method of abbreviating constant numbers. For example:

- 'h7B4 is an unsized hexadecimal value.
- 8'd9 is an 8-bit wide decimal value of 9.
- 8'h3F is an 8-bit wide hexadecimal value of `0x3F`. This is equivalent to b0011 1111.
- 8'b1111 is an 8-bit wide binary value of b0000 1111.

Further reading

This section shows publications by ARM Limited, and by third parties.

ARM Limited periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets, addenda, and the Frequently Asked Questions list.

The ETM documentation suite

This architecture specification is part of the ETM documentation suite and contains information that is relevant to all implementations of the ETM. The other manuals in the ETM documentation suite are IMPLEMENTATION SPECIFIC. Usually, the IMPLEMENTATION SPECIFIC documentation for an ETM comprises:

- An *ETM Technical Reference Manual*, describing the IMPLEMENTATION DEFINED behavior of the ETM.
- An *ETM Integration Manual*, describing how to integrate the ETM into an ASIC.
- An *ETM Configuration and Sign-off Guide*, that gives information about implementing the ETM. A Configuration and Sign-off Guide is complemented by reference methodology documentation from an EDA tools vendor that describes the implementation flow.

  For some ETMs, an *ETM Implementation Guide* is supplied instead of a Configuration and Sign-off Guide. An Implementation Guide includes a description of the implementation flow.

Some exceptions to the usual document set are:

- For the Cortex™-A8 processor, the ETM™ is tightly integrated with the processor core, and the ETM-A8 is described in the *Cortex™-A8 Technical Reference Manual* (ARM DDI 0344).
- For the Cortex™-M3 processor, the CoreSight™ ETM™-M3 is described in the *Cortex™-M3 Technical Reference Manual* (ARM DDI 0337).
- There is no IMPLEMENTATION SPECIFIC documentation for some ETMv1 implementations. See *Supported standard configurations in ETMv1* on page 2-73 for more information about these implementations.

See the processor TRM for information about its ETM interface, and any IMPLEMENTATION SPECIFIC ETM documentation.

See *Other ARM publications* for details of CoreSight Design Kit documents.

Other ARM publications

A CoreSight Design Kit includes the following documents:

- *CoreSight Architecture Specification* (ARM IHI 0029)
- *CoreSight Technology System Design Guide* (ARM DGI 0012)
- *CoreSight Components Technical Reference Manual* (ARM DDI 0314)
- *CoreSight Components Implementation Guide* (ARM DII 0143)
- the appropriate *CoreSight Design Kit Integration Manual*

- *AMBA AHB Trace (HTM) Technical Reference Manual* (ARM DDI 0328)
- *RealView ICE User Guide* (ARM DUI 0155).

The following documents include other relevant information:

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)

  Note

  This document replaces the earlier *ARM Architecture Reference Manual* (ARM DDI 0100) and its supplements.

- *ARM Debug Interface v5 Architecture Specification* (ARM IHI 0031)
- *ARMv7-M Architecture Reference Manual* (ARM DDI 0403).

Feedback

ARM Limited welcomes feedback both on the Embedded Trace Macrocell, and on its documentation.

Feedback on the Embedded Trace Macrocell

If you have any comments or suggestions about this product, contact your supplier giving:

- the product name
- a concise explanation of your comments.

Feedback on this specification

If you have any comments on this specification, send email to `errata@arm.com` giving:

- the document title
- the document number
- the page number(s) to which your comments refer
- a concise explanation of your comments.

ARM Limited also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter contains a brief introduction to the *Embedded Trace Macrocell* (ETM). It contains the following sections:

1.1 About Embedded Trace Macrocells

An *Embedded Trace Macrocell* (ETM) is a real-time trace module providing instruction and data tracing of a processor. An ETM is an integral part of an ARM RealView® debug solution.

1.1.1 Structure of an ETM

The main features of an ETM are:

Trace generation Outputs information that help you to understand the operation of the processor. The trace protocol provides a real-time trace capability for processor cores that are deeply embedded in much larger ASIC designs. (You cannot determine how the processor core is operating by observing the pins of the ASIC, because the ASIC typically includes significant amounts of on-chip memory.)

Triggering and filtering facilities

An extensible specification enables you to control tracing by specifying the exact set of triggering and filtering resources required for a particular application. Resources include address comparators and data value comparators, counters, and sequencers.

1.1.2 The debug environment

A software debugger provides the user interface to the ETM. The debugger can configure all the ETM facilities, such as the trace port, typically using a JTAG interface. The debugger also displays the trace information that has been captured.

The ETM compresses the trace information and either:

- Exports it through a trace port. An external *Trace Port Analyzer* (TPA) captures the trace information as shown in Figure 1-1 on page 1-3.
- Writes it directly to an on-chip *Embedded Trace Buffer* (ETB). The trace is read out at low speed using the JTAG interface when the trace capture is complete as shown in Figure 1-2 on page 1-4.

When the trace has been captured the debugger extracts the information from the TPA or ETB and decompresses it to provide a full disassembly, with symbols, of the code that was executed. The debugger can also link this back to the original high-level source code, providing you with a visualization of how the code was executed on the target system.

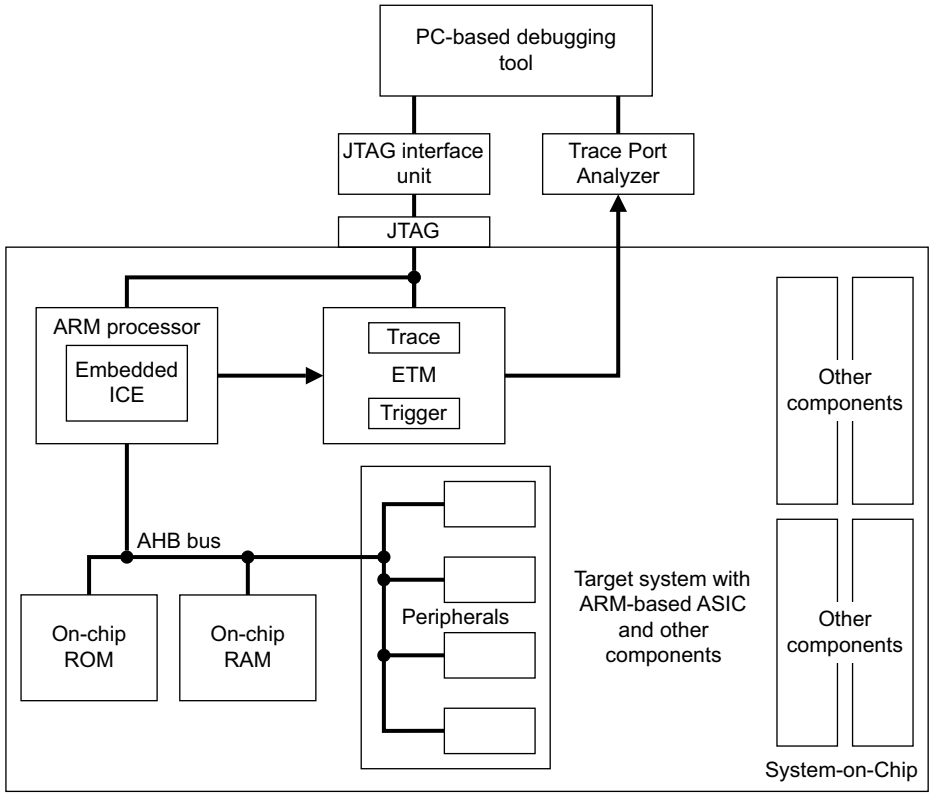


Figure 1-1 Example debugging environment with TPA

Figure 1-2 on page 1-4 shows how the ETM is used in a complete debug environment. The JTAG interface is also used for other debugging functions, such as downloading code and single-stepping through the program.

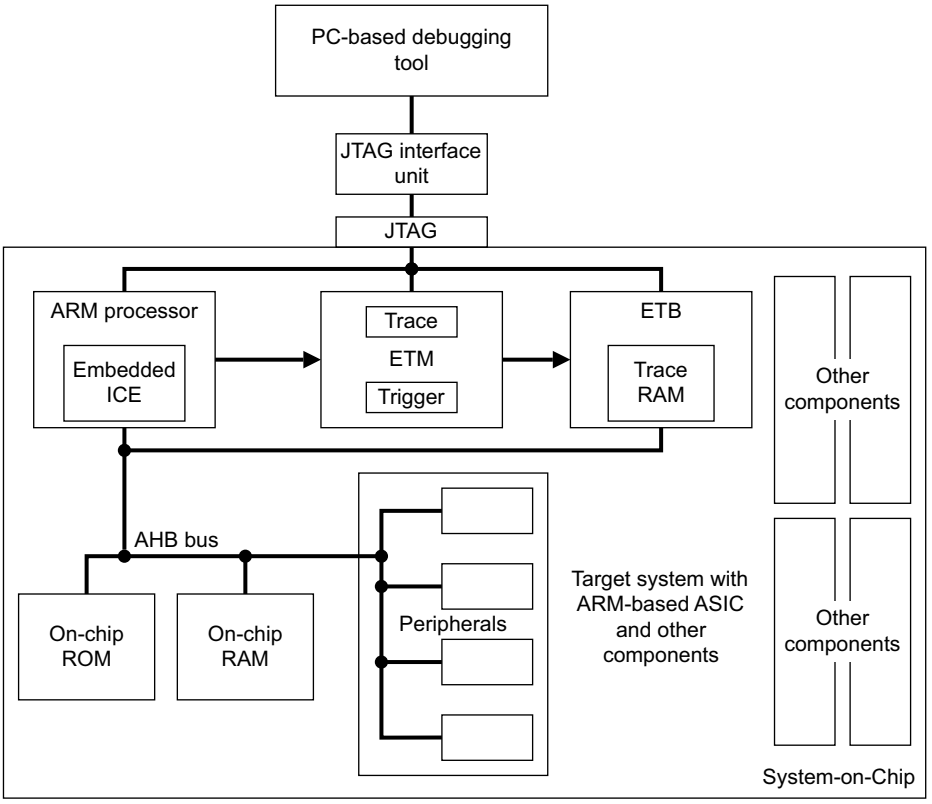


Figure 1-2 Example debugging environment with ETB

1.1.3 Thumb and Java support

Both ARM and Thumb® instructions can be fully traced. In cores supporting Jazelle®, Java bytecodes executed while in Jazelle state can also be traced. The trace contains information about when the ARM core switches between states.

1.1.4 Trace compression

The trace produced by the ETM is compressed to reduce the number of additional pins required on the ASIC, or to reduce the amount of memory required by the ETB.

The trace is compressed using the following techniques:

- Address information is only output when the processor branches to a location that cannot be directly inferred from the source code.
- When an address is output, high-order bits that have not changed are not output.

- Instruction and data trace can be independently filtered.
- For data accesses you can choose to output:
  - — only the data
  - — only the address
  - — both.
- Trace is only output when the full width of the trace port can be used.
- Some ETMs perform leading zero compression on data values.
- Some ETMs run-length encode instruction trace events.

Note

For the debugger to be able to decode the trace, you must supply a static image of the code being executed. Self-modifying code cannot be traced because of this restriction.

1.2 ETM versions and variants

The ETM is subject to continuous improvement in conjunction with the development of ARM processors. The history of ETM versions and variants is shown in Table 1-1. When Table 1-1 does not list different revisions of an ETM the information in the table applies to all revisions.

Table 1-1 ETM versions and variants

ETM name	Protocol number	Architecture version
ETM7 Rev 0	1	ETMv1.1
ETM7 Rev 1	2	ETMv1.2
ETM7 Rev 1a	4	ETMv1.2
ETM9 Rev 0	0	ETMv1.0
ETM9 Rev 0a	1	ETMv1.1
ETM9 Rev 1	2	ETMv1.2
ETM9 Rev 2	3	ETMv1.3
ETM9 Rev 2a	5	ETMv1.3
ETM9 r2p2	7	ETMv1.3
CoreSight ETM9	Not applicable	ETMv3.2
ETM10	Not applicable	ETMv2.0
ETM10RV	Not applicable	ETMv3.0
ETM11	Not applicable	ETMv3.1
CoreSight ETM11	Not applicable	ETMv3.2
CoreSight ETM-R4	Not applicable	ETMv3.3
CoreSight ETM-A8	Not applicable	ETMv3.3
CoreSight ETM-M3	Not applicable	ETMv3.4

- In ETMv1, protocol version numbers are used to distinguish between versions of the architecture, and to distinguish between different implementations of the ETM. Each protocol version corresponds to an architecture version.
- Protocol version numbers are not used in ETMv2 and later, because with these ETM versions you can read the architecture version from the ETM ID register, see *ETM ID Register, ETMv2.0 and later* on page 3-71.

Note

- ETM protocol version 4 is equivalent to ETM protocol version 2.
- ETM protocol versions 5 and 7 are equivalent to ETM protocol version 3.

These ETMs do not report an architecture version. The different protocol versions enable software tools to implement workarounds for errata in earlier versions.

Chapter 2
Controlling Tracing

This chapter describes the control mechanisms and configuration options provided with the ETM.

Note

Implementing the tracing controls requires an understanding of Chapter 3 *Programmer's Model*. Make sure you have a good general understanding of Chapter 2 and Chapter 3 before implementing specific controls.

This chapter contains the following sections:

- *Supported configurations from ETMv2* on page 2-75
- *Behavior when non-invasive debug is disabled* on page 2-76.

2.1 About controlling tracing

You control tracing in two ways:

Triggering Triggering controls when the collection of the trace data occurs. Setting a trigger enables you to focus trace collection around your region of interest.

Filtering Filtering controls the type of trace information that is collected. It is important to optimize usage of the trace port bandwidth, especially when a narrow trace port is used. Filtering the trace serves two purposes:

- It prevents overflow of the internal FIFO by minimizing the number of data transfers traced. This is especially important when the FIFO is small or the trace port is narrow.
- It limits the amount of trace stored by the *Trace capture device* (TCD), for example a TPA or an on-chip trace buffer. This enables more useful information to be stored around the trigger.

You can filter the instruction trace or the data trace as follows:

- Filter the instruction trace by enabling and disabling trace generation. This is the **TraceEnable** function.
- Filter the data trace by indicating the specific data accesses that must be traced. This is the **ViewData** function.

You use the *ETM event logic* to configure the *ETM event resources* that are used for triggering and filtering. Resources *match* for one or more cycles when the condition they have been programmed to check for occurs. Resources are selected to control different aspects of ETM operation, for example:

- when to trigger
- when to perform instruction tracing
- when to perform data tracing.

In the most simple case, where a resource is selected to enable tracing, tracing is performed whenever the resource matches.

In most cases you can define an ETM *event*, where you select a boolean function and two resources to define a condition.

For more information, see *ETM event resources* on page 2-4 and *ETM event logic* on page 2-15.

2.2 ETM event resources

The possible ETM event resource types are:

- Address comparators. These can operate on both instruction and data access addresses.
- Data value comparators.
- Context ID comparators, in ETMv2.0 and later.
- Memory map decoders.
- EmbeddedICE™ module watchpoint comparators.
- Counters.
- A three-state sequencer.
- External inputs.
- Extended external inputs, in ETMv3.1 and later.
- Trace start/stop, in ETMv1.2 and later.
- From ETMv3.3, Instrumentation resources, controlled by software instructions.

Different ETMs implement different selections of resources, and different numbers of some resources such as address comparators. You can read the resource configuration of a particular ETM using its programming interface.

For the number of available resources in standard ETM configurations, see the following:

For ETM7 and ETM9 (ETMv1)

See *ETM7 supported configurations* on page 2-73.

For other ETMs See the appropriate *Technical Reference Manual*.

Resource identification on page 3-108 describes the exact bit encoding for each resource type. The bit encodings enable you to uniquely identify a particular resource. For each resource type, resources are numbered from 1 to n. Each of the resource types is described in later sections of this chapter.

The resources are classified and described as follows:

- *Memory access resources* on page 2-5

  Note

  This section introduces all of the memory access resources. The principal memory access resources are the comparators. Because of the range of comparator options, the detailed description of the behavior and use of the comparators is given in:

  — *Address comparators* on page 2-36

  — *Operation of data value comparators* on page 2-56.

- *Instrumentation resources, ETMv3.3 and later* on page 2-9
- *Derived resources* on page 2-10
- *External inputs* on page 2-11.

For information about programming the registers that control these resources, see Chapter 3 *Programmer's Model*.

2.2.1 Memory access resources

There are five types of memory access resource:

- single address comparators, used with or without data value comparators
- address range comparators, used with or without data value comparators
- Context ID comparators
- EmbeddedICE module watchpoint comparators
- device-specific memory map decoders.

Single address comparators

Address comparators compare either the instruction address or the data address against a user-programmed value. There are between zero and 16 single address comparators, but there must be an even number of them. Each pair can have an associated bit-masked data value comparator, see *Data value comparators* on page 2-7.

Each comparator has several configuration bits to determine the match conditions. The available options are:

- instruction fetch
- instruction execute (irrespective of condition code passed or failed)
- instruction executed and condition code test passed (ETMv1.2 or later)
- instruction executed and condition code test failed (ETMv1.2 or later)
- data load or store
- data load only
- data store only.

Note

From ETMv3.3, an ETM implementation might not support data address comparisons. See *No data address comparator option, ETMv3.3 and later* on page 2-7 for more information.

Instruction execute means that the instruction at that address has reached the Execute stage of the pipeline and includes instructions that fail their condition codes. ETMv1.2 introduced the facility to control trace using the result of the condition code test whenever an instruction is executed.

The address comparators are not bit-masked. This means that you cannot use a single comparator to generate a binary range (starts at an offset of 0 and ends at an offset of 2^n). If a range is required, you must use one of the following:

- A pair of comparators configured for address range comparison, see *Address range comparators* on page 2-6.
- The IMPLEMENTATION SPECIFIC memory map decoders described in *Memory map decoder (MMD)* on page 2-8.
- The ARM EmbeddedICE module. This is only available in cores supporting the **RANGEOUT** signal, and enables you to carry out full masked address comparisons using a single EmbeddedICE comparator.

Typically, a single address comparator only *matches* for a single cycle, regardless of its configuration. This ensures that counter and sequencer transitions occur cleanly, without the possibility of multiple counts or transitions from, for example, memory wait states. For more information see *Address comparators* on page 2-36.

The 32-bit address from the ARM core, that might have bits [1:0] masked depending on whether these bits can be safely predicted, is compared with the address value.

In ETMv2.0 and later, you can make address comparators conditional on a Context ID comparator matching. Every Context ID comparator is available to every address comparator for use in this way.

In ETMv3.2 and later, with a processor that supports the Security Extensions, you can configure the comparator to match only in the Secure state, only in the Non-secure state, or in both Secure and Non-secure states.

If you use a comparator with the Exact match bit set to 1 in the programming of **TraceEnable** or **ViewData**, tracing is Imprecise. See *Exact matching, ETMv2.0 and later* on page 2-44 for more information.

Address range comparators

The single address comparators are arranged in pairs to form an address range resource. An address range comparator is programmed as follows:

- the first comparator is programmed with the range start address
- the second comparator is programmed with the range end address.
- the second comparator value *must* be greater than the first comparator value.

The resource matches if the address is in the following range:

(address >= range start address) AND (address < range end address)

An address range comparator can operate on instruction or data addresses.

UNPREDICTABLE behavior occurs if the two address comparators are not configured in the same way. For example, behavior is UNPREDICTABLE if one comparator is configured to match on instruction fetch and the other is configured to match on instruction execute.

Features such as *out of range* are dealt with using Boolean operations available in the event logic (see *ETM event logic* on page 2-15), and by exclude regions in **TraceEnable** and **ViewData** (see *Trace filtering* on page 2-18).

Note

From ETMv3.3, an ETM implementation might not support data address comparisons. See *No data address comparator option, ETMv3.3 and later* on page 2-7 for more information.

Typically, an address range resource matches for a continuous number of cycles. The match first occurs when the address is in the correct range. The comparator remains in this state until a new address outside the matching range is generated. Any access outside the matching range causes the comparator to go inactive. For more information see *Address comparators* on page 2-36.

No data address comparator option, ETMv3.3 and later

From ETMv3.3, it is IMPLEMENTATION DEFINED whether an ETM macrocell supports data address comparisons. Support for data address comparisons is indicated by bit [12] of the Configuration Code Extension Register, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76. This bit is set to 1 if data address comparisons are *not* supported. This means that, from ETMv3.1, this bit can be checked to see if data address comparisons are supported. If reading the Configuration Code Extension Register returns bit [12] = 0 then data address comparisons are supported.

If an implementation does not support data address comparisons:

- Setting the Access type field, bits [2:0], of an Address Access Type register to a data operation causes UNPREDICTABLE behavior. See *Address Access Type Registers* on page 3-51.
- Data value comparators are not supported:
  - The Number of data value comparators field, bits [7:4], of the ETM Configuration Code Register returns a value of b0000. See *ETM Configuration Code Register* on page 3-29.
  - The Data Comparator Value and Data Comparator Mask Registers are not implemented and Read-As-Zero. These are registers `0x030` to `0x04F`, at addresses `0x0C0-0x13C` in a memory-mapped implementation.

  Note

  The ETM architecture permits an implementation to support data address comparisons even if it does not implement any data value comparators.

Data value comparators

Each pair of address comparators can be associated with a specific data value comparator. An address comparator that has an associated data value comparator also has a data value comparison enable field.

A data value comparator monitors the data bus only when a load or store operation occurs.

Data value comparisons are not supported for address comparators configured for instruction addresses. UNPREDICTABLE behavior results if a data value comparison is enabled for an instruction Fetch or Execute comparator.

The number of data value comparators is IMPLEMENTATION DEFINED, between zero and eight address comparator pairs can have associated data value comparators. *Data value comparator registers* on page 3-54 describes exactly how data value comparators must be allocated to the address comparators.

A data value comparator has both a value register and a mask register, so it is possible to compare only certain bits of the pre-programmed value against the data bus.

For information on data value comparisons during aborts, see *Exact matching, ETMv2.0 and later* on page 2-44.

An address comparison, or address range comparison, is qualified by the data value comparison.

ETMv1.2 and later also supports matching if the data value comparison does not match.

The behavior of the comparators for data value comparisons, for all ETM versions, is described in *Operation of data value comparators* on page 2-56.

Context ID comparators

In ETMv2.0 and later, you can use Context ID comparators for trace filtering. Each has a 32-bit value register, and one mask register is shared between all Context ID comparators. Context ID comparators can be used directly by address comparators, or selected as part of an event. There are between zero and three Context ID comparators. For more information, see *Context ID comparator registers, ETMv2.0 and later* on page 3-66.

EmbeddedICE watchpoint comparators

You can use the EmbeddedICE module watchpoint comparators as additional trigger resources.

Note

- This resource is not available in all implementations. For example, it is not available on ETMs for the ARM10 and ARM11 product families, because these processors do not have the **RANGEOUT** output.
- EmbeddedICE comparators are architecturally defined in all versions of the ETM architecture, even though some ETMs do not implement them.

In ETMv3.3 and earlier, if an ETM implements EmbeddedICE watchpoint comparator inputs then it provides two inputs. These correspond to the **RANGEOUT[1:0]** signals, that are available from ARM7 and ARM9 processors only.

From ETMv3.4, the number of EmbeddedICE watchpoint comparator inputs is IMPLEMENTATION DEFINED, in the range 0 to 8. For more information about the implementation of EmbeddedICE watchpoint comparator inputs in ETMv3.4 and later, see *Behavior of EmbeddedICE inputs, from ETMv3.4* on page 7-62.

Memory map decoder (MMD)

Some system designs contain an address decoder that divides the memory space statically into different regions. For example, the MMD might divide the memory into separate regions for RAM, ROM, and peripherals. For these systems, you can customize the ETM with external logic for a particular application, to enable low-cost decoding of address regions.

Note

- The MMD is not available in all implementations. When an ETM does not implement an MMD you can use the **EXTIN** inputs as an imprecise tracing alternative. For more information, see *External inputs* on page 2-11.
- MMDs are architecturally defined in all versions of the ETM architecture, even though some ETMs do not implement them.

As with the full address comparator resources, up to 16 MMDs are supported. An additional control register is provided to enable you to configure statically the memory decode map to be used.

The interface to the external *Memory Map Decode* (MMD) logic is IMPLEMENTATION SPECIFIC. See the appropriate *Technical Reference Manual* for details.

The MMD behaves in a similar way to the address range comparators, except that the MMD always uses the full 32-bit address, and you must implement any masking of addresses externally.

The instruction and data addresses on which the decoder operates are *registered*. The MMD becomes active when the address first matches, and remains active until the comparison fails.

If precise memory comparisons are required, you must ensure that the match is active only for a single cycle. This ensures that the behavior is identical to the full address comparators described earlier in this section, see *Single address comparators* on page 2-5.

The comparisons are likely to be simple bit-masked comparisons. This behavior is similar to that of a typical memory decoder present in the ASIC memory system. The hardware required to implement this is minimal.

Memory map decoding is only possible as Fetch stage comparisons. No attempt is made to produce or select Execute stage versions. You are likely to use these resources primarily to decode the peripheral address map, and possibly to subdivide the ARM processor code and data space. **ViewData** is precise when based on memory map data address comparisons resources, see *ViewData and filtering the data trace* on page 2-26.

For a Harvard ARM processor (one with separate instruction and data memory interfaces), the designer of the memory map decoder must choose one of the following decode strategies:

- Apply the same decode map to both the instruction and data address buses.
- Decode the instruction and data address buses separately.

  This is preferable because, for example, instructions are never fetched from the peripheral memory space.

The exact MMD map is IMPLEMENTATION SPECIFIC. At any one time there are only 16 memory map resources available. Many ASICs have a more complex memory map than this, so the ASIC Control Register, `0x03`, can be used to configure the MMD logic. For example, you can use this register to switch between instruction address and data address decoding. You are unlikely to use more than one or two bits of this register for this purpose.

2.2.2 Instrumentation resources, ETMv3.3 and later

From ETMv3.3, an ETM can include up to four Instrumentation resources. These resources can be set, cleared or pulsed by low-overhead ARM or Thumb instructions. In this context, set means the resource is active, corresponding to a logic 1 output, and clear means a resource is inactive, corresponding to a logic 0 output.

When a resource is pulsed it is set for the current cycle and cleared from the following cycle.

Instrumentation resources are described in more detail in *Instrumentation resources, from ETMv3.3* on page 2-63.

2.2.3 Derived resources

There are three types of derived resource. These are resources that are not directly related to memory accesses, and are:

- 16-bit counters, see *Counters*
- the sequencer (state machine), see *Sequencer*
- the trace start/stop resource, see *Trace start/stop resource* on page 2-11.

There are between zero and four counters, and there is zero or one sequencer.

The trace start/stop resource is always present in ETMv1.2, and is optional in ETMv2.0 and later.

Counters

Each counter is clocked by the system clock, even on cycles when the core is stalled. The counter decrements when its counter enable is active. Operation of the counter is controlled by a count enable event. You configure the counter to decrement, at full system clock speed, by setting the count enable event to TRUE.

The counters are 16-bit, so they can count from 1 to 65535 events.

Each counter has a programmable reload register. You can define an event that causes the counter to be reloaded from this register. This reload event takes priority over the count enable event.

When the counter reaches zero it remains at zero and the resource becomes active. It remains active until the counter is reloaded.

You can read and write the counter values using registers, see *Counter registers* on page 3-58 and *ETM Programming bit and associated state* on page 3-19.

Sequencer

A three-state sequencer is provided. If you require multiple-stage trigger schemes, the trigger event is usually based on a sequencer state. If you want the trigger to be derived from a single event, you do not require the sequencer.

Figure 2-1 on page 2-11 shows the sequencer state diagram. The sequencer has three possible next states (the current state and two others), and can change state on every clock cycle. The state transitions are controlled with events. For more information see *ETM event logic* on page 2-15.

On every cycle the sequencer does one of the following:

- remains in the current state
- moves to one of the other two states.

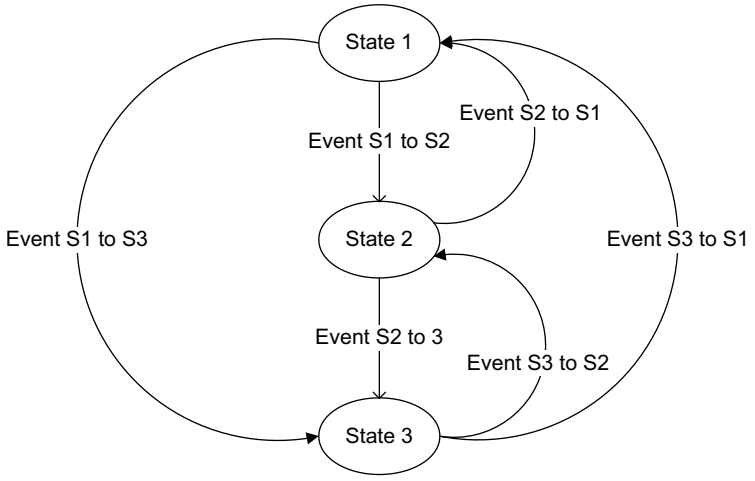


Figure 2-1 Sequencer state diagram

On an ETM reset, the sequencer goes to State 1. See *Reset behavior* on page 3-16.

Whatever the current state of the sequencer, there are two state transition events that change its state, and:

- if both of these state transition events are active the sequencer remains in its current state
- if neither of these state transition events is active the sequencer remains in its current state
- the behavior of the sequencer is UNPREDICTABLE if either of these state transition events has not been programmed.

You can read and write the current state of the sequencer, see *Sequencer registers* on page 3-62 and *ETM Programming bit and associated state* on page 3-19.

Trace start/stop resource

In ETMv2.0 and later, the trace start/stop resource is available and gives the current state of the trace start/stop block that is used in the generation of the **TraceEnable** filtering signal. This resource can be used even if it is not in use by **TraceEnable**. For more information see *The trace start/stop block* on page 2-23.

For more information about **TraceEnable**, see *Trace filtering* on page 2-18.

Before ETMv2.0, the trace stop/start block is available only for use by **TraceEnable**.

2.2.4 External inputs

There are five types of input resource:

- a hard-wired input, that is always TRUE
- external inputs
- extended external input selectors

- a Non-secure state resource
- a *prohibited region* resource.

Hard-wired input

External input 16 is hard-wired to provide a permanently active resource. This resource is always TRUE. It is used to permanently enable or disable events. For example, to enable tracing permanently the hard-wired input can be connected to the **TraceEnable** event.

FALSE is generated as the inverse of this resource, and can be used to disable an event.

External inputs

External inputs enable the ETM to respond to events outside the ETM, such as interrupts, or a trigger from another core. They are always Imprecise.

Note

The external inputs, **EXTIN**, are not related directly to memory accesses. Tracing is Imprecise if you use them in any way to enable or disable tracing. For more information about Imprecise Tracing, see *Imprecise TraceEnable events* on page 2-22.

Extended external input selectors

Extended external inputs are only defined in ETMv3.1 and later.

Extended external input selectors enable you to select inputs from a large number of extended external inputs. Figure 2-2 shows an example with four extended external inputs and two extended external input selectors.

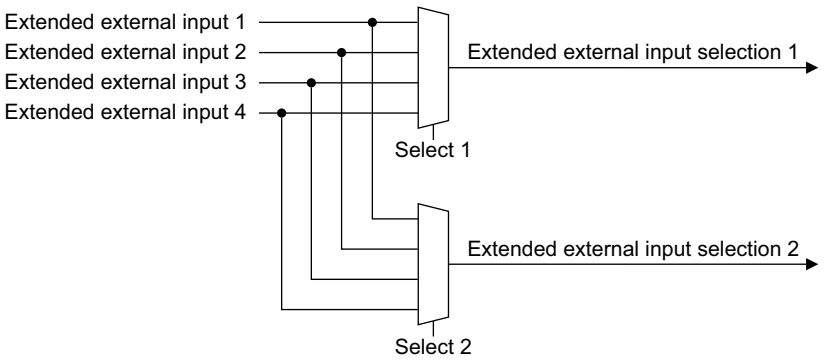


Figure 2-2 Extended external inputs example

Each extended external input selector is programmed to select from one of the extended external inputs, and is then available for use by any of the event blocks in the same way as any ordinary external input. An example of their use is to enable performance monitoring events to be used by the ETM.

Non-secure state resource

The Non-secure state resource enables events to be conditional on the security level.

Prohibited region resource

The prohibited region resource enables events to be disabled while trace is prohibited. This is particularly useful when using the ETM for performance monitoring, so that cycles spent in prohibited regions are not counted.

2.2.5 Example resource configuration

Figure 2-3 shows an example ETM resource configuration. It also shows the resource match signals that you can generate using the configuration.

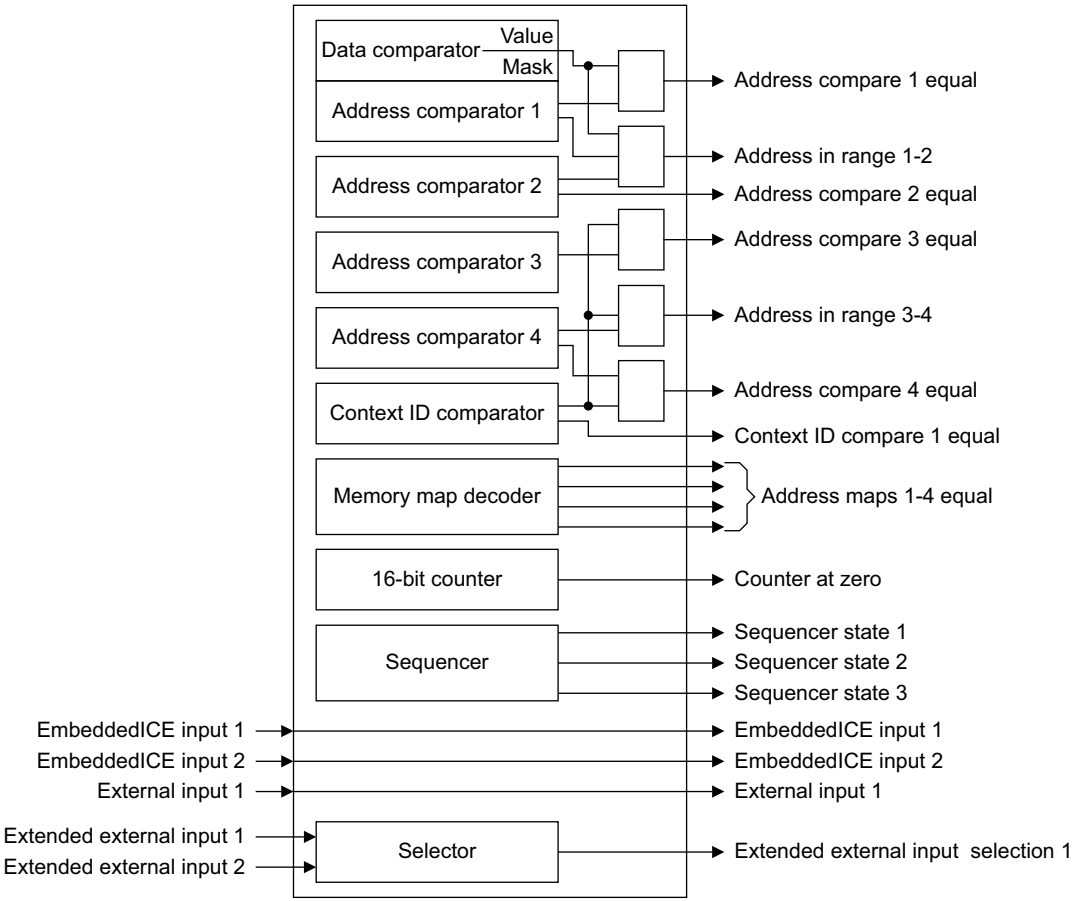


Figure 2-3 Example resource configuration

2.3 ETM event logic

In this document, the word *event* is used to indicate a Boolean combination of two ETM event resources. AND and OR operations are supported, and one or both of the two ETM event resource inputs can be negated. For more information about the ETM resources see *ETM event resources* on page 2-4.

Events control the basic transitions in the ETM, for example sequencer state changes. The combination of the two event resources enables many typical combinations to be easily expressed. For example, you can set a trigger to occur when a specified instruction executes a certain number of times.

See *Using ETM event resources* on page 3-108 for more information.

In some documentation:

- *complex event* is used to mean an ETM event
- *simple event* is used to mean an ETM event resource.

2.4 Triggering a trace run

You can use a trigger signal to specify when a trace run is to occur. You determine the trigger condition by using the event logic to configure the event resources. See *ETM event logic* on page 2-15 and *ETM event resources* on page 2-4.

The trigger event specifies the conditions that must be met to generate a trigger signal on the trace port. When the trigger event occurs, the trigger is output as soon as possible, and therefore might not be aligned with the rest of the trace. The trigger is output over the trace port using a code that can be readily understood by the *Trace capture device* (TCD), see *Decoding required by trace capture devices* on page 4-7.

The TCD uses the trigger in the following ways:

Trace after The trigger can indicate to the TCD that the trace information must be collected from the trigger point onwards. This is often called a *start trigger* and is used to find out what happens after a particular event, for example what happens after entering an interrupt service routine. Often, in addition, a small amount of trace data is collected before the trigger condition. This enables the decompression software to synchronize with the trace, ensuring that it can successfully decompress the code around the trigger point.

Trace before

The trigger can be used to stop collection of the trace. In this case the TCD acts like a large FIFO, so that it always contains the most recent trace information and the older information overflows out of the trace memory. The trigger indicates that the FIFO must stop, so the memory contains all the trace information before the trigger event. This is often called a *stop trigger* and is used to find out what caused a certain event, for example, to see what sequence of code was executed before entering an error handler routine. Often, in addition, a small amount of trace data is collected after the trigger condition.

Trace about

You can set the trigger between the start point and the stop point, so that the trace memory contains a defined number of events before the trigger point and a defined number of events after it. This is often called a *centre trigger*.

The generation of a trigger does not affect the tracing in any way.

A simple trigger can be based on memory access address or data matches, for example the execution of an instruction from a particular address. However, a more complicated set of trigger conditions is possible, such as executing a particular instruction several times, or a particular sequence of events occurring before the trigger is asserted.

In any trace run, only a single trigger can be generated by the ETM. However multiple triggers from different sources are permitted in a CoreSight system, see the CoreSight Architecture Specification for more information. When the trigger has been asserted you must set the ETM Programming bit of the ETM Control Register to 1, and then clear it to 0, before another run can begin. For more information see *ETM Control Register* on page 3-20.

2.5 External outputs

Some applications can use external outputs, for example to trigger a second ETM on-chip. Up to four external outputs are supported. Each output is controlled by an event, programmable in the same way as any ETM event.

When the ETM Programming bit is set to 1 (see *ETM Control Register* on page 3-20), the external outputs are forced LOW.

2.6 Trace filtering

You can use ETM events to disable and enable tracing as the trace run proceeds. Suspending and enabling tracing under certain conditions enables you to make best use of the storage capacity of the trace capture device, because capacity is not used up while tracing is suspended. For example, you might want to trace the execution of a particular function that occurs infrequently. If a particular trace run proceeds without interruption, you might observe only one or two occasions when the function executes. However, by disabling trace when that function is not executing you can ensure that the trace memory is filled only with relevant information. In this way, you can capture many more instances of the execution of the function, and the information extends over a greater period of time.

Whenever the trace is re-enabled, a full 32-bit address is output on the trace port, giving the address of the first instruction traced. If Context ID tracing is enabled, the current Context ID is also output. This provides full synchronization, so the decompressor is always able to start decompression from this point. Because of this overhead, we recommend that you do not disable the trace for only a small number of instructions. Enabling and disabling the trace over only a few instructions results in an increase in trace information passed through the trace port.

Note

If you attempt to disable tracing so briefly that the sequence required to re-enable tracing cannot be output, the ETM might continue tracing during the intended interval.

There are two principal ways to filter the trace:

- You can use the **TraceEnable** function to filter the instruction trace by enabling or disabling tracing dynamically. Figure 2-4 on page 2-21 shows the logic used for **TraceEnable** configuration.
- When **TraceEnable** is asserted, you can use the **ViewData** function to filter the data trace by enabling address and data tracing, for either regions of code or individual addresses. Figure 2-7 on page 2-27 shows the logic used for **ViewData** configuration.

You can also configure the **FIFOFULL** signal to optimize use of the FIFO. Figure 2-9 on page 2-31 shows the logic used for **FIFOFULL** configuration.

ETMv3.0 and later provides a data suppression mechanism. When data suppression is enabled and the amount of data in the FIFO exceeds the preset FIFO level, then no more data can be traced. The ETM stops tracing data rather than stopping the core. See *Data suppression* on page 2-33 for more information.

From ETMv3.3, it is IMPLEMENTATION DEFINED whether data suppression is supported. For more information, see *Checking whether data suppression is supported, ETMv3.3 and later* on page 2-34.

2.6.1 Definitions of when an ETM is tracing

In this specification:

- An ETM is said to be tracing when **TraceEnable** is active and no condition exists that prohibits tracing. Conditions that prohibit tracing include:
  - The processor is in Debug state.

— The processor is in a *Wait For Interrupt* (WFI) or *Wait For Event* (WFE) condition, see *Wait For Interrupt and Wait For Event* on page 4-26.

Note

Some ETMs might not prohibit tracing while the processor is in a WFI or WFE condition.

— the ETM FIFO has overflowed

— tracing is prohibited, see *Behavior while tracing is prohibited*.

- Tracing is said to have *restarted* whenever tracing becomes active. This includes tracing becoming active after the removal of a condition that prohibits tracing.

2.6.2 Behavior while tracing is prohibited

Some cores prohibit tracing at certain times, for example when executing some Secure code. These areas are called *prohibited regions*. When entering a prohibited region:

- A branch packet is generated. The address of the first instruction in the prohibited region is not given, and is traced as 0. If an exception caused entry to the prohibited region, the exception type is given in the branch address packet.
- If the security state changed on entry to the prohibited region the new Context ID is not traced.
- When the branch packet has been generated, tracing stops. No instructions in the prohibited region are traced.
- The security state of the processor is considered to be Secure. The Non-secure resource is LOW.
- Any out-of-order data corresponding to out-of-order placeholders that have already been traced are traced when the data value is returned.
- If cycle-accurate mode is enabled, the cycle counter continues to count. When tracing restarts, cycles spent in the prohibited region are included in the cycle count.
- Instruction address comparators do not match on any instruction in the prohibited region.
- Data address comparators ignore any data transfers corresponding to instruction in the prohibited region. If an address range comparator matches on the last data transfer before entering the prohibited region, it continues to match throughout the prohibited region. See *Resource identification* on page 3-108 for more information.
- Context ID comparators are disabled during the prohibited region.
- Instrumentation instructions executed when in a prohibited region have no effect. Entry to a prohibited region has no effect on the current state of any of the instrumentation resources.
- Other resources, such as the counters, sequencers, external outputs and trigger, behave as normal. These can be disabled using the trace prohibited resource, b110 1110. See *Resource identification* on page 3-108.

Normally, entry into a prohibited region from a non-prohibited region occurs only because of entering a Secure privileged mode. Tools must be aware of whether tracing is prohibited in Secure modes, so that they can detect prohibited regions.

An implementation can include signals that determine whether non-invasive debug is permitted in Secure modes. If these signals change dynamically to prohibit tracing of the current code, the ETM must behave as described above, with the following additional instructions:

- Whether the branch packet is generated is IMPLEMENTATION SPECIFIC.
- The entry point to the prohibited region might be imprecise. This means that tracing might continue after the assertion of the signals indicating that non-invasive debug is prohibited.

If the signals change dynamically and cause exit from a prohibited region, the restart of tracing might be imprecise. When the signal prohibiting non-invasive debug is deasserted there might be a delay before tracing restarts.

2.6.3 Programming strategies

Both **TraceEnable** and **ViewData** are controlled with events and resources. There are two possible programming strategies. You can do either or both of the following:

- use address comparators to set address ranges inside which trace is enabled
- set one or more addresses to turn on tracing and set one or more addresses to turn it off.

  This is controlled by the trace start/stop block. The state of the trace start/stop block is given by the trace start/stop resource. For more information see:

  — *The trace start/stop block* on page 2-23

  — *Trace Start/Stop Resource Control Register, ETMv1.2 and later* on page 3-37.

The advantage of using the trace start/stop resource is that any subroutine calls that are outside the function address range are also included in the trace.

2.6.4 TraceEnable and filtering the instruction trace

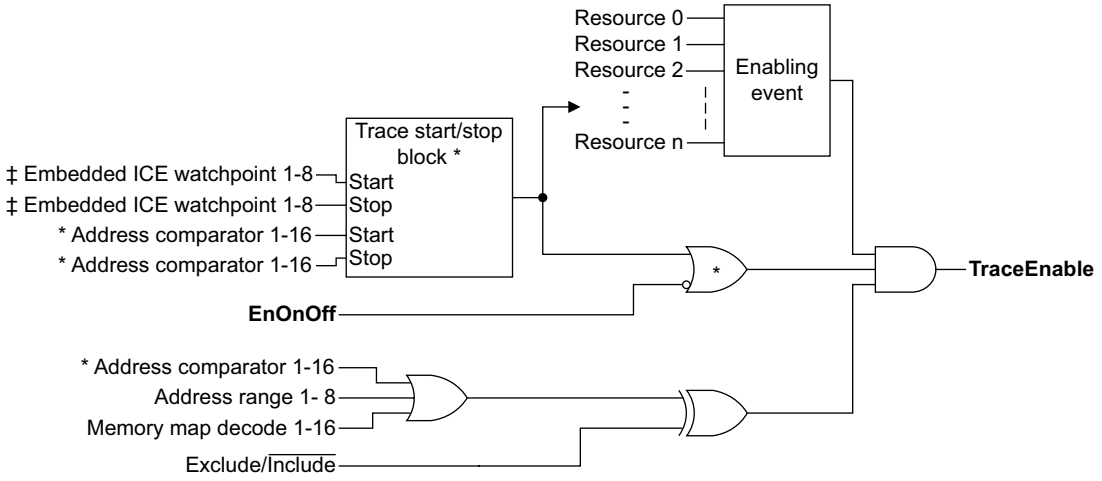
The trace port uses the **TraceEnable** signal to turn tracing on and off during a trace run.

TraceEnable is generated by:

- An enabling trace enable event. When the event is active tracing can occur.
- A trace start/stop address comparator, the trace start/stop block, ETMv1.2 and later.
- Either include or exclude address regions that are specified by:
  — the address range resources
  — memory map decode resources
  — individual addresses using the address comparator resources (ETMv1.2 and later).

Caution

If data address comparators are used in exclude regions, **TraceEnable** behavior is UNPREDICTABLE. Therefore, if you want to use data address comparisons to define trace exclude regions, use the data address comparators as ETM resources that can be used to define the **TraceEnable** enabling event.



Notes: * indicates components present only in ETMv1.2 and later.
‡ indicates optional components, only present in ETMv3.4 and later.
Most Resource inputs are not shown, see text for details.

Figure 2-4 TraceEnable configuration

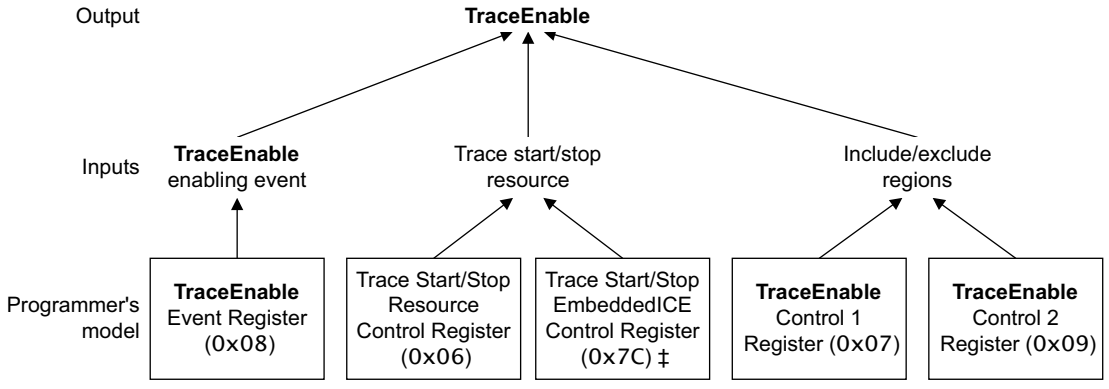
The operating mode, include or exclude, is defined statically for each trace run. Mixing include and exclude regions is not supported. Figure 2-4 shows the structure of the **TraceEnable** signal.

An exclude region is useful for excluding library code or particular functions that are known to generate a lot of data.

An include region enables all code inside a simple range to be traced, for example the FIQ and IRQ handlers.

You configure the **TraceEnable** logic by programming the **TraceEnable** registers as shown in Figure 2-5 on page 2-22. For more information, see *TraceEnable registers* on page 3-37.

From ETMv3.3, if the **TraceEnable** include/exclude function is used to exclude a specific instruction then TraceEnable remains low until the next instruction.



‡ Only present in ETMv3.4 and later.

Figure 2-5 Programming the TraceEnable logic

Data-controlled instruction tracing

You can control instruction tracing using data accesses. This means that you can trace a load or store instruction based on the data accesses that the instruction performs, rather than only on its instruction address. You can configure the access type by setting the access type value in the Address Access Type Register (see *Address comparator registers* on page 3-50).

Note

In ETMv1.x, the data for an `LSM` instruction is traced if and only if **ViewData** is active for the first data access in the instruction. If **ViewData** is active only for the second or subsequent data access, then the instruction is traced as Instruction Executed rather than Instruction with Data.

Caution

- If data address comparators are used in exclude regions, **TraceEnable** behavior is UNPREDICTABLE.
- If you want to use data address comparisons to define trace exclude regions, use the data address comparators as ETM resources that define the **TraceEnable** enabling event.

Imprecise TraceEnable events

If **TraceEnable** is imprecise for any reason, any of the following might occur:

- tracing might not turn on in time to trace the required instruction
- tracing might not turn off in time to avoid tracing a specific instruction
- the data for an instruction might not be traced
- trace might be missing at the start of a trace region
- extra trace might appear at the end of a trace region.

With the exception of some IMPLEMENTATION DEFINED configurations, the **TraceEnable** signal is Imprecise if the resource that causes it to change is any of the following:

- Anything selected by the enabling event
- An address comparator configured for Fetch-stage instruction addresses
- An address comparator with its Exact match bit set to 1 (ETMv2.0 and later)
- An address comparator configured for data addresses (before ETMv1.2)
- An address comparator connected to a Context ID comparator, where the Context ID changes. This is imprecise for ETMv3.0 and later. It is precise in ETMv2.x.
- A memory map decoder.

Note

See the appropriate ETM *Technical Reference Manual* for information about any IMPLEMENTATION DEFINED configurations for which the **TraceEnable** signal is not imprecise.

Rules for the transition of TraceEnable

Transitions of **TraceEnable** obey the following rules:

- **TraceEnable** can transition from LOW to HIGH at any time.
- When instruction tracing is enabled, **TraceEnable** can transition from HIGH to LOW only at the end of an instruction. If the processor supports out-of-order data transfers:
  - **TraceEnable** must remain HIGH until the execution of all outstanding data instructions has completed.
  - If there is no outstanding data instruction, **TraceEnable** must remain HIGH until the end of the current instruction.

  Note

  Instruction tracing is enabled by default. From ETMv3.1 it can be disabled by setting the Data-only mode bit, bit [20], of the ETM Control Register to 1. See *ETM Control Register* on page 3-20.
- When instruction tracing is disabled, **TraceEnable** can transition from HIGH to LOW at any time.

The trace start/stop block

The trace start/stop block is shown in Figure 2-6 on page 2-24. It is only available in ETMv1.2 and later:

- in ETMv1.2, the trace start/stop block is always present
- from ETMv2.0, it is optional whether the trace start/stop block is implemented.

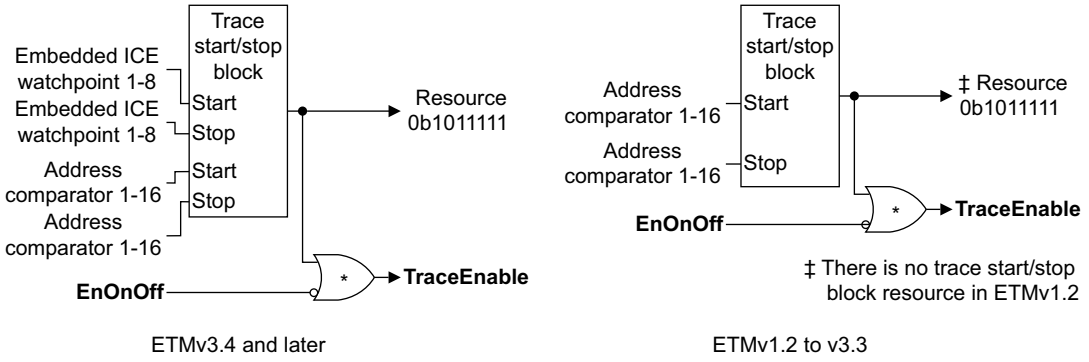


Figure 2-6 Trace start/stop block

From ETMv2.0, the output of the block has two uses, both of which are shown in Figure 2-6, and in Figure 2-4 on page 2-21:

- It provides direct control of **TraceEnable** operation. This control is gated by the **EnOnOff** signal, as shown in Figure 2-4 on page 2-21. For more information see *Using the trace start/stop block to control TraceEnable* on page 2-25.
- It provides an ETM resource, resource type b101 with index b1111 (resource number b101 1111). If the trace start/stop block is implemented this resource is always available, regardless of the state of the **EnOnOff** signal.

In ETM v1.2, the trace start/stop block does not provide an ETM resource, and the only use of the block is the direct control of **TraceEnable**.

In all implementations of the trace start/stop block, the address comparators can provide start and stop inputs to the block:

- Bits [15:0] of the Trace Start/Stop Resource Control Register define address comparators to use as start addresses for the trace start/stop block. If one of the specified address comparators matches then the trace start/stop block receives a start signal and asserts its output HIGH. The output remains asserted HIGH until the block receives a stop signal.
- Bits [31:16] of the Trace Start/Stop Resource Control Register define address comparators to use as stop addresses for the trace start/stop block. If one of the specified address comparators matches then the trace start/stop block receives a stop signal and takes its output LOW. The output remains deasserted (LOW) until the block receives a start signal.
- The behavior of the trace start/stop block is UNPREDICTABLE if the same address comparator is used as both the start input and the stop input to the block.

For more information about configuring the Trace Start/Stop Resource Register see *Trace Start/Stop Resource Control Register, ETMv1.2 and later* on page 3-37.

From ETMv3.4, if an ETM implements any EmbeddedICE watchpoint comparator inputs then those inputs can be used as start and stop inputs to the trace start/stop block. Bit [20] of the Configuration Code Extension Register indicates that the trace start/stop block can use the EmbeddedICE watchpoint inputs, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76. If this bit is set to 1, bits [7:0] of the TraceEnable Start/Stop Embedded ICE Control Register specify EmbeddedICE inputs to use as start inputs to the trace start/stop block, and bits [23:16] specify EmbeddedICE inputs to use as stop inputs to the block. For more information see *Trace Start/Stop EmbeddedICE Control Register, ETMv3.4 and later* on page 3-78.

Note

- From ETMv3.4, an ETM that does not implement any address comparators might implement a trace start/stop block that only has EmbeddedICE watchpoint comparator inputs.
- In all earlier ETMs, it is only possible to implement a trace start/stop block on an ETM that includes address comparators, and the address comparators are available as inputs to the trace start/stop block.

When an ETM reset occurs, the state of the trace start/stop logic is reset to the OFF state.

Using the trace start/stop block to control TraceEnable

In ETMv1.2 or later, you can turn instruction tracing on or off whenever certain instructions are executed or when specified data addresses are accessed. This means that you can trace functions, subroutines, or individual variables held in memory.

You can use the trace start/stop block to enable or disable tracing when any single address comparator matches. The effect is precise only if the address comparison is based on instruction execution or data addresses. You can use instruction fetch comparisons, but the effect is not precise.

From ETMv3.4, the EmbeddedICE watchpoint comparator inputs can also be used as inputs to the trace start/stop block, and therefore can be used to enable or disable tracing in the same way as address comparator matches.

The **EnOnOff** signal, shown in Figure 2-4 on page 2-21, determines whether the trace start/stop block controls **TraceEnable** operation:

EnOnOff LOW: The state of the trace start/stop logic is ignored, and does not directly control **TraceEnable**.

Note

The trace start/stop block output is still available as an ETM resource, and can be used to define the TraceEnable enabling event, as shown in Figure 2-4 on page 2-21.

EnOnOff HIGH: **TraceEnable** is controlled by the trace start/stop block. Tracing only occurs after a start address matches, but before an end address matches. The include/exclude logic still applies. For example, tracing does not occur if the address is in a valid excluded range.

EnOnOff is controlled by bit [25] of the TraceEnable Control 1 Register, see *TraceEnable Control 1 Register* on page 3-39. If you set bit [25] LOW (0), **TraceEnable** behavior is backwards-compatible with ETM versions 1.0 and 1.1.

Note

- Comparisons occur sequentially in address order. For every ON comparison there must be a corresponding OFF comparison. If two separate address comparators are set to conflict (one ON, one OFF), then the OFF comparison is ignored.
- Additional information about the operation of the trace start/stop block is given in *Parallel execution* on page 2-69.

An example of how to program the **TraceEnable** logic is given in *An example TraceEnable configuration* on page 3-115.

2.6.5 ViewData and filtering the data trace

The trace port uses **ViewData** to control whether or not the information for a particular data access is output in the trace stream. By reducing the amount of data trace that is output you can reduce the bandwidth required through the trace port and help prevent the on-chip FIFO from overflowing.

In ETMv1.x only, for *Load/Store Multiple* (LSM) instructions, **ViewData** is sampled only for the first access of the sequence. This means that either none or all of the words transferred are traced. In these ETM versions, this is necessary for successful decompression, and because the transferred data has to be associated with the correct ARM core registers.

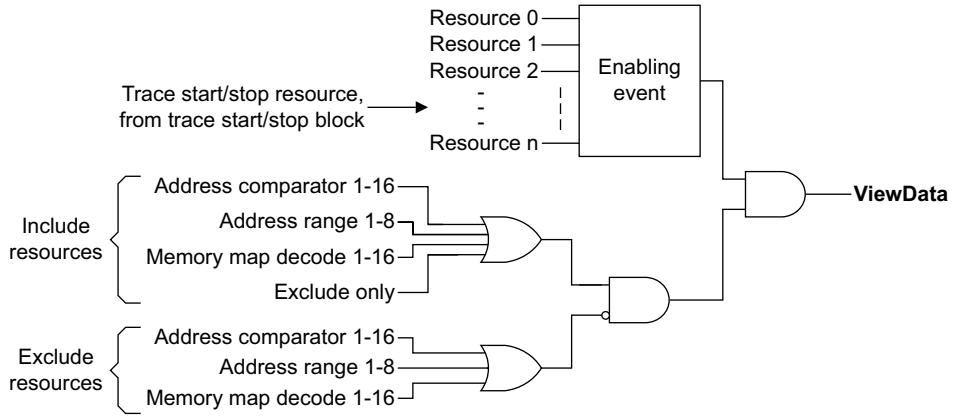
From ETMv2.0, **ViewData** is sampled for each access.

Note

The **ViewData** signal is ignored by the ETM if **TraceEnable** is not asserted.

Data tracing is controlled in a similar way to **TraceEnable**. Control is provided by:

- An enabling event, used to control data tracing.
- Include and exclude address regions, specified by:
  - the address range resources
  - memory map decode resources
  - individual addresses using the address comparator resources.



Note: Most Resource inputs are not shown, see text for details.

Figure 2-7 ViewData configuration

Exclude regions are provided so that you can:

- exclude large areas
- selectively exclude small regions or single addresses from an include region.

  The exclude region might be from a memory map decoder, for example. The exclude region, defined as not being inside the specified address range, is specified as:

  (address < range start address) OR (address >= range end address).

You configure the **ViewData** logic by programming the **ViewData** registers as shown in Figure 2-8. For more information, see *ViewData registers* on page 3-45.

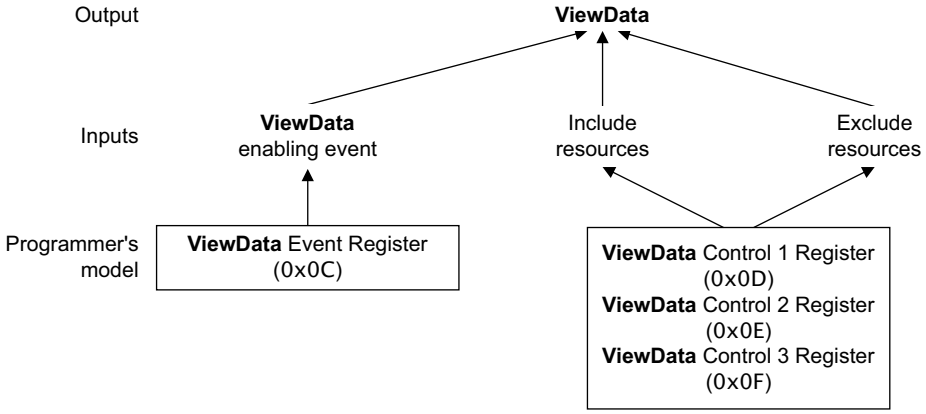


Figure 2-8 Programming the ViewData logic

Imprecise ViewData events

If **ViewData** is imprecise for any reason, any of the following might occur:

- **ViewData** might not turn on in time to trace the required data
- **ViewData** might not turn off in time to avoid tracing specific data.

ViewData is imprecise if the resource that causes it to change is any of the following:

- Anything selected by the enabling event.
- An address comparator configured for Fetch-stage instruction addresses.
- An address comparator with its Exact match bit set to 1.

  Note

  The Exact match bit is present only from ETMv2.0.

Setting start and stop conditions

In ETMv2.0 and later, you can use the trace start/stop resource with **ViewData**. The trace start/stop resource is used and selected through the enabling event, as shown in Figure 2-7 on page 2-27, but its effect is imprecise.

Note

Although it is permitted, it is unusual for the trace start/stop resource to be enabled in **TraceEnable** if you are using it to control **ViewData**.

Filter Coprocessor Register Transfers (CPRT) in ETMv3.0 and later

From ETMv3.0, two bits of the ETM Control Register, register 0, together control *Coprocessor Register Transfer* (CPRT) tracing. The two bits are:

- bit [1], Monitor CPRT
- bit [19], Filter CPRT.

The bit combinations are listed in Table 2-1 on page 2-29.

Table 2-1 Filter CPRT and monitor CPRT combinations

ETM Control Register bits:		Description
Monitor CPRT	Filter CPRT	
0	0	CPRTs not traced
0	1	UNPREDICTABLE
1	0	All CPRTs traced
1	1	CPRTs traced only when **ViewData** is active

See *Coprocessor operations* on page 4-24 for more information on CPRT instructions, and see *ETM Control Register* on page 3-20 for the full description of the ETM Control Register.

Operation of ViewData

A data transfer is traced only if all of the following conditions apply:

- the ViewData enabling event is active
- the transfer does not match any data address comparator that is selected as a ViewData Exclude resource
- the instruction that caused the transfer does not match any instruction address comparator that is configured as a ViewData Exclude resource
- no Memory Map Decode resource that is selected as a ViewData Exclude resource is active
- at least one of the following conditions applies:
  - — ViewData is configured for Exclude-only mode
  - — the transfer matches a data address comparator that is selected as a ViewData Include resource
  - — the instruction that caused the transfer matches an instruction address comparator that is selected as a ViewData Include resource
  - — at least one Memory Map Decode resource that is selected as a ViewData Include resource is active.

ViewData operation examples for Exclude mode

In the following examples, the ViewData enabling event is active and no Memory Map Decode resource is selected as a ViewData Exclude resource:

- if there is no comparator selected as a ViewData Exclude resource all data transfers are traced

- if a data transfer matches a data address range comparator that is selected as a ViewData Exclude resource the transfer is not traced
- if the instruction that caused a data transfer matched an instruction address range comparator that is selected as a ViewData Exclude resource the data transfer is not traced.

ViewData operation examples for Mixed mode

In the following examples, the ViewData enabling event is active and no Memory Map Decode resource is selected as a ViewData Exclude resource:

- if there is no comparator selected as a ViewData Include resource no data transfers are traced
- if a data transfer matches a data address range comparator that is selected as a ViewData Exclude resource the transfer is not traced
- if the instruction that caused a data transfer matched an instruction address range comparator that is selected as a ViewData Exclude resource the data transfer is not traced.

Restrictions on ViewData programming

The following restrictions apply to ViewData programming:

- If an instruction address comparator has its exact match bit set to 1 you must not use it as a ViewData Include or Exclude resource.
- If you use a data address comparator that has its exact match bit set to 1 as a ViewData Include or Exclude resource, then tracing is imprecise. This means that:
  - data that you intended to trace might not be traced
  - data that you intended to exclude from the trace might be traced.

ARM Limited recommends that, if a comparator has its exact match bit set to 1, you do not use that comparator to control ViewData.

2.6.6 Preventing FIFO overflow

When a FIFO overflow occurs, tracing is suspended until the contents of the FIFO have been drained. The resulting gap in the trace is marked, but a large number of overflows can affect the usefulness of the trace.

FIFO overflows are usually the result of large quantities of data tracing combined with a narrow trace port. You can try the following if you experience a large number of overflows:

- if possible, increase the trace port size
- turn off data value tracing, data address tracing, or both
- use **ViewData** and **TraceEnable** to filter the data trace so that only the important data transfers are traced.

Note

Frequent toggling of **TraceEnable** can increase the number of overflows. This is because of the large amount of extra trace produced at the beginning of each trace region to ensure synchronization. It is recommended that you do not disable tracing unless it is switched off for a significant number of cycles. See *Programming strategies* on page 2-20 for details.

In addition, the ETM can support one or both of the following mechanisms to reduce the likelihood of overflow:

- processor stalling, **FIFOFULL**
  - it is IMPLEMENTATION DEFINED whether **FIFOFULL** is supported
- data suppression
  - data suppression is available in ETMv3.0 and later
  - from ETMv3.3, it is IMPLEMENTATION DEFINED whether data suppression is supported.

Data suppression is generally the more effective mechanism. At the time of writing, no ETM implementation supports both options. If both mechanisms are implemented, only one can be enabled at any time, see *Restriction if FIFOFULL and data suppression are both implemented* on page 2-35.

Processor stalling, FIFOFULL

Processor stalling causes the processor to be stalled when the FIFO is close to overflow. This affects the performance of the system. Where supported, an output called **FIFOFULL** tells the processor when to stall.

Processor stalling requires support from both the ETM and the system. Most processors for ETMs supporting **FIFOFULL** have a **FIFOFULL** input, but some, such as the ARM7TDMI™ processor, require support to be built into the memory system to stall the processor core. See the *Technical Reference Manuals* for your core and your ETM for more information. Therefore there are two bits to indicate support:

- A bit in the ETM Configuration Code Register, register `0x01`, indicates whether the ETM supports **FIFOFULL**. See *ETM Configuration Code Register* on page 3-29.
- A bit in the System Configuration Register of the processor core, register `0x05`, indicates whether the system supports **FIFOFULL**. See the description of the System Configuration Register in the *Technical Reference Manual* for your core.

Figure 2-9 shows the generation of the **FIFOFULL** signal.

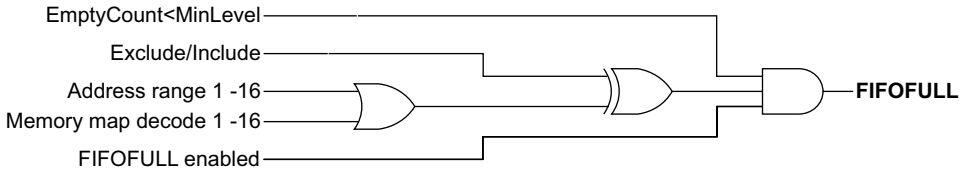


Figure 2-9 FIFOFULL generation

You configure the **FIFOFULL** logic by programming the FIFO overflow registers as shown in Figure 2-10. For more information, see *FIFO overflow registers (FIFOFULL control)* on page 3-41.

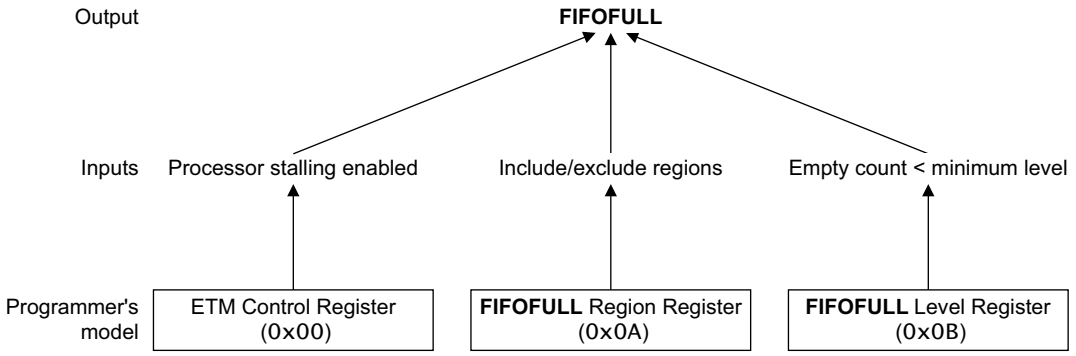


Figure 2-10 Programming the FIFOFULL logic

Some implementations of ETMv2.0 and later might ignore the recommended minimum byte count, and instead assert **FIFOFULL** (if enabled) whenever any bytes are present in the FIFO. This means that the ETM can respond earlier in the ETM pipeline, reducing the chance of overflow.

The **FIFOFULL** signal is a request to the core for it to halt as soon as possible until **FIFOFULL** is deasserted. It can take several cycles for the core to respond to a **FIFOFULL** signal, so in some systems the use of **FIFOFULL** cannot eliminate overflows entirely.

Several early revisions of ARM cores have no support for an additional stall signal. The memory system usually asserts **nWAIT** to stall the core, based on address-based wait states or bus arbitration. In these cases you must design the **FIFOFULL** stall signal into the system because it is at this level that the stalling occurs. To establish whether your ARM processor has a **FIFOFULL** input, see the *Technical Reference Manual* for your core.

You must consider the effect of **FIFOFULL** on interrupt latency. If the assertion of **FIFOFULL** causes a load or store multiple (`LSM`) instruction to be delayed, an `IRQ` or `FIQ` is not taken until the delayed instruction completes. This means that the worst-case interrupt latency can be affected. You can configure some cores, such as the ARM966E-S (Rev 1) and ARM926EJ-S (Rev 0) processors, to ignore **FIFOFULL** when interrupts occur. See the *Technical Reference Manual* for your core to find out if it supports this feature.

Processor stalling takes several cycles to take effect. After **FIFOFULL** is asserted there is a delay before the core is stalled, and there can be an additional delay while trace that has already entered the ETM pipeline enters the FIFO. The FIFOFULL Level Register must take account of this.

Depending on the size of the FIFO and the core in use, the delay can mean that some overflows still occur regardless of the value of the FIFOFULL Level Register. In particular, some cores are only able to stall on instruction boundaries. This reduces the effectiveness of **FIFOFULL** on long `LSM`s.

FIFO overflow is independent of all resource matching, events, and sequencer state changes. No ETM resources are affected by a FIFO overflow.

Data suppression

Data suppression causes data tracing to be disabled when the FIFO is close to overflow. This does not affect the performance of the system.

Instruction tracing is unaffected. Because the bandwidth required for instruction trace is generally far lower than the bandwidth required for data trace, data suppression is normally highly successful in preventing overflow. The resulting gaps in the data trace are marked in the signal protocol.

For more information on the effect of data suppression on the trace, see *Data suppressed packet* on page 7-51.

Data suppression is only available from ETMv3.0:

- it is always supported in ETMv3.0, ETMv3.1 and ETMv3.2
- from ETMv3.3, it is IMPLEMENTATION DEFINED whether it is supported.

If an implementation supports both **FIFOFULL** processor stalling and data suppression, the two features must not be enabled at the same time, see *Restriction if FIFOFULL and data suppression are both implemented* on page 2-35.

A minimum empty byte count is provided to specify the point below which the FIFO is considered full, as shown in Figure 2-11. This setting is made in the FIFOFULL Level Register and is shared with the **FIFOFULL** logic, if present.

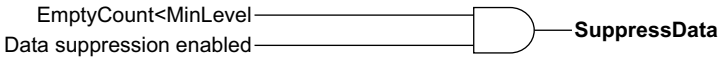


Figure 2-11 SuppressData inputs

You configure the data suppression logic by programming the FIFOFULL Level Register, as shown in Figure 2-12. **SuppressData** ignores the FIFOFULL Region Register, because data suppression cannot be controlled by address regions. An ETM implementation that supports data suppression but does not implement the **FIFOFULL** logic does not implement the FIFOFULL Region Register, but must implement the FIFOFULL Level Register.

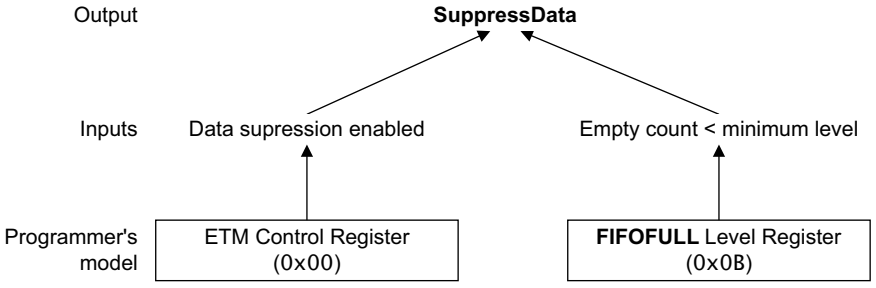


Figure 2-12 Programming the data suppression logic

Checking whether data suppression is supported, ETMv3.3 and later

From ETMv3.3, it is IMPLEMENTATION DEFINED whether an ETM macrocell supports data suppression. Tools can write and then read the ETM Control Register to find whether data suppression is supported.

To avoid changing other ETM control settings, the test process is:

1. Read the ETM Control Register.
2. In the returned data, set bit [18], the Suppress data bit, to 1.
3. Write the modified value back to the ETM Control Register.
4. Read the ETM Control Register again, and check the value of bit [18].
5. Write the original value, from stage 1, back to the ETM Control Register.

Checking bit [18] of the register value returned at stage 4 of the test indicates whether data suppression is supported. The possible results are shown in Table 2-2.

Table 2-2 Testing whether data suppression is supported, ETMv3.3 and later

ETM Control Register bit [18]	Data suppression option
1	Data suppression supported
0	Data suppression not supported

For details of the ETM Control Register, see *ETM Control Register* on page 3-20.

Note

From ETMv3.3, the data tracing options provided by an ETM macrocell are IMPLEMENTATION DEFINED. If a macrocell provides none of the optional data trace features then data suppression is not supported, and bit [18] of the ETM Control Register reads-as-zero. For more information see *Data tracing options, ETMv3.3 and later* on page 7-54.

Restriction if FIFOFULL and data suppression are both implemented

If an ETM implements both FIFOFULL and data suppression, then only one of these features can be active at any one time. This means that there are restrictions on the permitted values of bits [18, 7] in the ETM Control Register. These are shown in Table 2-3.

Table 2-3 Permitted Suppress data and Stall processor settings, ETM Control Register

ETM Control Register		Effect
Bit [18][a]	Bit [7][b]	
0	0	**FIFOFULL** processor stalling and data suppression both disabled.
0	1	**FIFOFULL** processor stalling enabled, data suppression disabled.
1	0	**FIFOFULL** processor stalling disabled, data suppression enabled.
1	1	*Prohibited combination*. ETM behavior is UNPREDICTABLE.

a. Suppress data bit.
b. Stall processor bit (**FIFOFULL**).

For more information about the register, see *ETM Control Register* on page 3-20.

Note

- If an ETM implementation does not support one of these features then any write to the corresponding bit of the ETM Control Register is ignored. For example, if an implementation does not support **FIFOFULL** processor stalling then any write to bit [7] of the register is ignored.
- **FIFOFULL** processor stalling requires support by the connected processor core, see *Processor stalling, FIFOFULL* on page 2-31.

2.7 Address comparators

This section describes in detail the address comparators and how they are used. It assumes you are familiar with the general operation of the comparators, as described in the sections:

- *Single address comparators* on page 2-5
- *Address range comparators* on page 2-6.

Address comparators are controlled by the Address Comparator Value Registers and the Address Access Type Registers. These registers are introduced in the section *Address comparator registers* on page 3-50, and described in detail in the sections:

- *Address Comparator Value Registers* on page 3-50
- *Address Access Type Registers* on page 3-51.

This description of the address comparators assumes you are familiar with these registers and how to use them.

2.7.1 Comparator access size

The *Access size* field of the Address Access Type Register indicates the size of the address being monitored:

Instruction address comparisons

When an address comparator is configured to perform instruction address matching, the access size field must be set to correspond to the instruction set:

- access size word for the ARM instruction set
- access size halfword for the Thumb and ThumbEE instruction sets
- access size byte for the Jazelle instruction set.

Data address comparisons

When a single address comparator is configured to perform data address comparisons the access size field is used generate a match when the byte, halfword or word at the selected address matches, even if the 32-bit address in the Address Comparator Value Register does not match completely.

For more information about the behavior of data address comparators see *Operation of data value comparators* on page 2-56.

While unaligned transfers are correctly monitored, the address being monitored cannot itself be unaligned. If the size field is set to b01, halfword data, the address comparator value must be halfword-aligned. If the size field is set to b11, word data, the address must be word-aligned.

Note

- For Instruction Address comparisons, no filtering is performed on the size of the access itself. A single address comparator matches if the value in the Address Comparator Value Register matches the address of the access, regardless of the setting of the size field. Similarly, no filtering is performed based on the actual instruction set in use. However the size field must be set correctly for the instruction set in use.

- When an instruction address comparison must match from multiple instruction sets, the field size must be set to largest instruction size required. For example, to match on word (ARM) or halfword (Thumb) instructions, set the size field to word. This applies to single address comparisons and address range comparisons.

The behavior of the access size field depends on the ETM architecture version, and is described in detail in the following sections:

- *Comparator access size field behavior, in ETMv3.1 and later*
- *Comparator access size field behavior, in ETMv3.0 and earlier* on page 2-39.

Note

Compilers often choose to use a word transfer to access bytes or halfwords. For example, a data structure is usually copied using word transfers, regardless of whether it contains byte or halfword quantities. In ETMv3.0 and earlier, a byte at address `0x1003`, when accessed as part of a word transfer at `0x1000`, does not match an address comparator programmed for address `0x1003`, because the address the ETM is comparing against is `0x1000`. See Figure 2-19 on page 2-42 for an example.

2.7.2 Comparator access size field behavior, in ETMv3.1 and later

Behavior of the size field in ETMv3.1 and later depends on the type of comparison, and is described in the following sections:

- *Single address comparators configured for data addresses*
- *Single address comparators configured for instruction addresses* on page 2-38
- *Address range comparators configured for data addresses* on page 2-38.
- *Address range comparators configured for instruction addresses* on page 2-39.

Note

- If data address comparators are used in exclude regions, the ETM **TraceEnable** behavior is UNPREDICTABLE. See the Caution in *TraceEnable and filtering the instruction trace* on page 2-20 for more information.
- From ETMv3.3, an ETM implementation might not support data address comparisons. See *No data address comparator option, ETMv3.3 and later* on page 2-7 for more information.

Single address comparators configured for data addresses

The *Access size* field enables any access to any byte in the selected byte, halfword or word to cause the comparator to match. This behavior is required to perform reliable address comparisons on unaligned accesses.

Figure 2-13 on page 2-38 shows how this can be used.

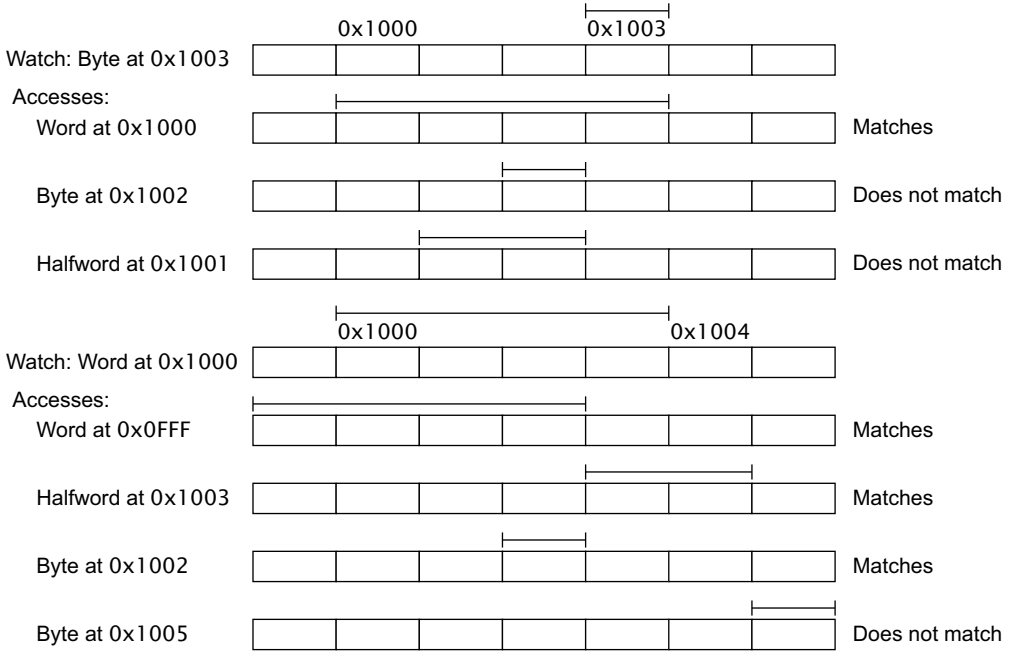


Figure 2-13 Single address comparisons in ETMv3.1 and later

For more information about the behavior of data value comparators see *Operation of data value comparators* on page 2-56

Single address comparators configured for instruction addresses

Instruction address comparators ignore the *Access size* field, and the address must match exactly. However, the size field must still be set to the expected instruction set to help some ETMs adapt to matching instructions in different states.

Address range comparators configured for data addresses

In ETMv3.1 and later, range comparators ignore the value of the *Access size* field.

The address range comparator matches if any of the accessed bytes fall in the defined range. Figure 2-14 on page 2-39 shows an example of this.

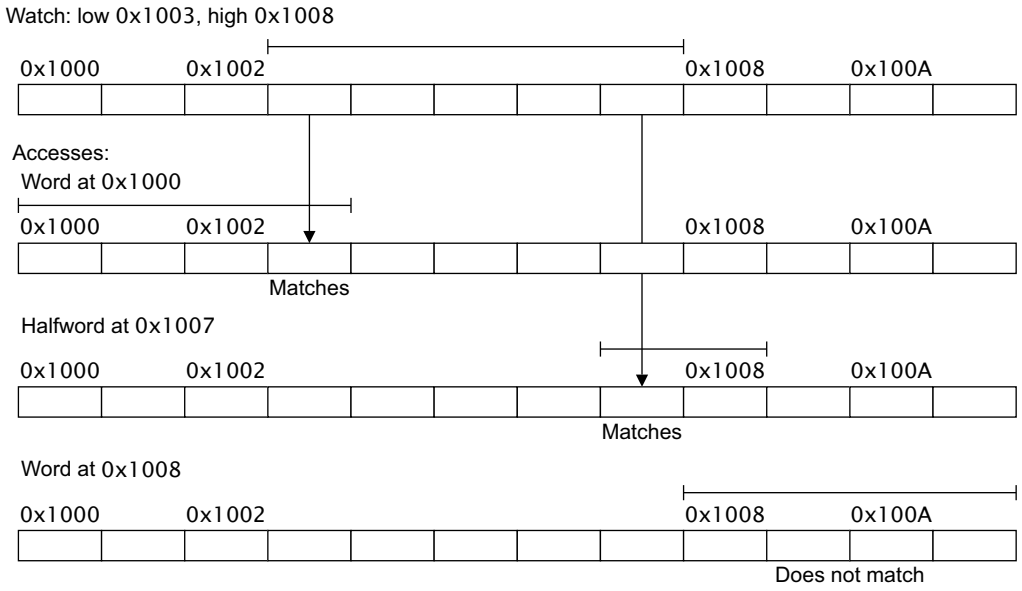


Figure 2-14 Range comparisons in ETMv3.1 and later

Address range comparators configured for instruction addresses

The address range comparator matches if the first byte of the instruction falls in the range. Although the *Access size* field is ignored, it must still be set to the expected instruction set to help some ETMs adapt to matching instructions in different states.

2.7.3 Comparator access size field behavior, in ETMv3.0 and earlier

The *Access size* field is implemented as a size mask on the bottom two bits of the address of the access:

Word data bits [1:0] of the access address are masked

Halfword data bit [0] of the access address is masked

Byte data no masking is performed.

In general, you must mask bits [1:0] for word quantities, because this causes byte accesses to locations in the word to be traced. See Figure 2-15 on page 2-40.

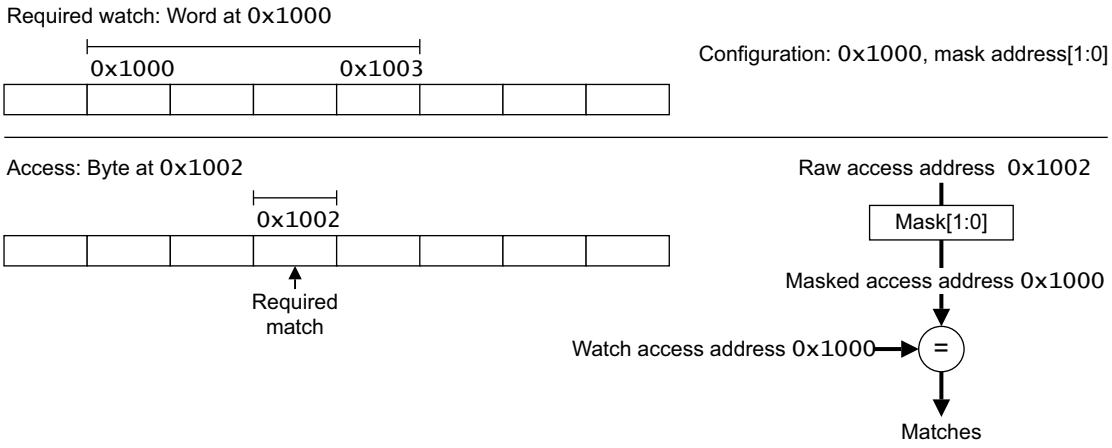


Figure 2-15 Successful match of a byte access with word mask set

To catch all possible accesses to a given byte, you must mask bits [1:0] as shown in Figure 2-16 on page 2-41 and Figure 2-19 on page 2-42. This has the side effect of falsely matching accesses to other bytes in the same word, as shown in Figure 2-18 on page 2-42.

Where all accesses are by a byte transfer, no masking is necessary, and the comparator matches only if the required byte is accessed. This is shown in Figure 2-17 on page 2-41

You must also mask bits [1:0] for halfword quantities that might be accessed as part of the containing word, or only mask bit [0] to avoid matching on accesses to the other halfword in the word.

Unaligned accesses are not supported by these implementations.

Figure 2-16 on page 2-41 shows an example of a successful match of word access with word mask set.

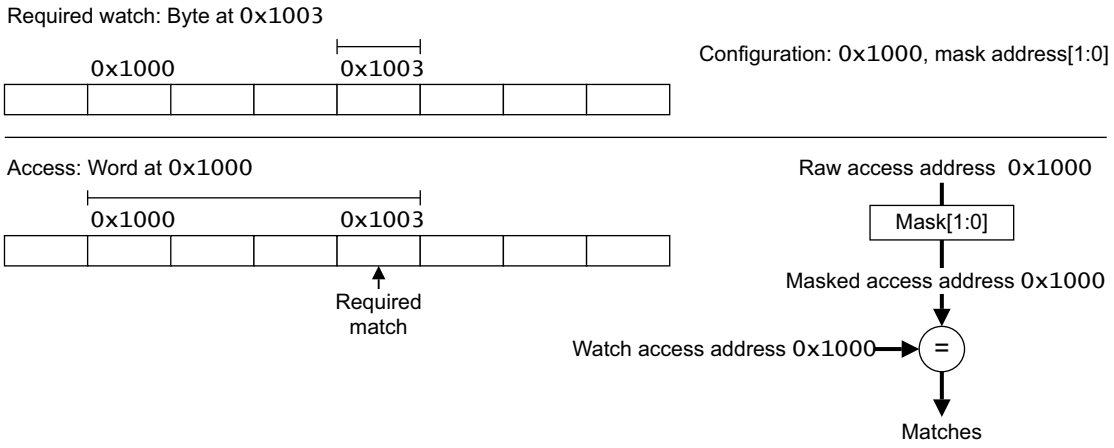


Figure 2-16 Successful match of word access with word mask set

Figure 2-17 shows a successful match of byte access with word mask set.

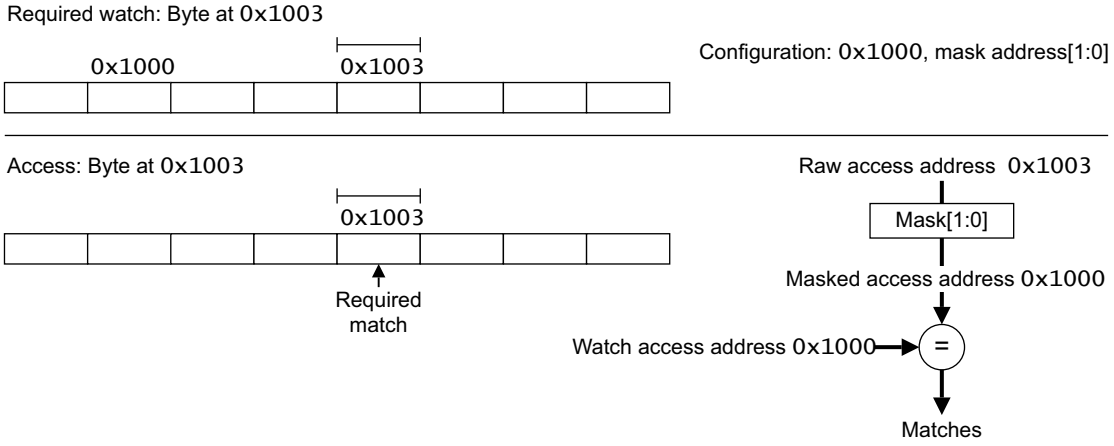


Figure 2-17 Successful match of byte access on byte watch with word mask set

Figure 2-18 on page 2-42 shows an example of an unwanted match with word mask set.

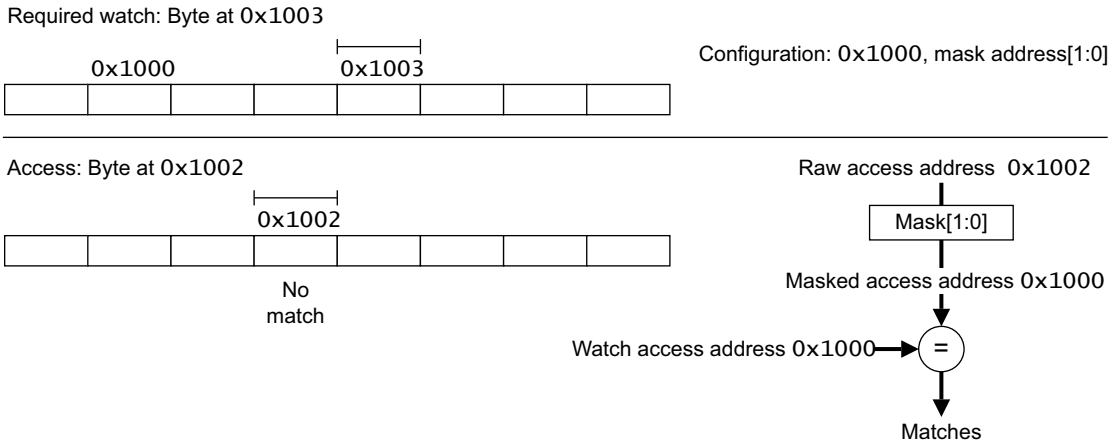


Figure 2-18 Unwanted match of byte access on byte watch with word mask set

Figure 2-19 shows an example of a failed match when no mask is used.

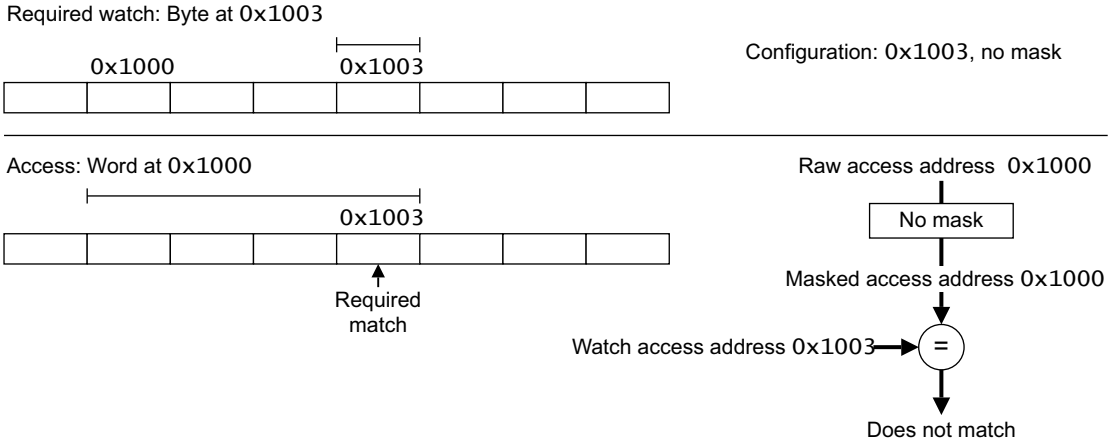


Figure 2-19 Failed match with no mask

Address range comparison behavior of ETMv3.0 and earlier

The address in the Address Comparator Value Register must be aligned correctly for the access width (word, halfword or byte) indicated by the *Access size* field, see *Comparator access size* on page 2-36. When this is done, the value of the Access size field has no effect on address range comparisons. The address range comparator matches if the base access address is in the range. This means that the low address must be chosen carefully to catch all required accesses, as shown in Figure 2-20 and Figure 2-21 on page 2-44.

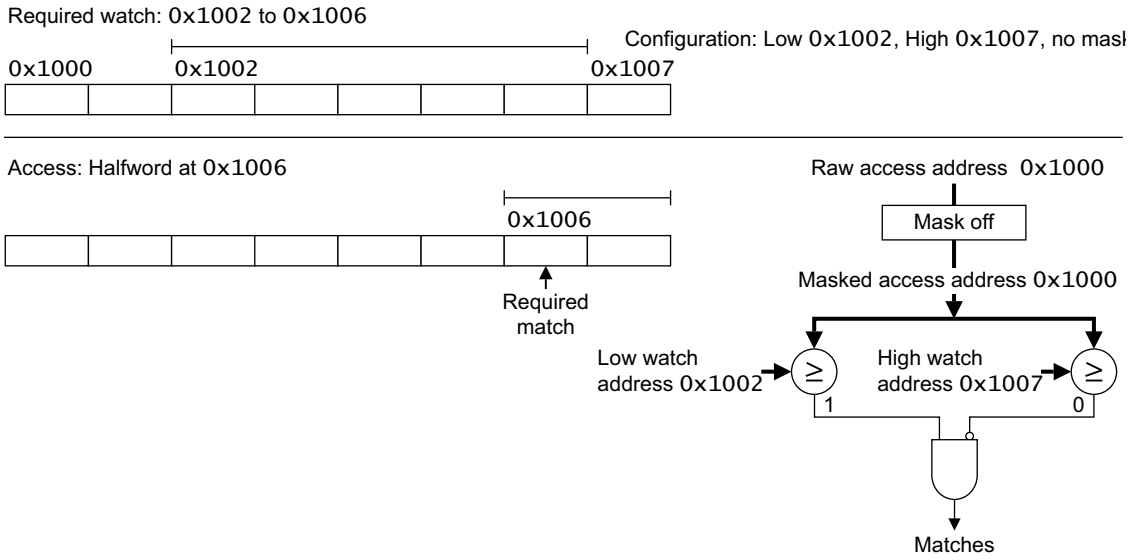


Figure 2-20 Range address successful match, ETMv3.0 or earlier

Figure 2-21 on page 2-44 shows a range address failed match.

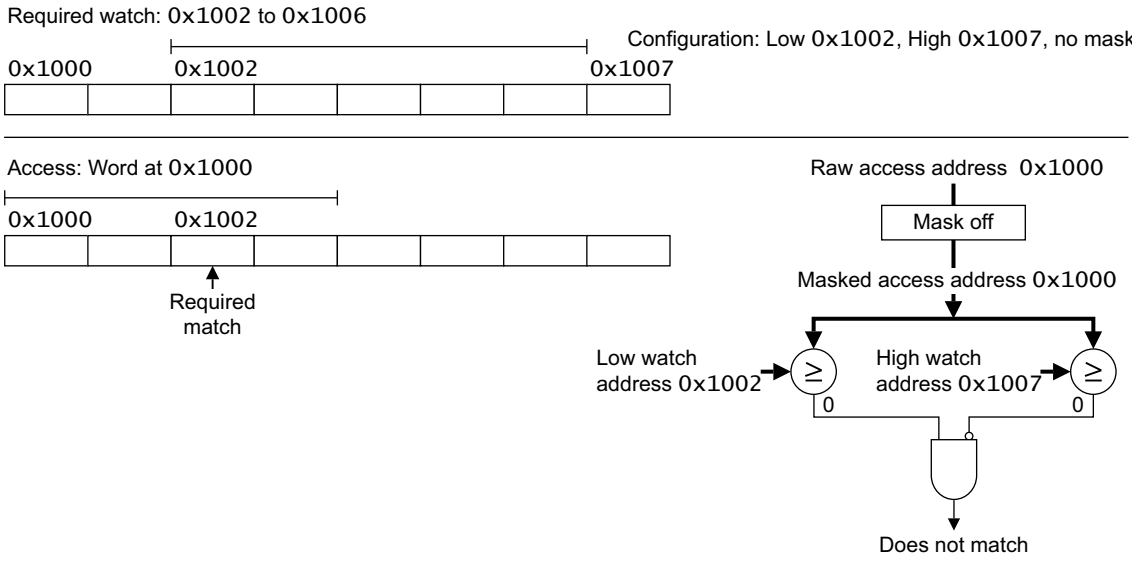


Figure 2-21 Range address failed match, ETMv3.0 or earlier

2.7.4 Exact matching, ETMv2.0 and later

This section describes exact matching in ETMv2.0 and later, where exact matching is controlled by a bit in the Address Access Type Register. For information about exact matching in ETMv1.x see *Exact matching, ETMv1.x* on page 2-49.

The *Exact match* bit is bit [7] of the Address Access Type Register. In ETMv2.0 and later you can set the *Exact match* bit to 1 to enable the comparator to *match later*, as described in this section. This late matching can improve accuracy in the comparison. For example, in the case of out-of-order data it enables the comparator to wait until the result of the data value comparison is known before matching. The functionality of the Exact match bit depends on:

- the access type
- the occurrence of interrupts and prefetch aborts
- the occurrence of out-of-order transfers
- the occurrence of data aborts.

Use of a comparator with the Exact match bit set to 1 in the programming of **TraceEnable** or **ViewData** results in Imprecise Tracing.

In general, the Exact match bit is set to 1 when the comparator is used by a derived resource, and set to 0 when the comparator is used directly as an include/exclude region.

From ETMv3.3, setting the Exact match bit affects the holding behavior of the address range comparators, see *Behavior of address comparators* on page 2-49 for details.

Exact matching behavior for the different access types is described in the following sections:

- *Exact matching for instruction address comparisons*
- *Exact matching for data address comparisons* on page 2-46.

Exact matching for instruction address comparisons

For Execute-stage instruction address comparisons, the behavior of the Exact match bit depends on whether the instruction is canceled because of an exception. This is shown in Table 2-4.

Table 2-4 Effect of exact match bit settings for instruction address comparisons

Exact match bit value	Instruction canceled	Instruction not canceled
0	Comparator matches	Comparator matches
1	Comparator does not match	Comparator matches

For Fetch stage instruction address comparisons, the Exact match bit is ignored. Canceled instructions can cause the comparator to match.

In Jazelle state, the comparator matches at the beginning of the bytecode if the Exact match bit is set to 0, and at the end of the bytecode if the Exact match bit is set to 1. This enables the comparator to wait until it knows whether or not the bytecode was interrupted.

Whenever an instruction is considered for tracing, because **TraceEnable** enables tracing, the ETM must compare the instruction address with the address comparators. For example, an ETM can trace a prefetch abort in either of two ways:

- the instruction that prefetch aborts is traced and then a prefetch abort exception indicates that the instruction is cancelled.
- the instruction that prefetch aborts is not traced and the prefetch abort exception in non-cancelling.

When a prefetch abort is traced as cancelling, the comparators compare the address of the instruction that prefetch aborted. The exact match bit is used here to ensure that the comparator only matches when the prefetch abort does not occur. When traced as non-cancelling, the ETM does not have the address of the prefetch aborted instruction because the instruction is not traced, so the comparators do not compare based on this address and will never match.

Exact matching for data address comparisons

This section describes exact matching behavior on accesses to data addresses, whether or not data value comparison is enabled for the access. For additional information about comparator behavior when data value comparisons are enabled see *Operation of data value comparators* on page 2-56.

The rules for data address comparisons are fairly complex, and depend on:

- Whether the comparison is on a normal transfer, or on an out-of-order transfer.
  - In ETMv3.0 and earlier, matching behavior on an out-of-order transfer is different to the behavior in ETMv3.1 and later.
- The setting of these fields in the Address Access Type Register:
  - Data value comparison control field, bits [6:5]
  - Exact match bit, bit [7]

  For more information see *Address Access Type Registers* on page 3-51.
- Whether the data value matches the comparison value held in the Data Comparator Value Register. This data value comparison is masked with the value held in the Data Comparator Mask Register. For more information see *Data value comparator registers* on page 3-54.
- Whether the data transfer:
  - causes a synchronous data abort
  - is a failed store-exclusive transfer.

  Synchronous data aborts and failed store-exclusive transfers have the same effect on the data value comparison.

Table 2-5 shows the data comparison results for normal transfers, and Table 2-6 on page 2-47 shows the results for comparisons on out-of-order transfers. In all cases, the final comparator result is the logical AND of the result from one of these tables with the result of the associated address comparison.

For more information about the effect of the value of the Exact match bit see *Additional details of the effect of the Exact match bit* on page 2-48.

Table 2-5 Data value comparisons for normal transfers

Address Access Type Register values:		Data comparison result when:		
DCompare mode[a]	**Exact match bit[b]**	**Transfer aborts or is a Store fail[c]**	**Data value matches**	**Data value does not match**
No data value comparison	0	1	1	1
No data value comparison	1	0	1	1
Data value matches	0	1	1	0

Table 2-5 Data value comparisons for normal transfers (continued)

Address Access Type Register values:		Data comparison result when:		
DCompare mode[a]	Exact match bit[b]	Transfer aborts or is a Store fail[c]	Data value matches	Data value does not match
Data value matches	1	0	1	0
Data value does not match	0	1	0	1
Data value does not match	1	0	0	1

a. Data compare mode, bits [6:5] of the Address Access Type Register, see *Address Access Type Registers* on page 3-51. The permitted values are:
   b00: No data value comparison is made
   b01: Comparator can match only if Data value matches
   b11: Comparator can match only if Data value does not match.
b. Bit [7] of the Address Access Type Register, see *Address Access Type Registers* on page 3-51.
c. Values in this column apply if the transfer causes a synchronous data abort or is a Store Exclusive that fails. The result is not affected by the value of the data associated with the transfer.

Note

From the table, notice that if the Exact match bit is *not* set to 1, a data comparison match is generated by:

- a transfer that causes a synchronous data abort
- a store-exclusive that fails.

Table 2-6 Data value comparisons on an out-of-order transfer

Address Access Type Register values:		Data comparison result when:		
DCompare mode[a]	Exact match bit[b]	Transfer aborts or is a Store fail[c]	Data value matches	Data value does not match
No data value comparison	0	1	1	1
No data value comparison	1	0[d]	1[d]	1[d]
Data value matches	0	1	1	1
Data value matches	1	0[e]	1[e]	0[e]
Data value does not match	0	1	1	1
Data value does not match	1	0[e]	0[e]	1[e]

a. Data compare mode, bits [6:5] of the Address Access Type Register, see *Address Access Type Registers* on page 3-51. The permitted values are:
   b00: No data value comparison is made
   b01: Comparator can match only if Data value matches
   b11: Comparator can match only if Data value does not match.
b. Bit [7] of the Address Access Type Register, see *Address Access Type Registers* on page 3-51.
c. Values in this column apply if the transfer causes a synchronous data abort or is a store-exclusive that fails. The result is not affected by the value of the data associated with the transfer.
d. In ETMv3.1 and later, the comparator waits for the out-of-order data to return, and then gives the result shown.
In ETMv3.0 and earlier, the result is returned immediately, *and is always 1*.
e. The comparator waits for the out-of-order data to return and then gives this result.

Note

From the table, notice that if the Exact match bit is *not* set to 1, the data value comparator *always* reports a match when an out-of-order transfer occurs.

Additional details of the effect of the Exact match bit

Exact match bit set to 0 (default setting)

When an out-of-order transfer, synchronous data abort, or Store Exclusive fail occurs, the comparator matches immediately. This means that tracing of out-of-order transfers, data aborts and Store Exclusive fails is based on the data address only, because the data value is assumed to be invalid.

Tracing out-of-order transfers based on the address alone is useful when the comparator is used for trace filtering, if you do not mind generating some additional trace that you do not require.

Exact match bit set to 1

When an out-of-order transfer occurs, the comparator waits for the data value to be returned, then matches if the data value matches.

Waiting for the data value compare to occur is useful when data values are used by derived resources to create triggers and other events.

Waiting for the data value compare to occur causes the out-of-order transfer to be missed if the comparator is used directly as an include region by **TraceEnable** or **ViewData**. Because out-of-order data is traced only if the out-of-order placeholder is traced, the result of having the exact match bit set to 1 is that the data is not traced even if it matches.

When a data abort or a Store Exclusive fail occurs, the comparator does not output a match regardless of whether or not a data value comparison is requested. This behavior is often preferred when a comparator is meant to match only once, because aborted accesses are usually re-attempted when the condition causing the abort condition has been resolved.

When counting the execution of load or store instructions, the occurrence of data aborts and the subsequent retrying of instructions causes the instruction count to be larger than expected.

Note

Using data values to create an event, such as a sequencer transition, can result in out-of-order events occurring because the data might be returned out-of-order. If you are concerned that the nonblocking cache might affect programmed events, you can disable it in the core. For more information, see the *Technical Reference Manual* for your core.

2.7.5 Exact matching, ETMv1.x

When an instruction address comparator is selected as an event resource, ETMv1.x always behaves as though the Exact match bit is 1, so canceled instructions do not match.

When an instruction address comparator is selected as an include or exclude region, or by the trace start/stop block, ETMv1.x behaves as though the Exact match bit is 0, so canceled instructions do match.

Data aborts always match, as though the Exact match bit is 0.

2.7.6 Behavior of address comparators

Address comparator behavior depends on whether the comparator is used for **TraceEnable**, **ViewData** or events:

- Table 2-7 on page 2-50 shows the differences in the behavior of the different outputs of a single address comparator
- Table 2-8 on page 2-50 shows the differences in the behavior of the different outputs of an address range comparator, for ETMv3.3 and later
- Table 2-9 on page 2-50 shows the differences in the behavior of the different outputs of an address range comparator, for ETMv3.2 and earlier.

In these tables:

Fire Means that the comparator matches for one cycle only.

Sticky, for data address comparators

Means that the comparator matches until a new data transfer occurs.

Sticky, for instruction address comparators

The meaning is IMPLEMENTATION DEFINED, but must be one of:

- If the processor supports an independent Load/Store Unit (LSU) where data transfers can return out of order in relation to the instruction stream, then *Sticky* means that the comparator matches, and continues to match until the last executed data instruction completes. If there is no data instruction outstanding then the comparator matches until the end of the current instruction.

- If the processor returns all data transfers in-order in relation to the instruction stream, then *Sticky* means that the comparator matches up to, but not including, the next instruction.

Table 2-7 Context-dependent behavior of single address comparators

Comparator type	Exact Match bit[a]	Event resource	TraceEnable
Instruction address	0	Fire	Fire
Instruction address	1	Fire	Fire
Data address	0	Fire	Fire
Data address	1	Fire	Fire

a. Bit [7] of the Address Access Type Register, see *Address Access Type Registers* on page 3-51.

Table 2-8 Context-dependent behavior of address range comparators, from ETMv3.3

Comparator type	Exact Match bit[a]	Event resource	TraceEnable
Instruction address	0	Sticky	Sticky
Instruction address	1	Fire	Fire
Data address	0	Sticky	Sticky
Data address	1	Fire	Fire

a. Bit [7] of the Address Access Type Register, see *Address Access Type Registers* on page 3-51.

Table 2-9 Context-dependent behavior of address range comparators, before ETMv3.3

Comparator type	Exact Match bit[a]	Event resource	TraceEnable
Instruction address	0	Sticky	Sticky
Instruction address	1	IMPLEMENTATION DEFINED[b]	IMPLEMENTATION DEFINED[b]
Data address	0	Sticky	Sticky
Data address	1	IMPLEMENTATION DEFINED[b]	IMPLEMENTATION DEFINED[b]

a. Bit [7] of the Address Access Type Register, see *Address Access Type Registers* on page 3-51.
b. In ETMv3.2 and earlier, the typical IMPLEMENTATION DEFINED behavior is Sticky.

Note

- As shown in Table 2-7 on page 2-50, Table 2-8 on page 2-50, and Table 2-9 on page 2-50:
  - The **TraceEnable** and the Event resource outputs of a comparator always behave in the same way.
  - For the address range comparators, from ETMv3.3, as shown in Table 2-8 on page 2-50:
    when the exact match bit is set to 1 the comparator output behavior is always Fire
    when the exact match bit is set to 0 the comparator output behavior is always Sticky.
- *Rules for the transition of TraceEnable* on page 2-23 describes the restrictions on when **TraceEnable** can transition.
- *Operation of ViewData* on page 2-29 describes how the comparators affect ViewData operation.

2.7.7 Access types for address range comparators

If you are using two address comparators as an address range comparator, the access type must be identical for each, otherwise the behavior of the comparator is UNPREDICTABLE. The only exceptions to this are:

- Bits [6:5] must be set only for the first comparator in the pair. These bits control data value comparisons.
- The special case where the range includes the address `0xFFFFFFFF`. See *Selecting a range to include address 0xFFFFFFFF*.

Note

This information is also included in the section *Address comparator registers* on page 3-50.

Selecting a range to include address 0xFFFFFFFF

Ranges are defined to be exclusive of the upper address, so if you specify an upper address of `0xFFFFFFFF`, only addresses up to and including `0xFFFFFFFE` match. To specify a data address to include `0xFFFFFFFF`, configure the upper address comparator as follows:

- Value Register = `0xFFFFFFFF`
- set Access Type Register bits [4:3] (size mask) to b11.

This is the only case where the size mask can be different between the two address comparators of an address range comparator.

For more information, see *Address range comparators* on page 2-6.

2.7.8 Comparator precision

The section *Single address comparators* on page 2-5 summarizes the configuration options for the address comparators.

Address comparators can cause imprecise tracing or imprecise events in the following cases:

- When the address comparator is configured for instruction fetch comparisons, by setting the *access type* field of the Address Access Type Register to b000, Instruction fetch.
- When an address comparator is configured with Context ID comparison enabled, by setting the *Context ID comparator control* field of the Address Access Type Register to a value other than b00.

For more information about these configuration options see *Address Access Type Registers* on page 3-51. For more information about imprecise tracing see *Imprecise TraceEnable events* on page 2-22.

2.7.9 Coprocessor transfers

Coprocessor transfers (CPRT) do not have an associated data address, so address comparators never match on a coprocessor transfer. When a coprocessor transfer occurs, any address range comparators stop matching.

If CPRT tracing is not supported in the ETM, then coprocessor transfers are ignored by the ETM and the comparator outputs are unaffected by any coprocessor transfers. For more information about the IMPLEMENTATION DEFINED features of data value comparators see *Data tracing options, ETMv3.3 and later* on page 7-54.

2.7.10 Comparator configuration example

This section uses an example to illustrate how the comparators work:

- Example 2-1 summarizes the configuration of the comparators for the example
- *Operation of the comparators* on page 2-53 describes how these comparators operate
- *Programming the comparator registers for this example* on page 2-54 gives details of how the comparator registers must be programmed to implement this example.

Note

The description given in this example only considers the behavior of the address comparators when used to control **TraceEnable**.

Example 2-1 Configuration to demonstrate comparator behavior

Four single address comparators are configured as follows:

Single address comparator 1

Instruction execute, address `0x1000`.

Single address comparator 2

Instruction execute, address `0x1008`.

Address range comparator 1

Formed from address comparators 1 and 2.

Single address comparator 3

Data store, address `0x2000`.

Single address comparator 4

Data store, address `0x2008`.

Address range comparator 2

Formed from address comparators 3 and 4.

Operation of the comparators

With comparators set up as described in Example 2-1 on page 2-52, consider the sequence of operations described in Table 2-10.

Table 2-10 Single address and address range comparators example

Cycle	Action	Comparators, TraceEnable output					
		SAC1	SAC2	ARC1	SAC3	SAC4	ARC2
1	Instruction at `0x1000` executed. Data written to `0x2000` for instruction at `0x1000`.	Matches	-	Matches	Matches	-	Matches
2	Instruction at `0x1004` executed.	-	-	Matches	-	-	Matches
3	Data read from `0x2004` for instruction at `0x1004`.	-	-	Matches	-	-	-
4	Instruction at `0x1008` executed.	-	Matches	-	-	-	-

The following matches are produced:

- Single address comparator 1, SAC1 in the table, matches during cycle 1.
- Single address comparator 2, SAC2 in the table, matches during cycle 4.
- Address range comparator 1, formed from address comparators 1 and 2, SAC1 and SAC2 in the table, matches during cycles 1, 2, and 3.

  No instruction address is accessed during cycle 3, so the address range comparator retains its previous state.
- Single address comparator 3, SAC3 in the table, matches during cycle 1.

- Single address comparator 4, SAC4 in the table, never matches.
- Address range comparator 2, formed from address comparators 3 and 4, (SAC3 and SAC4 in the table) matches during cycles 1 and 2.

  No data address is accessed during cycle 2, so the address range comparator retains its previous state.

  Address range comparator 2 does not match during cycle 3 because it is configured for stores and a load occurs.

Only the first address comparator of an address range comparator pair can have a data value comparator. In this example, only address range comparator 2 can have a data value comparator, because address range comparator 1 is matching on instruction accesses. If a data value comparator is programmed for address range comparator 2, it applies to the whole address range. See the Notes in the section *Programming the comparator registers for this example* for more information.

Programming the comparator registers for this example

To implement the example, the comparator registers must be programmed as follows.

Single address comparator 1 (start address for address range comparator 1) Address Comparator Value Register 1

Program with the start address for address range comparator 1, `0x00001000`.

Address Access Type Register 1

Set the Data value comparison control field to b00, for no data comparison.

Set the Access size field to b11 for ARM instructions, or as appropriate for the current processor state.

Set the Access type field to b001, for instruction execute.

The security level, Context ID and exact match fields of the register might also be used.

Single address comparator 2 (end address for address range comparator 1) Address Comparator Value Register 2

Program with the end address for address range comparator 1, `0x00001008`.

Address Access Type Register 2

Because this address comparator forms part of an address range comparator this register *must* be programmed with exactly the same value as that used for Address Access Type Register 1.

Single address comparator 3 (start address for address range comparator 2) Address Comparator Value Register 3

Program with the start address for address range comparator 2, `0x00002000`.

Address Access Type Register 3

Set the Data value comparison control field to b00, for no data comparison.

Note

If required, data value comparison can be made part of the address range comparator 2 comparisons. To do this you must:

- set the Data value comparison control field to b01 or b11, as appropriate
- program the Data Comparator Value Register 1 and Data Comparator Mask Register 1 for the required data comparison, as described in *Data value comparator registers* on page 3-54.

Set the Access size field to b11 for word access address matching, or as appropriate for the required matching. The operation of this field depends on the ETM version, see *Comparator access size* on page 2-36.

Set the Access type field to b110, for data store.

The security level, Context ID and exact match fields of the register might also be used.

Single address comparator 4 (end address for address range comparator 2) Address Comparator Value Register 4

Program with the end address for address range comparator 2, `0x00002008`.

Address Access Type Register 4

Because this address comparator forms part of an address range comparator this register *must* be programmed with exactly the same value as that used for Address Access Type Register 1.

Note

If a data value comparator is used with address range comparator 2 then the Data value comparison control field is only set in Address Access Type Register 3, and this field is b000 in Address Access Type Register 4. All other fields must be identical in the two Address Access Type registers.

For full details of the Address Comparator Value and Address Access Type registers see *Address comparator registers* on page 3-50.

2.8 Operation of data value comparators

This section the operation of data value comparators for all ETM versions, and describes the additional comparator features introduced from ETMv3.3.

This section is organized as follows:

- *Terms used in this section*
- *Operation of data value comparators, ETMv3.2 and earlier* on page 2-57
- *Operation of data value comparators, ETMv3.3 and later* on page 2-58
- *Summary of alignment and endianness considerations for different ETM versions* on page 2-61.

2.8.1 Terms used in this section

In this section, the following terms are used with the specific meanings given:

Access address

This is the data address accessed by the processor core.

Access size This is the size of the data access made by the processor core. The size is word, halfword or byte. Multiple word LSM accesses are traced as a sequence of word accesses.

Access value

This is the value of the data accessed at the access address. It depends on the access size, for example for a byte access the access value is 8 bits of data.

Comparison address

This is the address configured in the comparator registers. It can be a single address, programmed into a single Address Comparator Value Register, or an address range specified by a pair of Address Comparator Value Registers.

The Address Access Type Register, or Registers, associated with the comparison are programmed for data value comparisons.

For more information see *Address comparator registers* on page 3-50.

Comparison size

This is the size of the required comparison, as defined by the Comparison access size field of the Address Access Type Register for the comparison. For more information see *Comparator access size* on page 2-36 and *Address Access Type Registers* on page 3-51.

When the Comparison size is halfword or byte, from ETMv3.3 the appropriate Data Comparator Mask Register must have the same mask value set in each byte or halfword, as described in this section.

Note

This is a significant change from how the Data Comparator Mask Register is programmed in earlier ETM versions:

- In ETMv1, unused byte lanes must be masked out, based on the comparison size and the bottom bits of the Address Comparator Value Register.
- In ETM versions 2.0 to 3.2, the register must be set to mask out the unwanted area of the comparison. For example, if you want to compare a byte at address `0x2000`, bits [31:8] of the Data Comparator Mask Register must all be set to 1.

For more information see *Operation of data value comparators, ETMv3.2 and earlier*.

For more information see *Data value comparator mask registers* on page 3-56.

Comparison value

This is the value to be used for the data comparison, defined by programming the Data Comparator Value Register, see *Data value comparator value registers* on page 3-55. This is a 32-bit register, however the value programmed into the register must match the comparison size.

2.8.2 Operation of data value comparators, ETMv3.2 and earlier

In ETM implementations before ETMv3.3, when the access size is less than the comparison size a comparison is still attempted, with a result that non-valid byte lanes from the data access on the processor are compared with the comparison value. For example, if a word comparison at address `0x2000` is programmed and the processor performs a byte access to address `0x2000`, a comparison is attempted. This comparison attempts to compare bits [31:8] of the access with the corresponding bits of the Data Comparator Value Register, despite the fact that these bits of the access are UNPREDICTABLE.

The way data is presented to the comparators differs between ETMv1 and later versions of the ETM:

In ETMv1 The data value presented to the comparator is the raw data from the data bus. So, for example, if a byte at address `0x2001` is accessed, it is compared with bits [15:8] of the Data Comparator Value Register, and the comparison is masked by bits [15:8] of the Data Comparator Mask Register.

This means that you have to consider the bottom two bits of the comparison address, in the Address Comparator Value Register, to determine how to program the data value comparator registers.

It is not possible to perform data value comparisons on bytes and halfwords in ranges.

For information about how the lower address bits and the endianness affect the way the ARM processor reads the data bus, see the appropriate memory interface chapter of your ARM core data sheet or *Technical Reference Manual*.

In ETMv2 to ETMv3.2

The data presented to the comparator is the data that is actually traced, that is rotated as necessary from its position on the data bus. In this case, if a byte at address `0x2001` is accessed, it is compared using bits [7:0] of the Data Comparator Value and Data Comparator Mask Registers.

This means the programming of the data value comparator registers is independent of the value in the Address Comparator Value Register.

Note

From ETMv3.1, address comparators support unaligned accesses. However, the comparison address must be aligned in all versions of the ETM versions, including those versions where the address comparators support unaligned accesses.

2.8.3 Operation of data value comparators, ETMv3.3 and later

To explain how data value comparisons work from ETMv3.3, this section describes the situations when the access value can match the comparison value. It describes using a single address comparator, and also using address range comparators. In each case it considers each of the different possible comparison sizes:

- byte
- halfword
- word.

For each comparison size it indicates what accesses can generate a match, and which accesses never match. It also describes any special programming requirements for the comparator registers to enable correct matching.

Data value matching with single address comparators

The accesses for which a match occurs are:

Comparison size = byte

A match occurs for the following accesses, if the access value matches the comparison value:

- a byte access to the comparison address
- a halfword or word access to the comparison address
- a halfword or word access to a lower address, where the access overlaps the comparison address.

To achieve this matching behavior you must:

- program the same value into all four bytes of the Data Comparator Value Register
- program the same mask value into all four bytes of the Data Comparator Mask Register.

Comparison size = halfword

A match occurs for the following accesses, if the access value matches the comparison value:

- A halfword access to the comparison address.
- A word access to the comparison address.
- A word access to ((comparison address) -2). This means that the most significant halfword of the access overlaps the comparison address, and is a halfword aligned access.

To achieve this matching behavior you must:

- program the same value into the top and bottom halfwords of the Data Comparator Value Register
- program the same mask value into the top and bottom halfwords of the Data Comparator Mask Register.

No other access can match. In particular, accesses that overlap the comparison address but that are not halfword aligned with that address do not match. For example, a word access to ((comparison address) -2) never matches.

Comparison size = word

A match occurs only if there is a word access to the comparison address and the access value matches the comparison value.

Constraints and rules for data value matching with single address comparators

The described operation of data matching with single address comparators can be summarized by two constraints and two rules.

The two constraints are:

- The alignment of the comparison address must correspond to the comparison size:
  - if the comparison size is halfword the address must be halfword aligned
  - if the comparison size is word the address must be word aligned.
- If the comparison size is byte or halfword the same values must be written into all bytes, or halfwords, of the Data Comparator Value Register and the Data Comparator Mask Register.

The two rules are:

- the comparator does not match if the access size is smaller than the comparison size
- the comparator does not match on accesses that are not aligned appropriately for the comparison size:
  - if the comparison size is halfword matches can only occur if the access is halfword aligned
  - if the comparison size is word matches can only occur if the access is word aligned.

Data value matching with address range comparators

An address range comparator is constructed from a pair of single address comparators. In particular, the address matching of an address range comparator depends on the address matching of the two single address comparators, and therefore follows the address matching behavior described in *Data value matching with single address comparators* on page 2-58. More information about the address matching of an address range comparator is given in *Address matching of an address range comparator*.

However, with address range comparators that are configured for data value matching, the data value matching is only based on the first of the single address comparators. This is the comparator that defines the lower address, the start address, of the address range. More information about this is given in *Data value matching of an address range comparator*.

The constraints and rules that apply to data value matching with address range comparators are given in *Constraints and rules for data value matching with address range comparators*.

Address matching of an address range comparator

An address range comparator take the *greater than or equal to* (>=) outputs from a pair of single address comparators. The address range comparator matches if both:

- the >= output from the low address comparator is HIGH, indicating a match
- the >= output from the high address comparator is LOW, indicating no match.

When a pair of single address comparators are used to form an address range comparator, most fields of the low address and high address Address Access Type Registers must be programmed with identical values, see *Address comparator registers* on page 3-50. This means that qualifiers that might prevent a match apply to both of the single comparators. For example, when you want an address range comparator to match only in Secure state, both comparators are programmed to match only on Secure accesses. This is done by setting bits [11:10] of both Address Access Type Registers to b10.

Data value matching of an address range comparator

When you are performing data value matching with an address range comparator, the data value matching is only performed as part of the operation of the low address comparator. There is no data value matching associated with the high address comparator. If the data value comparison for the low address comparator means that the comparator does not match then the result of the high address comparator is irrelevant. However, the rules for data value matching for address range comparators are slightly different to those for single address comparators, and are given in *Constraints and rules for data value matching with address range comparators*.

Constraints and rules for data value matching with address range comparators

The operation of data value matching with address range comparators can be summarized by two constraints and two rules.

The two constraints are:

- The alignment of the comparison addresses must correspond to the comparison size:
  - if the comparison size is halfword the addresses must be halfword aligned
  - if the comparison size is word the addresses must be word aligned.

- If the comparison size is byte or halfword the same values must be written into all bytes, or halfwords, of the Data Comparator Value Register and the Data Comparator Mask Register.

  Note

  The data value comparison is only defined for the low address comparator.

These constraints are the same as the constraints for data value matching with single address comparators.

The two rules are:

- The address range comparator does not match if the access size is different to the comparison size.
- The comparator does not match on accesses that are not aligned appropriately for the comparison size:
  - — if the comparison size is halfword matches can only occur if the access is halfword aligned
  - — if the comparison size is word matches can only occur if the access is word aligned.

  This rule also applies to data value matching with a single address comparator.

The first of these rules means that data value matching is more restricted with address range comparators than it is with single address comparators. For example, if the comparison size is configured as byte:

- with a single address comparators, word and halfword data accesses can match, as described in *Data value matching with single address comparators* on page 2-58
- with an address range comparator, word and halfword data accesses never match.

Note

All ETMs treat doubleword transfers as two separate word transfers. Therefore, an address range comparator with data value comparison configured to match on word accesses also matches on doubleword accesses.

2.8.4 Summary of alignment and endianness considerations for different ETM versions

Alignment and endianness considerations are different for different versions of the ETM, and are summarized in:

- Table 2-11 on page 2-62, for ETMv1.x
- Table 2-12 on page 2-62, for ETMv2.0 to ETMv3.2
- Table 2-13 on page 2-62, for ETMv3.3 and later.

Table 2-11 Alignment considerations in ETMv1.x

Watch address	Match value	Endianness	Mask	Value
Byte at 0x1000	0xAB	Little	0xFFFFFF00	0x000000AB
Byte at 0x1002	0xAB	Little	0xFF00FFFF	0x00AB0000
Byte at 0x1002	0xAB	Big	0xFFFF00FF	0x0000AB00
Byte in range	0xAB	Little	Not possible	Not possible

Table 2-12 Alignment considerations in ETMv2.0 to ETMv3.2

Watch address	Match value	Endianness	Mask	Value
Byte at 0x1000	0xAB	Little	0xFFFFFF00	0x000000AB
Byte at 0x1002	0xAB	Little	0xFFFFFF00	0x000000AB
Byte at 0x1002	0xAB	Big	0xFFFFFF00	0x000000AB
Byte in range	0xAB	Little	0xFFFFFF00	0x000000AB

Table 2-13 Alignment considerations in ETMv3.3 and later

Watch address	Match value	Endianness	Mask	Value
Byte at 0x1000	0xAB	Little	0x00000000	0xABABABAB
Byte at 0x1002	0xAB	Little	0x00000000	0xABABABAB
Byte at 0x1002	0xAB	Big	0x00000000	0xABABABAB
Byte in range	0xAB	Little	0x00000000	0xABABABAB

2.9 Instrumentation resources, from ETMv3.3

Instrumentation resources are introduced in ETMv3.3, and provide a simple, low-overhead method of controlling tracing. This is based on:

- The provision of up to four new event resources, Instrumentation resource 1 to Instrumentation resource 4. For more information, see *The Instrumentation resource event resources* on page 2-64.
- The introduction of new ARM and Thumb instructions to control these resources. These instructions can:
  - — Set a specified Instrumentation resource, for the current and following cycles.
  - — Clear a specified Instrumentation resource, for the current and following cycles
  - — Pulse a specified Instrumentation resource. This means the resource is set for the current cycle, cleared for the next cycle, and remains clear for following cycles. If the resource is already set then it remains set for the current cycle and is cleared from the next cycle.

  For more information, see *Instructions for controlling the Instrumentation resources* on page 2-64.
- Programming the filtering resources to take account of the Instrumentation resources. This is described in *Trace filtering* on page 2-18, and is summarized in Appendix A *ETM Quick Reference information*.

The number of Instrumentation resources that an ETM provides is IMPLEMENTATION DEFINED:

- an ETM implementation does not have to provide any Instrumentation resources
- a maximum of four Instrumentation resources can be implemented.

Bits [15:13] of the Configuration Code Extension Register specify the number of Instrumentation resources provided by the ETM implementation, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

In addition, bit [24] of the ETM Control Register can be used to control the availability of Instrumentation resource programming. If this bit is set to 1, the Instrumentation resources can only be programmed when the processor is in a privileged mode. For more information see *ETM Control Register* on page 3-20.

2.9.1 The Instrumentation resource event resources

The Instrumentation resource event resources provided from ETMv3.3 are described in Table 2-14.

Table 2-14 The instrumentation resource event resources

Resource type[a]	Index value[a]	Description of resource type
3'b001	8	Instrumentation resource 1
3'b001	9	Instrumentation resource 2
3'b001	10	Instrumentation resource 3
3'b001	11	Instrumentation resource 4

a. The Resource type is bits [6:4] of the 7-bit resource identifier, and the Index value is bits [3:0] of the identifier. Sometimes, the combined 7-bit resource identifier is called the Resource number.

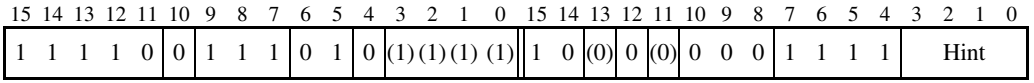
These event resources are also included in Table 3-81 on page 3-108 and Table A-1 on page A-2.

2.9.2 Instructions for controlling the Instrumentation resources

From ARM Architecture v7, both the ARM and Thumb instruction sets reserve twelve instructions for use as instrumentation instructions. These instructions are part of the Debug hint (`DBG`) part of the NOP-compatible hint space. The Thumb and ARM encodings of these instructions are:

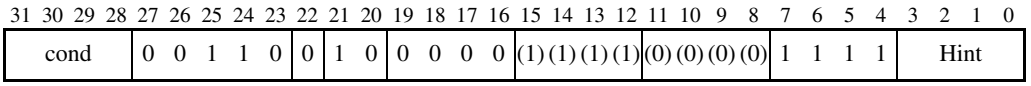
Encoding T1 ARMv7 (executes as NOP in ARMv6T2)

```
DBG<c> #<option>
```



Encoding A1 ARMv7 (executes as NOP in ARMv6K and ARMv6T2)

```
DBG<c> #<option>
```



For more information see the *ARM Architecture Reference Manual*.

In the `DBG` instruction, the value of the Hint field determines the instrumentation resource operation.

Hint field encodings for the DBG instruction instructions

Table 2-15 shows the hint field encodings for the DBG instrumentation instructions. These encodings are the same for the ARM and Thumb-2 instruction sets.

Table 2-15 Hint field encodings for the instrumentation instructions

Hint value	Effect of instruction
0x0	Set Resource 1
0x1	Set Resource 2
0x2	Set Resource 3
0x3	Set Resource 4
0x4	Clear Resource 1
0x5	Clear Resource 2
0x6	Clear Resource 3
0x7	Clear Resource 4
0x8	Pulse Resource 1
0x9	Pulse Resource 2
0xA	Pulse Resource 3
0xB	Pulse Resource 4

Note

The DBG hint instructions are defined in the ARMv7 architecture specification, see the *ARM Architecture Reference Manual*. This ETM Architecture Specification only defines the twelve field values given in Table 2-15 for use with ETM implementations.

2.9.3 Instrumentation resource behavior when tracing parallel execution

If multiple instrumentation resource instructions are executed in parallel, the instrumentation resource must behave as if the instructions were executed in sequence. However, if a resource is activated, by a set or pulse instruction, at any point in the resulting sequence then it must be active for the current cycle. Table 2-16 shows examples of this, for the case where two instructions are executed in parallel.

Table 2-16 Instrumentation resource parallel execution examples, for two instructions

Equivalent instruction sequence		Instrumentation resource behavior[a]
First instruction	**Second instruction**	
No effect	Set	Set on current cycle, continues[b] on future cycles.
No effect	Clear	Hold previous state for current cycle, clear[c] from next cycle.
Clear	Set	Set on current cycle, continues[b] on future cycles.
Clear	Pulse	Set on current cycle, clear[c] from next cycle.
No effect	Pulse	Set on current cycle, clear[c] from next cycle.

a. In the descriptions of resource behavior, the *current cycle* is the cycle when the parallel execution is performed.
b. *Continues* means that the resource remains set until another instruction clears the resource. The instruction immediately following the parallel execution might clear the resource. In this case the resource is only active during the current cycle.
c. Unless the instruction decoded in the next cycle sets the resource.

2.10 Trace port clocking modes

The ETM supports several clocking modes that enable the trace port to operate at a different speed from that of the core. See *Target connector pinouts* on page 8-3 for more information. Logic to implement these modes might be external to the ETM. See the relevant *Technical Reference Manual* for more information.

2.10.1 ETMv1 and ETMv2 behavior

In ETMv1.x and ETMv2.x, the trace port protocol assumes that the trace port runs at the same frequency as the core. This means that clocking modes in these devices are bandwidth invariant, because the number of pins in the trace port is varied to preserve the trace port bandwidth. The trace capture device must be aware of the mode in use and must reconstruct the trace to appear as if trace was captured at full speed from a normal port.

The modes are:

Normal The trace port runs at core clock speed.

Demultiplexed

The trace port runs at half the core clock speed over twice the number of pins.

Multiplexed The trace port runs at twice the core clock speed over half the number of pins.

Additionally, the trace port clock can be selected to run:

- at the same speed as the trace port, capturing off the rising edge
- at half the speed of the trace port, capturing off both clock edges.

Trace must always be captured off both edges in multiplexed mode.

The behavior is selected by two fields in the ETM Control Register, see *ETM Control Register* on page 3-20. The two fields are bit [13], Half-rate clocking, and bits [17:16], Port mode, as shown in Table 2-17.

Table 2-17 Clocking, port mode, port speed, and data pins in ETMv1 and ETMv2

Half-rate clocking	Port mode	Name	TRACECLK edge	Trace port clock speed: ETM clock speed	Trace port data rate: ETM clock speed	Data port pins: ETM data pins
0	b00	Normal	Rising	1:1	1:1	1:1
0	b01	Multiplexed	Both	1:1	2:1	1:2
0	b10	Demultiplexed	Rising	1:2	1:2	2:1
1	b00	Normal, half-rate clocking	Both	1:2	1:1	1:1
1	b10	Demultiplexed, half-rate clocking	Both	1:4	1:2	2:1

2.10.2 ETMv3 behavior

In ETMv3.x, the trace port protocol enables the trace port to run at a different speed from that of the core. Modes in these devices are port size invariant in that the number of pins in the trace port remains constant as the trace port bandwidth changes. The trace capture device is unaware of the mode in use.

In ETMv3.x, the trace is collected on both clock edges in all modes except dynamic mode. Dynamic mode is designed for capture on-chip using the ETM clock instead of **TRACECLK**.

The two fields used in ETMv1 and ETMv2 are combined into a single field, bits [13, 17:16], Port mode[2:0], as shown in Table 2-18.

Table 2-18 Port mode, port speed and data pins in ETMv3

Port mode [2:0]	Name	TRACECLK edge	Trace port clock speed: ETM clock speed	Trace port data rate: ETM clock speed	Data port pins: ETM data pins
b000	Dynamic, for capture on-chip	Not used	-	1:1	1:1
b001	2:1	Both	1:1	2:1	1:1
b010	Reserved	-	-	-	-
b011	IMPLEMENTATION DEFINED	Both	-	IMPLEMENTATION DEFINED	1:1
b100	1:1	Both	1:2	1:1, or asynchronous	1:1
b101	1:3	Both	1:6	1:3	1:1
b110	1:2	Both	1:4	1:2	1:1
b111	1:4	Both	1:8	1:4	1:1

The IMPLEMENTATION DEFINED encoding is available for non-standard ratios, such as 2:3.

2.11 Considerations for advanced cores, ETMv2 and later only

As far as possible, the ETM presents the view that all instructions and data transfers occur sequentially. However, this is not always possible where instructions or data transfers occur in parallel or out-of-order. This section contains rules for dealing with nonsequential behavior. These are all considerations that affect ARM10 family processors and later. The precise behavior of any ETM is IMPLEMENTATION DEFINED.

2.11.1 Parallel execution

The ETM must choose a particular stage in the core pipeline from which to trace instructions. This is chosen to be as close as possible to the program order of the instructions, without tracing instructions that are fetched but not executed.

The two types of parallel execution are:

Parallel instruction execution

This is where more than one instruction is executed in a single cycle.

In ETMv2.x, parallel instruction execution is supported, provided that only one of the instructions is capable of transferring data. The core cannot execute multiple data transfer instructions in parallel. This restriction is because the pin protocol makes this assumption.

From ETMv3, more general parallel instruction execution is supported.

Parallel data transfers

In cores with a 64-bit data bus, two 32-bit quantities might be transferred in a single cycle. This generally occurs only with *Load/Store Multiple* (`LSM`) instructions, see *Definitions* on page 4-21 for more information about these instructions.

Rules for parallel execution

Note

In this subsection, *item* refers to an object that is traced, either an instruction or data. When both instructions and data are being traced, an instruction and its associated data are separate items.

For either type of parallel execution, a small amount of extra trace is possible, but effects on the long-term state must be minimized. The following rules apply:

- In applying these rules, an instruction item must be considered as occurring before any associated data item. For example, if the trace stop control is a data address comparator, and the trace start/stop block is active before an instruction with data that matches this comparator, the trace start/stop block is active for the instruction. This is because the data comparison is considered after the instruction is traced.
- The trace start/stop block must view the instructions and data transfers as executing in the order in which they would be traced, regardless of whether tracing is enabled. An example of this ordering is given in Example 2-2 on page 2-70.

For each cycle, there are three situations to consider:

1. Only a start address matches. In this case, the start/stop block must be active on this cycle, and remains active until a stop address is encountered.
2. Only a stop address matches, when the start/stop block is already active. In this case:
   - if the stop address is the first item to be executed in this cycle then the start/stop block inactive on this cycle
   - if the stop address is not the first item then the start/stop block must be active on this cycle, to trace the items before the stop address.

   The start/stop block is inactive at the end of this cycle, and remains inactive until a start address is encountered.
3. Both the start address and the stop address match. In this case:
   - If the start address occurs before the stop address, the start/stop block must be active on this cycle, and inactive at the end of the cycle. It then remains inactive until a start address is encountered.
   - If the stop address occurs before the start address, the start stop block must be active on this cycle, and remains active after the cycle until another stop address is encountered.

This behavior means that a single address comparator must perform simultaneously a comparison for each instruction or data transfer, so that it can match or not match each item individually.

- If instructions would have been traced if they had been executed sequentially then they must be traced when executed in parallel. This applies to the **TraceEnable** include/exclude regions and to the trace start/stop block. Other instructions executed in the same cycle might also be traced as a result, but no trace must be lost.

  For example, consider two instructions executed in parallel, one of which causes a selected single address comparator to match. If **TraceEnable** is in include mode, the matching instruction must be traced, and the other might be traced. However if **TraceEnable** is in exclude mode, the non-matching instruction must be traced, and the other might be traced.

- **ViewData** must trace each data transfer if it would have been traced had the instructions been executed sequentially. This applies to the include/exclude regions.

  Note

  This rule might be reviewed if the ETM specification is extended to cores capable of executing multiple data transfer instructions in parallel.

- Any resource, when viewed as an event, must be active for the entire cycle if it matched for any instruction executed in that cycle. For example, if the trace start/stop resource is used as the enabling event of **ViewData**, but is logically active for only the first of two instructions executed in a cycle, the second instruction must have its data traced (assuming the include/exclude regions match).

Example 2-2 Trace Start/Stop block ordering of parallel instructions

Consider the case where these two instructions are executed in parallel:


```
LDRD  r4,  [r1]
LDR   r10, [r2]
```

The Trace Start/Stop block must behave as if the `LDRD r4, [r1]` is executed first, followed by `LDR r10, [r2]`. The means the block must behave as if the trace order is:

```
Instruction(LDRD r4)
Data loaded into r4, from address indicated by r1
Data loaded into r5, from (address indicated by r1) + 4
Instruction(LDR r10)
Data loaded into r10, from address indicated by r2
```

2.11.2 Independent load/store unit

If the core has an independent load/store unit, capable of continuing to transfer data values for an earlier instruction after later instructions have been executed, the later instructions are said to have executed underneath the data instruction. The following rules apply:

- A data address comparator that matches the data transfer must match when the transfer occurs, even if a later instruction is being executed at the same time.
- If **ViewData** is conditioned on instruction address comparators, a match on the instruction address must apply to all data corresponding to that instruction, regardless of whether another instruction has been executed since.
- If a data instruction has been traced, the ETM might require that all instructions executed underneath the data instruction are traced.

  Note

  The ETMv2.x and later protocols assume this.

- If **ViewData** becomes active because an instruction address comparator matches the address of an instruction executed underneath a data instruction, the ETM might trace extra data corresponding to the data instruction, even though it corresponds to a different instruction address.
- In addition to the specific cases already listed, a small number of additional instructions might be traced when *both* of these conditions occur together:
  - multiple instructions are executed on the same cycle
  - the start/stop resource matches for some *but not all* of the instructions.

  For example, if two instructions are executed in one cycle and:
  - the start/stop resource matches only the first instruction
  - the include region matches only the second instruction

  then both instructions are traced, even though neither instruction would have been traced if they had been executed sequentially.

2.11.3 Consequences of parallel execution on counters

Although some operations can be performed in parallel, the ETM counter can decrement only once every cycle. The only case where this might be a problem is where the OR of two single address comparators is used and they both match on the same cycle, for example, with a predicted branch and its target.

2.11.4 Consequences of parallel execution on the sequencer

If the sequencer receives multiple transition requests in the same cycle, no transitions take place and the sequencer remains in the original state. The ETM might have multiple transition requests in a cycle where instructions are executed in parallel. You must be aware of this behavior when programming the sequencer. There is a work-around for simple events, as shown in Example 2-3.

Example 2-3 Programming for parallel events

Consider the following transitions:

- transition from state 1 to state 2 based on event A
- transition from state 2 to state 3 based on event B.

To effect these transitions where A and B can occur in the same cycle, you must program the sequencer as follows:

- Program the transition from state 1 to state 2 (register `0x60`) to occur on event (A & !B).
- Program the transition from state 2 to state 3 (register `0x62`) to occur on event B.
- Program the transition from state 1 to state 3 (register `0x65`) to occur on event (A & B).

Programming the ETM sequencer in this way ensures the correct handling of simultaneous occurrences of event A and event B.

2.12 Supported standard configurations in ETMv1

ETMv1 specifies four standard configurations. They are described in this section.

2.12.1 Choosing a configuration

The choice of configuration is largely cost-based, because it depends on:

- the amount of silicon and pins that you want to use
- the degree of data trace required.

A small ETM is adequate for instruction trace. However, data tracing is less satisfactory with a small configuration, because the small FIFO and lack of support for the 21-pin interface reduces the data that can be traced without overflowing. The only implementations of the small configuration are in ETM7 and ETM9.

A medium-sized ETM gives reasonable data trace under most conditions. It is the configuration chosen by most users of ETM7.

Mediumplus is a medium-sized ETM configuration with a large FIFO. This configuration is only supported in ETM9 and is the ETM9 configuration chosen by most users.

The large ETM adds extensive triggering facilities and an enlarged FIFO. The large FIFO helps to smooth out short bursts of data trace.

Not all configurations are available for all implementations. Any future ETMv1 implementations are likely to be available only in the Mediumplus configuration.

2.12.2 ETM7 supported configurations

Table 2-19 shows the three standard configurations for ETM7.

Note

ETM7 is not available in the Mediumplus configuration.

Table 2-19 ETM7 configurations

Resource description	Small	Medium	Large
Pairs of address comparators	1	4	8
Data value comparators	0	2	8
Memory map decoders	4	8	16
Counters	1	2	4
Sequencer present	No	Yes	Yes

Table 2-19 ETM7 configurations (continued)

Resource description	Small	Medium	Large
External inputs	2	4	4
External outputs	0	1	4
FIFOFULL present	Yes[a]	Yes[a]	Yes
FIFO depth	10	20	45
Port size[b]	4/8	4/8/16	4/8/16

a. Not available in ETM7 Rev 0. In ETM7 Rev 0, **FIFOFULL** is supported only in the Large configuration.
b. Software-selectable using the ETM Control Register, `0x00`.

2.12.3 ETM9 supported configurations

Table 2-20 shows the four standard configurations for ETM9.

Table 2-20 ETM9 configurations

Resource description	Small	Medium	Mediumplus	Large
Pairs of address comparators	1	4	4	8
Data value comparators	0	2	2	8
Memory map decoders	4	8	8	16
Counters	1	2	2	4
Sequencer present	No	Yes	Yes	Yes
External inputs	2	4	4	4
External outputs	0	1	1	4
FIFOFULL present	Yes[a]	Yes[a]	Yes[a]	Yes
FIFO depth	9	18	45	45
Port size[b]	4/8	4/8/16	4/8/16	4/8/16

a. Not available in ETM9 Rev 0. In ETM9 Rev 0, **FIFOFULL** is supported only in the Large configuration.
b. Software-selectable using the ETM Control Register, `0x00`.

2.13 Supported configurations from ETMv2

The *Technical Reference Manual* for each ETM includes a description of the supported configurations.

2.14 Behavior when non-invasive debug is disabled

Some systems support security extensions that enable non-invasive debug to be disabled. Sometimes a signal called **NIDEN**, Non Invasive Debug ENable, is used to disable or enable ETM functionality. Systems do not have to support the Security Extensions to implement this functionality.

When non-invasive debug is disabled, the ETM behaves as if the processor has entered a prohibited region. For more information, see *Behavior while tracing is prohibited* on page 2-19. The following additional restrictions apply:

- Whether the branch packet is output is IMPLEMENTATION SPECIFIC.
- All trace in the ETM FIFO must be output.
- Trigger generation is disabled.
- The trace prohibited resource, if supported, is HIGH.
- The security state of the processor is considered Secure, so the Non-secure resource, if supported, is LOW.
- External inputs, extended external inputs, and Memory Map Decoders must be ignored.
- Other resources such as counters, the sequencer and external outputs, stop operating and are held in their current state. Some ETM implementations might drive the external outputs LOW.
- It is IMPLEMENTATION DEFINED if the cycle counter continues to count.

As defined in the *CoreSight Architecture Specification*, the effect of the timing of disabling non-invasive debug is imprecise. Therefore, tracing might continue after non-invasive debug is disabled, and may take time to re-enable when non-invasive debug is re-enabled.

When non-invasive debug is disabled, the Authentication Status Register represents this. For more information, see *Authentication Status Register (AUTHSTATUS), ETMv3.2 and later* on page 3-92.

When non-invasive debug is disabled, the ETM programmer's model behaves normally.

ARMv7 processors must implement the **NIDEN** functionality, and ETMs that are connected to ARMv7 processors must implement this functionality.

Chapter 3
Programmer's Model

This chapter describes the configuration registers that you can program to set up and control the ETM. It contains the following sections:

3.1 About the programmer's model

Where a register bit is assigned for use from a particular version of the ETM architecture onwards, then in previous versions of the architecture:

- The read value of the bit is UNKNOWN, unless the assignment of the bit specifies it as backwards-compatible with earlier versions of the ETM architecture. If the assignment of the bit specifies it as backwards-compatible you can ignore the ETM architecture version when interpreting the bit.
- The bit must be written as zero.

The register bit assignment tables include a column that shows the first version of the ETM architecture that uses that bit assignment, unless otherwise specified.

Any of the following causes UNPREDICTABLE behavior:

- writing to a reserved register
- writing to a read-only register
- writing a nonzero value to reserved bits in a register
- using a reserved encoding in a register field.

3.2 Programming and reading ETM registers

There are three methods of access to the ETM registers, these are described in:

- *JTAG access*
- *Coprocessor access, ETMv3.1 and later* on page 3-4.
- *Memory-mapped access, ETMv3.2 and later* on page 3-6.

It is IMPLEMENTATION DEFINED which interfaces are supported. Concurrent access from multiple interfaces is supported.

The description of each method of access includes details of any restrictions that apply to that method. However you must also see *Restrictions on the type of access to ETM registers* on page 3-7 for information about restrictions that apply to all three methods. ETM register access models, for different versions of the ETM architecture, are described in *ETM register access models* on page 3-7.

3.2.1 JTAG access

The JTAG interface is an extension of the ARM TAP controller, and is assigned scan chain number 6. The scan chain consists of a 40-bit shift register comprising:

- a 32-bit data field
- a 7-bit address field
- a read/write bit.

Only registers `0x00-0x7F` can be accessed using JTAG access. The general arrangement of the ETM JTAG registers is shown in Figure 3-1.

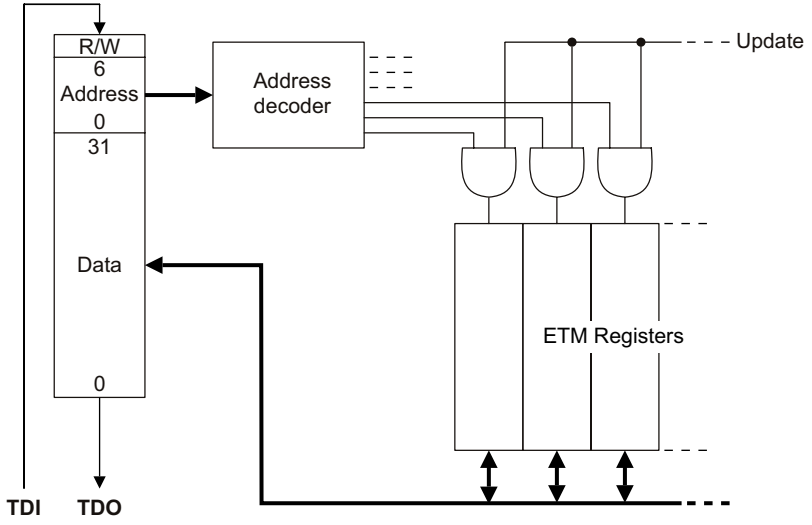


Figure 3-1 ETM JTAG structure

The data to be written is scanned into the 32-bit data field, the address of the register into the 7-bit address field, and a 1 into the read/write bit. A register is read by scanning its address into the address field and a 0 into the read/write bit. The 32-bit data field is ignored.

A read or a write takes place when the TAP controller enters the UPDATE-DR state.

Restricting JTAG access

Debugger access to the ETM registers can be made read-only by setting bit [22] of the ETM Control Register, `0x00`. This bit can only be set from software. This bit is not supported in all implementations, see *ETM Control Register* on page 3-20. Tools can determine if a non-JTAG interface is present by reading bit [27] of the Configuration Control Register, see *ETM Configuration Code Register* on page 3-29.

3.2.2 Coprocessor access, ETMv3.1 and later

Provision of a coprocessor interface for register access is optional in ETMv3.1 and later. This enables you to use the ETM as an extended breakpoint unit to test for unit failure while testing multiple devices. The coprocessor access also means that you do not have to program each device individually by connecting a probe to each device. You can do the following without external hardware:

- program the ETM
- collect trace, in conjunction with *Embedded Trace Buffer* (ETB)
- examine the buffer in conjunction with ETB

This section describes the changes to the programmer's model, in the following subsections:

- *Coprocessor models*
- *Restricting coprocessor access* on page 3-6
- *Determination of support* on page 3-6.

Coprocessor models

Where a co-processor model is supported, all the accessible ETM registers are mapped to a single coprocessor. All instructions in Coprocessor 14 with Opcode_1 equal to 1 are reserved for ETM use.

There are two coprocessor models, described in the following sub-sections:

- *Limited register set model, ETMv3.1 and ETMv3.2 only*
- *Full access model, ETMv3.3 and later* on page 3-5.

See *Behavior of coprocessor accesses* on page 3-5 for information that applies to both models.

Limited register set model, ETMv3.1 and ETMv3.2 only

The coprocessor model provided in ETMv3.1 and ETMv3.2 provides access to ETM registers `0x00-0x7F` only. See *The ETM registers* on page 3-11 for a list of all the ETM registers, in register-number order.

The instructions to read and write the ETM registers are as follows:

```
MRC <p14>, 1, <Rd>, c0, reg[3:0], reg[6:4]
MCR <p14>, 1, <Rd>, c0, reg[3:0], reg[6:4]
```


In these instructions, `reg[6:0]` is the ETM register number.

These instructions have CRn equal to c0 and the register number encoded in Opcode_2 and CRm.

Full access model, ETMv3.3 and later

From ETMv3.3, the coprocessor model provides access to all of the ETM registers, including the CoreSight management registers and the OS Save/Restore registers. See *The ETM registers* on page 3-11 for a list of all the ETM registers, in register-number order.

Note

When accessed through the coprocessor interface, the Lock Access and Lock Status Registers (registers `0x3EC` and `0x3ED`) read-as-zero. You do not have to set a lock to access the ETM registers through the coprocessor interface.

The instructions to read and write the ETM registers are as follows:

```
MRC <p14>, 1, <Rd>, reg[9:7], reg[3:0], reg[6:4]
MCR <p14>, 1, <Rd>, reg[9:7], reg[3:0], reg[6:4]
```

In these instructions, `reg[9:0]` is the ETM register number.

Figure 3-2 shows the mapping between the bits of the ETM register number and the fields of the CP14 instruction.

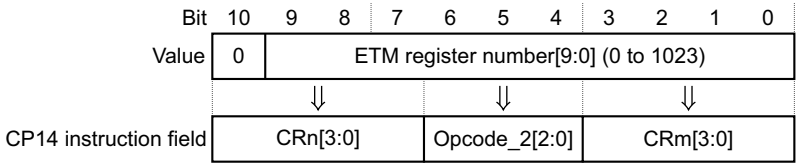


Figure 3-2 Mapping from register number to CP14 instruction fields

Behavior of coprocessor accesses

This information applies to both of the coprocessor models. In other words it applies to all ETM register accesses through Coprocessor 14, in ETMv3.1 and later.

Coprocessor access to the ETM registers is only permitted when the ARM processor is in a privileged mode. An attempt to read or write an ETM register using coprocessor instructions while the processor is in User mode results in an Undefined Instruction exception. Coprocessor accesses initiated by a debug tool when the processor is halted in Debug state are always privileged regardless of the state of the CPSR.

A read from a nonexistent register returns zero, and a write to a nonexistent register is ignored. The tools must determine what registers are valid from the programmer's model. For more information on access permissions see *Access permissions for ETM registers* on page 3-128.

Note

This behavior is different to coprocessor access to debug registers, where attempting to access a nonexistent register usually results in an Undefined Instruction exception.

Restricting coprocessor access

Software access to the ETM registers can be made read-only by setting bit [23] of the ETM Control Register, `0x00`. This bit can only be set by the debugger. See *ETM Control Register* on page 3-20.

Determination of support

To determine whether coprocessor access is supported, read the ETM ID Register:

```
MRC p14, 1, <Rd>, c0, c9, 7
```

If no Undefined Instruction exception is generated and a nonzero value is returned, then coprocessor access is supported.

Behavior of other CP14 accesses with Opcode_1 equal to 1

All instructions in Coprocessor 14 with Opcode_1 equal to 1 are reserved for ETM use. However, only a limited range of these instructions are used for ETM access. Details are given in the following sub-sections:

- *ETMv3.1 and v3.2*
- *ETMv3.3 and later*.

ETMv3.1 and v3.2

Only instructions with a CRn value of c0 are used for ETM register accesses. `MRC` and `MCR` accesses to Coprocessor 14 with a CRn value greater than 4'b0000 are:

- UNDEFINED in user mode
- UNPREDICTABLE in privileged modes.

ETMv3.3 and later

Only instructions with CRn values from 4'b0000 to 4'b0111 are used for ETM register accesses. `MRC` and `MCR` accesses to Coprocessor 14 with a CRn value of 4'b1000 or greater are:

- UNDEFINED in user mode
- UNPREDICTABLE in privileged modes.

3.2.3 Memory-mapped access, ETMv3.2 and later

ETMv3.2 and later provides optional memory-mapped access. This is usually used in a CoreSight system, and provides all the benefits of coprocessor access, along with other benefits described in *The CoreSight Architecture Specification*.

Memory-mapped access provides a 4KB address space, and can access all registers. Each register occupies 4 bytes, making a total of 1024 registers available in this way. For example, register `0x080` is at offset `0x200` from the base address of the ETM.

A read from a nonexistent register returns zero, and a write to a nonexistent register is ignored.

The ETM can distinguish between memory-mapped accesses from on-chip software and memory-mapped accesses from a debugger, for example by using the CoreSight *Debug Access Port* (DAP). Software accesses require the ETM to be first unlocked using the lock registers described in *Lock registers, ETMv3.2 and later* on page 3-90.

3.2.4 Restrictions on the type of access to ETM registers

In *The ETM registers* on page 3-11, Table 3-3 on page 3-11 shows the access type, read-only, write-only, or read/write, of each ETM register. Debug software must take account of these access types. Behavior is UNPREDICTABLE if you attempt a read access to a write-only register, or a write access to a read-only register. This is true for all three methods of accessing the registers (JTAG, coprocessor and memory-mapped access).

3.2.5 ETM register access models

Table 3-1 summarizes the more common ETM register access models, with an indication of the situations when they are likely to be appropriate.

Table 3-1 Typical ETM register access implementations

Register interfaces	ETM versions	Access requirements
JTAG only	All versions	Implemented where access is required only to registers `0x000` to `0x07F`.
JTAG and Coprocessor	From ETMv3.1	Implemented where access is required only to registers `0x000` to `0x07F`, and software access to these registers is required.
Coprocessor only	From ETMv3.1	Debugger access is through an ARM Debug Interface v5[a]. In addition, software access is possible.
JTAG and Memory-Mapped	From ETMv3.2	Implemented where access is required only to registers `0x000` to `0x07F`, and software access to these registers is required.
Memory-Mapped only	From ETMv3.2	Debugger access is through an ARM Debug Interface v5[a]. In addition, software access is possible.

a. For more information see the *ARM Debug Interface v5 Architecture Specification*.

For information about the access controls that can apply to ETM register accesses see *Access permissions for ETM registers* on page 3-128.

When power down support is implemented as described in *Power-down support, ETMv3.3 and later* on page 3-119, a JTAG interface to the ETM is not permitted. Access to the ETM registers from an external debugger must use the ARM Debug Interface v5. For more information, see the *ARM Debug Interface v5 Architecture Specification*.

3.2.6 Synchronization of ETM register updates

Software running on the processor can program the debug registers through at least one of:

- a CP14 coprocessor interface
- the memory-mapped interface, if it is implemented.

It is IMPLEMENTATION DEFINED which interfaces are implemented.

For the CP14 coprocessor interface, the following synchronization rules apply:

- All changes to CP14 ETM registers that appear in program order after any explicit memory operations are guaranteed not to affect those memory operations.
- Any change to CP14 ETM registers is guaranteed to be visible to subsequent instructions only after one of:
  - performing an ISB operation
  - taking an exception
  - returning from an exception.

  However, the following rules apply to coprocessor ETM register accesses:
  - when an `MRC` instruction directly reads a register using the same register number as was used by an `MCR` instruction to write it, the `MRC` is guaranteed to observe the value written, without requiring any context synchronization between the `MCR` and `MRC` instructions
  - When an `MCR` instruction directly writes a register using the same register number as was used by a previous `MCR` instruction to write it, the final result is the value of the second `MCR`, without requiring any context synchronization between the two `MCR` instructions.

  This is important when changing the value of the programming bit in the ETM Control Register. After writing to the ETM Control Register to change the value of the programming bit you must make at least one read of the ETM Status Register before you program any other registers. For more information see *Use of the Programming bit* on page 3-17. You must perform an ISB between writing to the ETM Control register and reading the ETM Status register.

  ARM recommends that, after programming the ETM registers, you always execute an `ISB` instruction to ensure that all updates are committed to the ETM before you restart normal code execution.

For the memory-mapped interface, the following synchronization rules apply:

- Changes to memory-mapped ETM registers that appear in program order after an explicit memory operation are guaranteed not to affect that previous memory operation only if the order is guaranteed by the memory order model or by the use of a DMB or DSB operation between the memory operation and the register change.
- A DSB operation causes all writes to memory-mapped ETM registers appearing in program order before the DSB to be completed.

However, the following rules apply to memory-mapped ETM register accesses:

— when a load directly reads a register using the same address as was used by a store to write it, the load is guaranteed to observe the value written, without requiring any context synchronization between the store and load

— When a store directly writes a register using the same address as was used by a previous store 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, after programming the ETM registers, you always execute a `DSB` instruction followed by an `ISB` instruction, to ensure that all updates are committed to the ETM before you restart normal code execution.

Some memory-mapped ETM registers are not idempotent for reads or writes. Therefore, the region of memory occupied by the ETM 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. Memory used for memory-mapped ETM registers must have the Strongly-ordered or Device attribute, otherwise the effects of accesses to the registers are UNPREDICTABLE.

Synchronization between register updates made through the external debug interface and updates made by software running on the processor 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.

3.3 CoreSight support

CoreSight is a system-level debug and trace solution that enables debug and trace components to share resources and work together. It defines a Visible Component Architecture that specifies requirements of all CoreSight components that are visible to development tools. The following sections describe features of the Visible Component Architecture:

- *Programmer's model requirements*
- *Topology detection requirements*.

See the *CoreSight Architecture Specification* for more information about the CoreSight Architecture.

3.3.1 Programmer's model requirements

The programmer's model specifies that the registers of each component are memory-mapped in a 4KB region. The top 256 bytes of this space, registers `0x3C0-0x3FF`, are reserved for management registers that must be present in all CoreSight Components. These registers are documented in *Detailed register descriptions* on page 3-20.

See *Memory-mapped access, ETMv3.2 and later* on page 3-6 for more information about memory-mapped access to the ETM registers.

3.3.2 Topology detection requirements

All ETMs implement the logical interfaces shown in Table 3-2. These logical interfaces must implement registers to support topology detection, as described in the *CoreSight Architecture Specification*.

Table 3-2 ETM logical interfaces

Port	Direction	Number
Trace output	Master	1
Core interface	Slave	1 + bits [14:12] of System Configuration Register (`0x05`)
External output	Master	Bits [22:20] of ETM Configuration Code Register (`0x01`)
External input	Slave	Bits [19:17] of ETM Configuration Code Register (`0x01`)
Trigger output	Master	1

Registers `0x380-0x3BF` are reserved for topology detection and integration registers, and use of these registers for this purpose is IMPLEMENTATION DEFINED.

3.4 The ETM registers

Table 3-3 shows the ETM registers, in order.

Table 3-3 ETM registers summary

Register[a]	Function	Version[b]	Type	Description
0x000-0x0BF, ETM Trace Registers[c]				
0x000	ETM Control	v1.0	R/W	See *ETM Control Register* on page 3-20
0x001	ETM Configuration Code	v1.0	RO	See *ETM Configuration Code Register* on page 3-29
0x002	Trigger Event	v1.0	WO[d]	See *Trigger Event Register* on page 3-31
0x003	ASIC Control	v1.0	WO[d]	See *ASIC Control Register* on page 3-32
0x004	ETM Status	v1.1 to v3.0	RO	See *ETM Status Register, ETMv1.1 and later* on page 3-33
		v3.1	R/W	
0x005	System Configuration	v1.2	RO	See *System Configuration Register, ETMv1.2 and later* on page 3-35
TraceEnable configuration:				
0x006	Trace Start/Stop Resource Control	v1.2	WO[d]	See *Trace Start/Stop Resource Control Register, ETMv1.2 and later* on page 3-37
0x007	TraceEnable Control 2	v1.2	WO[d]	See *TraceEnable Control 2 Register, ETMv1.2 and higher* on page 3-38
0x008	TraceEnable Event	v1.0	WO[d]	See *TraceEnable Event Register* on page 3-41
0x009	TraceEnable Control 1	v1.0	WO[d]	See *TraceEnable Control 1 Register* on page 3-39
FIFOFULL configuration:				
0x00A	FIFOFULL Region	v1.0	WO[d]	See *FIFOFULL Region Register* on page 3-42
0x00B	FIFOFULL Level	v1.x only	WO	See *FIFOFULL Level Register* on page 3-43
		v2.0	R/W	
ViewData configuration:				
0x00C	ViewData Event	v1.0	WO[d]	See *ViewData Event Register* on page 3-45

Table 3-3 ETM registers summary (continued)

Register[a]	Function	Version[b]	Type	Description
0x00D	ViewData Control 1	v1.0	WO[d]	See *ViewData Control 1 Register* on page 3-46
0x00E	ViewData Control 2	v1.0	WO[d]	See *ViewData Control 2 Register* on page 3-47
0x00F	ViewData Control 3	v1.0	WO[d]	See *ViewData Control 3 Register* on page 3-48
Address comparators:				
0x010-0x01F	Address Comparator Value 1-16	v1.0	WO[d]	See *Address Comparator Value Registers* on page 3-50
0x020-0x02F	Address Access Type 1-16	v1.0	WO[d]	See *Address Access Type Registers* on page 3-51
Data value comparators: **Note** Only the even-numbered registers can be implemented. The odd-numbered registers are reserved.				
0x030-0x03E, even	Data Comparator Value 1-16	v1.0	WO[d]	See *Data value comparator value registers* on page 3-55
0x031-0x03F, odd	-	-	-	Reserved
0x040-0x04E, even	Data Comparator Mask 1-16	v1.0	WO[d]	See *Data value comparator mask registers* on page 3-56
0x041-0x04F, odd	-	-	-	Reserved
Counters:				
0x050-0x053	Counter Reload Value 1-4	v1.0	WO[d]	See *Counter Reload Value Registers* on page 3-59
0x054-0x057	Counter Enable 1-4	v1.0	WO[d]	See *Counter Enable Registers* on page 3-59
0x058-0x05B	Counter Reload Event 1-4	v1.0	WO[d]	See *Counter Reload Event Registers* on page 3-61
0x05C-0x05F	Counter Value 1-4	v1.0 to v3.0	RO	See *Counter Value Registers* on page 3-61
		v3.1	R/W	

Table 3-3 ETM registers summary (continued)

Register[a]	Function	Version[b]	Type	Description
Sequencer:				
0x060-0x065	Sequencer State Transition Event	v1.0	WO[d]	See *Sequencer State Transition Event Registers* on page 3-63
0x066	-	-	-	Reserved
0x067	Current Sequencer State	v1.0 to v3.0	RO	See *Current Sequencer State Register* on page 3-64
		v3.1	R/W	
0x068-0x06B	External Output Event 1-4	v1.0	WO[d]	See *External Output Event Registers* on page 3-65
Context ID comparators:				
0x06C-0x06E	Context ID Comparator Value	v2.0	WO[d]	See *Context ID Comparator Value Registers* on page 3-66
0x06F	Context ID Comparator Mask	v2.0	WO[d]	See *Context ID Comparator Mask Register* on page 3-67
Other ETM Trace[c] registers:				
0x070-0x077	Implementation-specific		WO[d]	See *implementation specific registers* on page 3-68
0x078	Synchronization Frequency	v2.0	WO[d] or RO[e]	See *Synchronization Frequency Register, ETMv2.0 and later* on page 3-69
0x079	ETM ID	v2.0	RO	See *ETM ID Register, ETMv2.0 and later* on page 3-71
0x07A	Configuration Code Extension	v3.1	RO	See *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76
0x07B	Extended External Input Selection	v3.1	WO[d]	See *Extended External Input Selection Register, ETMv3.1 and later* on page 3-77
0x07C	Trace Start/Stop Embedded ICE Control	v3.4	WO[d]	See *Trace Start/Stop EmbeddedICE Control Register, ETMv3.4 and later* on page 3-78
0x07D	Embedded ICE Behavior Control	v3.4	WO[d]	See *EmbeddedICE Behavior Control Register, ETMv3.4 and later* on page 3-79

Table 3-3 ETM registers summary (continued)

Register[a]	Function	Version[b]	Type	Description
0x07E-0x07F	-	-	-	Reserved.
0x080	CoreSight Trace ID	v3.2	R/W	See *CoreSight Trace ID Register, ETMv3.2 and later* on page 3-80
0x081-0x0BF	-	-	-	Reserved.
0x0C0-0x0C5, ETM Management Registers[c]				
Operating system save and restore registers:				
0x0C0	OS Lock Access	v3.3	WO	See *OS Lock Access Register (OSLAR), ETMv3.3 and later* on page 3-81
0x0C1	OS Lock Status	v3.3	RO	See *OS Lock Status Register (OSLSR), ETMv3.3 and later* on page 3-82
0x0C2	OS Save/Restore	v3.3	R/W	See *OS Save and Restore Register (OSSRR), ETMv3.3 and later* on page 3-84
Other ETM Management Registers:				
0x0C3-0x0C4	-	-	-	Reserved.
0x0C5	Power-Down Status	v3.3	R/W	See *Device Power-Down Status Register (PDSR)* on page 3-85
0x0C6-0x3BF, ETM Trace Registers[c]				
0x0C6-0x37F	-	-	-	Reserved.
0x380-0x3BF	Integration registers	v3.2	-	Reserved for IMPLEMENTATION DEFINED topology detection and integration registers.
0x3C0-0x3FF, ETM Management Registers[c]				
0x3C0	Integration Mode Control	v3.2	R/W	See *Integration Mode Control Register (ITCTRL), ETMv3.2 and later* on page 3-87
0x3E8	Claim Tag Set	v3.2	R/W	See *Claim Tag Set Register (CLAIMSET)* on page 3-88
0x3E9	Claim Tag Clear	v3.2	R/W	See *Claim Tag Clear Register (CLAIMCLR)* on page 3-89
0x3EC	Lock Access	v3.2	WO	See *Lock Access Register (LAR or LOCKACCESS)* on page 3-90

Table 3-3 ETM registers summary (continued)

Register[a]	Function	Version[b]	Type	Description
0x3ED	Lock Status	v3.2	RO	See *Lock Status Register (LSR or LOCKSTATUS)* on page 3-91
0x3EE	Authentication Status	v3.2	RO	See *Authentication Status Register (AUTHSTATUS), ETMv3.2 and later* on page 3-92
0x3F2	Device Configuration	v3.2	RO	See *Device Configuration Register (DEVID), ETMv3.2 and later* on page 3-95
0x3F3	Device Type	v3.2	RO	See *Device Type Register (DEVTYPE), ETMv3.2 and later* on page 3-96
Peripheral and Component ID registers:				
0x3F4	Peripheral ID4	v3.2	RO	See *Peripheral ID4 Register* on page 3-102
0x3F5	Peripheral ID5	v3.2	RO	Reserved in current implementations.See *Peripheral ID5 to Peripheral ID7 Registers* on page 3-103
0x3F6	Peripheral ID6	v3.2	RO	
0x3F7	Peripheral ID7	v3.2	RO	
0x3F8	Peripheral ID0	v3.2	RO	See *Peripheral ID0 Register* on page 3-98
0x3F9	Peripheral ID1	v3.2	RO	See *Peripheral ID1 Register* on page 3-99
0x3FA	Peripheral ID2	v3.2	RO	See *Peripheral ID2 Register* on page 3-100
0x3FB	Peripheral ID3	v3.2	RO	See *Peripheral ID3 Register* on page 3-101
0x3FC	Component ID0	v3.2	RO	See *Component ID0 Register* on page 3-104
0x3FD	Component ID1	v3.2	RO	See *Component ID1 Register* on page 3-105
0x3FE	Component ID2	v3.2	RO	See *Component ID2 Register* on page 3-106
0x3FF	Component ID3	v3.2	RO	See *Component ID3 Register* on page 3-106

a. The Register column gives the *register number*. Registers are numbered sequentially from zero. Where registers are accessed in a memory-mapped scheme, the offset of a register is (4 x register number).
b. The first ETM architecture to define the register, or (if the register type is different in different architecture versions) the first architecture version to which the description applies.
c. The split into Trace and Management registers applies from ETMv3.3, see *ETM Trace and ETM Management registers, from ETMv3.3* on page 3-16.
d. In ETMv3.1 and later, register is read/write if bit [11] of the ETM Configuration Code Extension Register (0x7A) is set to 1.

e. From ETMv3.4, it is IMPLEMENTATION DEFINED whether the Synchronization Frequency register is implemented as a write-only register that is read/write when bit [11] of the ETM Configuration Code Extension Register (0x7A) is set to 1, or as a read-only register. See the register description for more information.

3.4.1 ETM Trace and ETM Management registers, from ETMv3.3

From ETMv3.3, the ETM register map is split into two areas, as shown in Table 3-4.

Table 3-4 Split of ETM register map into Trace and Management registers

Area	Register numbers	Register addresses
ETM Trace Registers	0x000-0x0BF, 0x0C6-0x3BF	0x000-0x2FF, 0x318-0xEFF
ETM Management Registers	0x0C0-0x0C5, 0x3C0-0x3FF	0x300-0x314, 0xF00-0xFFF

Note

- Table 3-4 is based on the ETM registers implemented in ETMv3.3 and ETMv3.4. However, any ETM register numbers not specified in Table 3-3 on page 3-11 are reserved and might be used in the future as either Trace or Management registers.
- In previous issues of the ETM Architecture Specification the ETM Trace Registers have been called the ETM Debug Registers. This name change does not indicate any change in how the registers are used.

This split of the register map is made for register save/restore purposes. For more information see:

- *Operating System Save and Restore Registers, ETMv3.3 and later* on page 3-81
- *Power-down support, ETMv3.3 and later* on page 3-119.

3.4.2 Reset behavior

This document describes the following resets, or reset operations:

Processor reset

This resets the processor, making it start execution from the reset vector address. This does not reset any ETM registers. The ETM indicates the processor reset by inserting an exception packet in the trace stream. The branch address in the exception packet indicates that the exception was a processor reset.

ETM reset This resets all resettable ETM registers, as defined by the register descriptions. This is the main reset for the entire ETM.

TAP reset In an ETM that supports JTAG connections, this resets the TAP controller:

- in ETMv3.0 and earlier, this might also perform an ETM reset
- from ETMv3.1, this might not reset any ETM registers.

Power-on reset

Whether an ETM supports a power-on reset is IMPLEMENTATION DEFINED. If it is supported, a power-on reset must perform an ETM reset. If the ETM supports JTAG connections it must also perform a TAP reset.

Writing to the Programming bit

Writing to the Programming bit of the ETM control register is a reset operation that resets parts of the ETM to their ETM reset state. You reset some parts of the ETM by writing a 1 to this bit, and reset other part by writing 0 to this bit. For more information see *ETM Programming bit and associated state* on page 3-19.

On an ETM reset, the state of the ETM Control Register is set to the state described in Table 3-5 on page 3-21. In particular, the power-down bit and the Programming bit are set to 1, see *ETM Programming bit and associated state* on page 3-19.

Moving the TAP state machine into Test-Logic Reset state resets only the TAP controller. No ETM registers are affected. To prevent UNPREDICTABLE ETM behavior, a TAP reset must be asserted when the ETM is initially powered on.

In ETMv3.1 and later, a TAP reset might not reset the ETM registers because this might be done by a power-on reset, depending on the core reset methodology. See the appropriate *Technical Reference Manual* for more information. You can achieve the same effect by writing the reset value to the ETM Control Register, `0x00`.

On an ETM reset, the status of registers or individual bits is UNKNOWN where not specified.

Note

See *ETM Programming bit and associated state* on page 3-19 for a description of how the value of the Programming bit affects these registers.

3.4.3 Use of the Programming bit

When programming the ETM registers you must enable all the changes at the same time. For example, if the counter is reprogrammed, it might start to count based on incorrect events, before the trigger condition has been correctly set up. In addition, the programming interface clock can be asynchronous to the ETM clock.

You can use the ETM Programming bit, Progbit, in the ETM Control Register, to disable all operations during programming. For more information see *ETM Control Register* on page 3-20.

Figure 3-3 on page 3-18 shows the procedure for using Progbit to control the programming of the ETM registers. When the Programming bit is set to 0 you must not write to registers other than the control register, because this can lead to UNPREDICTABLE behavior.

When setting the Programming bit, you must not change any other bits of the ETM Control Register. You must only change the value of bits other than the Programming bit of the Control Register when bit [1] of the Status Register is set to 1. ARM recommends that you use a read-modify-write procedure when modifying the ETM Control Register.

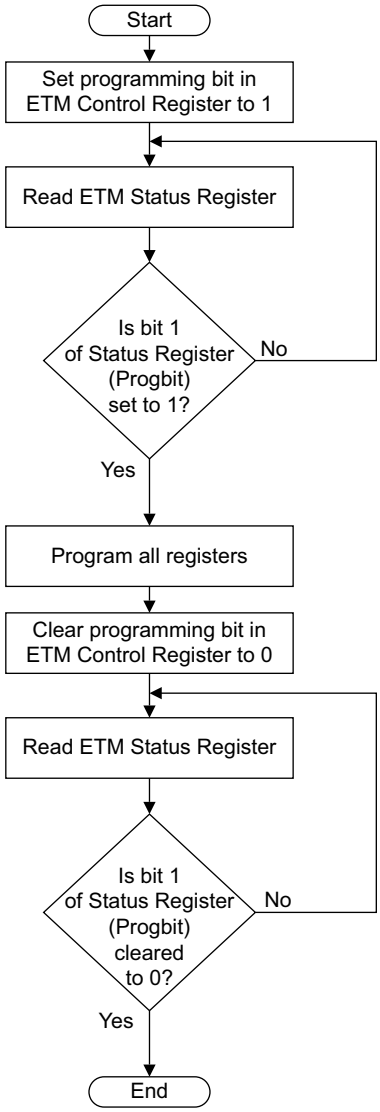


Figure 3-3 Programming ETM registers

The processor does not have to be in Debug state when programming the registers.

3.4.4 ETM Programming bit and associated state

The ETM Programming bit disables all ETM functions. See Figure 3-3 on page 3-18 for the procedure to change this bit. While it is asserted:

- The trace port is disabled. The FIFO is emptied and then no more trace is produced.
- The counters, sequencer, and start/stop block are held in their current state.
- The external outputs are forced LOW.

ETM state items

The ETM has five items of state that are affected by the Programming bit:

- the value of the counters (Counter Value Registers, `0x5C-0x5F`)
- the sequencer
- the start/stop resource status, bit [2] of the ETM Status Register
- the start/stop block
- the trigger flag, bit [3] of the ETM Status Register.

ETMv3.0 and earlier

In ETMv3.0 and earlier none of this state can be directly written. It is reset when the Programming bit is set. The values are then held and tracing is disabled until the Programming bit is cleared. For example, a free-running counter does not decrement while the Programming bit is set, and triggers cannot occur.

The state is reset as follows:

- the counters are reloaded with the value of the appropriate Counter Reload Value Register
- the sequencer is reset to state 1
- the start/stop block is reset to off
- the triggered bit is cleared to 0, meaning triggers can occur.

The counter value is also updated when the Counter Reload Value Register is written, if the Programming bit is set.

The start/stop block state can only be read in ETMv2.0 and later. The triggered bit cannot be read until ETMv3.1 See *ETMv3.1 and later*.

Care must be taken to read the state before setting the Programming bit, otherwise the state is lost.

ETMv3.1 and later

In ETMv3.1 and later the state information can be directly read and written. This means the state information can be saved when powering down the ARM core, and restored when the system is restarted. See *ETM state items* for a list of the state information that can be saved and restored in this way.

The state is held while the Programming bit is set and tracing is disabled as before. The state is reset when the Programming bit is cleared, unless written to since the Programming bit was last set. You must set the Programming bit before reading the state. This holds the state stable, ensuring you obtain a consistent result. See *Use of the Programming bit* on page 3-17 for more information.

3.5 Detailed register descriptions

The ETM registers are listed in Table 3-3 on page 3-11. This section describes each of the registers. Registers are read/write, unless the register description says otherwise.

Note

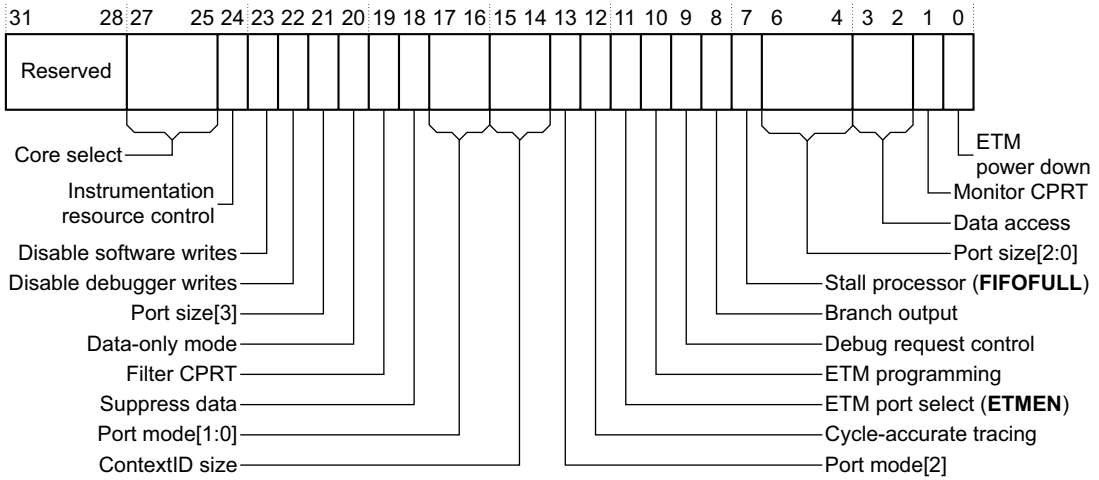
- In the register bit description tables, the *Version* column indicates:
  - differences in how the bit is used in different architecture versions
  - the first version of the architecture that uses the bit.
- Register bits are reserved in architecture versions earlier than the first use shown in the *Version* column. You must treat these reserved bits as read UNKNOWN and Write-As-Zero, unless the register description indicates otherwise.

3.5.1 ETM Control Register

The ETM Control Register controls general operation of the ETM, such as whether tracing is enabled or coprocessor data is traced. It is:

- register `0x00`, at offset `0x00` in a memory-mapped implementation
- a read/write register.

Figure 3-4 shows the bit assignments for the ETM Control Register, for ETM architecture version 3.3:



Shown for ETM architecture v3.3. See the register bit assignments table for differences in other architecture versions.

Figure 3-4 ETM Control Register bit assignments for architecture v3.3

Table 3-5 shows the bit assignments for the ETM Control Register, and describes how these differ in different versions of the ETM architecture:

Table 3-5 ETM Control Register bit assignments

Bit	Function	Version[a]	Description
[31:28]	Reserved	-	Must be written as 0.
[27:25]	Core select	v3.2	If an ETM is shared between multiple cores, selects which core to trace. For the maximum value permitted, see bits [14:12] of the *System Configuration Register bit assignments* on page 3-35. To guarantee that the ETM is correctly synchronized to the new core, you must update these bits as follows: 1. Set bit [10], ETM programming, and bit [0], ETM power down, to 1. 2. Change the core select bits. 3. Clear bit [0], ETM power down, to 0. 4. Perform other programming required as normal. The ETM cannot be shared if JTAG access is supported. On an ETM reset these bits are all zero.
[24]	Instrumentation resources access control	v3.3	When this bit is set to 1, the Instrumentation resources can only be controlled when the processor is in a privileged mode. When this bit is set to 0, the Instrumentation resources can be accessed in both privileged and User modes. On an ETM reset this bit is 0. If no Instrumentation resources are implemented this bit reads as zero and ignores writes. This bit is only writable if at least one instrumentation resource is implemented. Otherwise, it reads as zero and ignores writes.
[23]	Disable software writes	v3.2	Register writes from software disabled. This bit can only be written by the debugger. This bit is not supported in all implementations. This bit reads back as zero if not supported. On an ETM reset this bit is 0.

Table 3-5 ETM Control Register bit assignments (continued)

Bit	Function	Version[a]	Description
[22]	Disable register writes from the debugger	v3.1	Register writes from the debugger disabled. This bit can only be written by software. **Note** Typically a debugger can halt the processor to simulate software accesses. This means that, even if this bit is set, the debugger might be able to access the ETM registers. This bit is not supported in all implementations. This bit reads as zero if not supported. On an ETM reset this bit is 0.
[21]	Port size[3]	v3.0	For ETMv3.0 and later use this in conjunction with bits [6:4]. On an ETM reset this bit is 0.
[20]	Data-only mode	v3.1	The possible values of this bit are: **0** Instruction trace enabled. **1** Instruction trace disabled. Data-only tracing is possible in this mode. On an ETM reset this bit is 0.
[19]	Filter (CPRT)	v3.0	In ETMv2.x and earlier, CPRT tracing ignores **ViewData** and is controlled by a single bit, bit [1] of this register. From ETMv3.0, this bit is used in conjunction with bit [1], the MonitorCPRT bit. See *Filter Coprocessor Register Transfers (CPRT) in ETMv3.0 and later* on page 2-28. On an ETM reset this bit is 0.
[18]	Suppress data	v3.0	Used with bit [7] to suppress data, see *Data suppression* on page 2-33. On an ETM reset this bit is 0. For information about the interaction of this bit with bit [7] see *Restriction if FIFOFULL and data suppression are both implemented* on page 2-35.
[17:16]	Port mode	v1.2 up to v2.1	These bits enable the trace port clocking mode to be set. See *Trace port clocking modes* on page 2-67. On a TAP reset or ETM reset these bits are cleared to 0.
	Port mode [1:0]	v3.0	These bits, in conjunction with bit [13], enable the trace port clocking mode to be set. See *Trace port clocking modes* on page 2-67.

Table 3-5 ETM Control Register bit assignments (continued)

Bit	Function	Version[a]	Description
[15:14]	ContextIDsize	v1.2	The possible values of this field are: **b00** No Context ID tracing. **b01** Context ID bits [7:0] traced. **b10** Context ID bits [15:0] traced. **b11** Context ID bits [31:0] traced. **Note** Only the number of bytes specified is traced even if the new value is larger than this. From ETMv1.2, these bits Read-as-Zero if Context ID tracing is not supported. On an ETM reset these bits are zero.
[13]	Half-rate clocking	v1.2 up to v2.1	This bit controls whether trace is captured off both edges of **TRACECLK** or only the rising edge. See *Trace port clocking modes* on page 2-67. On an ETM reset this bit is 0.
	Port mode[2]	v3.0	This bit enables the trace port clocking mode to be set in conjunction with bits [17:16]. See *Trace port clocking modes* on page 2-67. On an ETM reset this bit is 0.
[12]	Cycle-accurate tracing	v1.0	When set to 1, a precise cycle count of executed instructions can be extracted from the trace. In ETMv1 and ETMv2, this is achieved by causing trace to be captured on every cycle when **TraceEnable** is active. In ETMv3, this is achieved by adding extra information into the trace, giving cycle counts even when **TraceEnable** is inactive. On an ETM reset this bit is 0.
[11]	ETM port selection	v1.0	This bit controls the external **ETMEN** pin. The possible values are: **0** **ETMEN** is LOW. **1** **ETMEN** is HIGH. This bit must be set by the trace software tools to ensure that trace output is enabled from this ETM. See also *Restrictions on the use of the ETMEN signal* on page 3-27. **ETMEN** can be used to enable the trace port pins to be shared with GPIO pins under the control of logic external to the ETM. On an ETM reset this bit is 0.

Table 3-5 ETM Control Register bit assignments (continued)

Bit	Function	Version[a]	Description
[10]	ETM programming	v1.0	When set to 1, the ETM is being programmed, see *ETM Programming bit and associated state* on page 3-19. On an ETM reset this bit is set to b1.
[9]	Debug request control	v1.0	When set to 1 and the trigger event occurs, the **DBGRQ** output is asserted until **DBGACK** is observed. This enables the ARM processor to be forced into Debug state. On an ETM reset this bit is 0.
[8]	Branch output	v1.0	When set to 1 all branch addresses are output, even if the branch was because of a direct branch instruction. Setting this bit enables reconstruction of the program flow without having access to the memory image of the code being executed. On an ETM reset this bit is 0. This bit is not supported by all ETMs. From ETMv2.0, if unsupported, this bit ignores writes and Reads-As-Zero.
[7]	Stall processor	v1.0	The **FIFOFULL** output can be used to stall the processor to prevent overflow. This signal is only enabled when the stall processor bit is set to 1. When this bit is 0 the **FIFOFULL** output remains LOW at all times and the FIFO overflows if there are too many trace packets. On an ETM reset this bit is 0. For information about the interaction of this bit with bit [18] see *Restriction if FIFOFULL and data suppression are both implemented* on page 2-35. If the **FIFOFULL** signal is not implemented then this bit reads as zero and ignores writes.
[6:4]	Port size [2:0]	v1.0	The port size determines how many external pins are available to output the trace information. In ETMv1 and ETMv2 the port size is the number of bits in **TRACEPKT**. In ETMv3 the port size is the number of bits in **TRACEDATA**. This configuration determines how quickly the trace packets are extracted from the FIFO. From ETMv3 the port size field is 4 bits wide and bits [6:4] must be used in conjunction with bit [21], so that the port size encoding is given by bits [21, 6:4]. See *ETM port size encoding* on page 3-26 for the encoding of these bits. On an ETM reset these bits correspond to the lowest supported port width.

Table 3-5 ETM Control Register bit assignments (continued)

Bit	Function	Version[a]	Description
[3:2]	Data access	v1.0	The possible values of this field are: **b00** No data tracing. **b01** Trace only the data portion of the access. **b10** Trace only the address portion of the access. **b11** Trace both the address and the data of the access. On an ETM reset these bits are b00.
[1]	MonitorCPRT	v1.0	When 0, the CPRTs are not traced. When set to 1, the CPRTs are traced. On an ETM reset this bit is 0. From ETMv2.1, if CPRT tracing is not supported then this bit reads back as 0. From ETMv.3.0, this bit is used with bit [19]. See *Filter Coprocessor Register Transfers (CPRT) in ETMv3.0 and later* on page 2-28
[0]	ETM power down	v1.0	A pin controlled by this bit enables the ETM power to be controlled externally. The external pin is often **ETMPWRDOWN** or inverted as **ETMPWRUP**. This bit must be cleared by the trace software tools at the beginning of a debug session. When this bit is set to 1, the ETM must be powered down and disabled, and then operated in a low power mode with all clocks stopped. When this bit is set to 1, writes to some registers and fields might be ignored. You can always write to the following registers and fields: • ETM Control Register, bit [0] and bits [27:25] • Lock Access Register • Claim Tag Set Register • Claim Tag Clear Register • Operating System Lock Access Register. When the ETM Control Register is written with this bit set to 1, bits other than bit [0] and bits [27:25] might be ignored. On an ETM reset this bit is set to 1.

a. The first ETM architecture version that defines the field, or (where the use of a field is different in different versions) the first architecture version to which the description applies.

Additional information on the ETM Control Register

Additional information about fields of the ETM Control Register is given in:

- *ETM port size encoding* on page 3-26

- *Restrictions on the use of the ETMEN signal* on page 3-27.

Note

The debug tools must read back the ETM Control Register after modification, to confirm that writes were successful. In particular:

- If you select a port width that is not supported by an ETM configuration, the closest supported size is selected (ETMv1.x and ETM2.x only). In ETMv3.x and later selection of an unsupported port size results in invalid trace.
- The branch output bit is not supported by all ETM versions. If this bit is not supported, it is 0 when read back.

ETM port size encoding

Table 3-6 shows the encoding of the ETM port size in the ETM Control Register:

- from ETMv3.0 the port size is encoded in register bits [21, 6:4]
- before ETMv3.0 the port size is encoded in register bits [6:4] only, and the first bit in the Register bits column of the table must be ignored.

Table 3-6 ETM port size

Register bits [21, 6:4][a]	Port size	Available[b] in ETM versions:
b0000	4 bit	All
b0001	8 bit	All
b0010	16 bit	All
b0011	24 bit[c]	From ETMv3.0
b0100	32 bit[c]	From ETMv3.0
b0101	48 bit[c]	From ETMv3.0
b0110	64 bit[c]	From ETMv3.0
b0111	Reserved	All
b1000	1 bit	From ETMv3.0
b1001	2 bit	From ETMv3.0
b101X	Reserved	From ETMv3.0

Table 3-6 ETM port size (continued)

Register bits [21, 6:4][a]	Port size	Available[b] in ETM versions:
b110X	Reserved	From ETMv3.0
b1110	User defined 1	From ETMv3.0
b1111	User defined 2	From ETMv3.0

a. Before ETMv3.0, the port size is encoded in register bits [6:4] only.
b. An ETM implementation might not support all available encodings, see the information in this section.
c. Reserved in ETM versions earlier than ETMv3.0.

Not all port sizes are supported by all implementations. You can determine which port sizes from the Maximum port size bits of the System Configuration Register, `0x05`, see *System Configuration Register, ETMv1.2 and later* on page 3-35.

Restrictions on the use of the ETMEN signal

You must not use the **ETMEN** signal to gate the ETM clock or any other functionality required for basic operation. The **ETMEN** signal can be used to control functionality that is only required for off-chip tracing, such as multiplexing between two ETMs. Use the **ETMPWRDOWN** signal to control basic operation of the ETM.

Checking for IMPLEMENTATION DEFINED features, from ETMv3.3

From ETMv3.3, a number of ETM features become IMPLEMENTATION DEFINED, and debug tools can interrogate the ETM Control Register to check whether an ETM macrocell supports these features. Table 3-7 summarizes where these checks are described.

Table 3-7 ETM Control Register checks for IMPLEMENTATION DEFINED features

ETM feature	ETM Control Register	For more information, see:
Data tracing options	Bits [20:18, 3:1]	*Detecting which data tracing options are available* on page 7-55
Data suppression support	Bit [18]	*Checking whether data suppression is supported, ETMv3.3 and later* on page 2-34
Cycle-accurate tracing support	Bit [12]	*Checking support for cycle-accurate tracing, ETMv3.3 and later* on page 3-28

Checking support for cycle-accurate tracing, ETMv3.3 and later

From ETMv3.3, whether cycle-accurate tracing is defined is IMPLEMENTATION DEFINED, and debug tools can write and then read the ETM Control Register to find whether cycle-accurate tracing is supported. To avoid changing other ETM control settings, the test process is:

1. Read the ETM Control Register.
2. In the returned data, set bit [12], the Cycle-accurate tracing bit, to 1.
3. Write the modified value back to the ETM Control Register.
4. Read the ETM Control Register again.

Checking bit [12] of the register value returned at stage 4 of the test indicates whether data suppression is supported. The possible results are shown in Table 3-8.

Table 3-8 Testing whether cycle-accurate tracing is supported, ETMv3.3 and later

ETM Control Register bit [12]	Data suppression option
1	Cycle-accurate tracing supported
0	Cycle-accurate tracing not supported

Checking which data tracing options are available, ETMv3.3 and later

From ETMv3.3, it is IMPLEMENTATION DEFINED whether the following data trace options are available:

- data address tracing
- data value tracing
- CPRT tracing
- data-only mode.

These options are not independent, see *Data tracing options, ETMv3.3 and later* on page 7-54 for details of the permitted implementations.

Debug tools can find out which data tracing options are implemented by writing to the ETM Control Register with the appropriate bits set to one, and then reading the register back to see whether those bits have been set. To avoid changing other ETM control settings, the test process is:

1. Read the ETM Control Register.
2. Set the following bits or fields in the returned data:
   - bit [1], the Monitor CPRT bit, to 1
   - bits [3:2], the Data access field, to b11
   - bit [18], the Suppress data bit, to 1
   - bit [19], the Filter CPRT bit, to 1
   - bit [20], the Data-only mode bit, to 1.
3. Write the modified value back to the ETM Control Register.
4. Read the ETM Control Register again.

Checking bits [21:18, 3:1] of the register value returned at stage 4 of the test indicates which data tracing features are implemented. The values that can be returned are shown in Table 3-9.

Table 3-9 Testing which data tracing features are implemented, ETMv3.3 and later

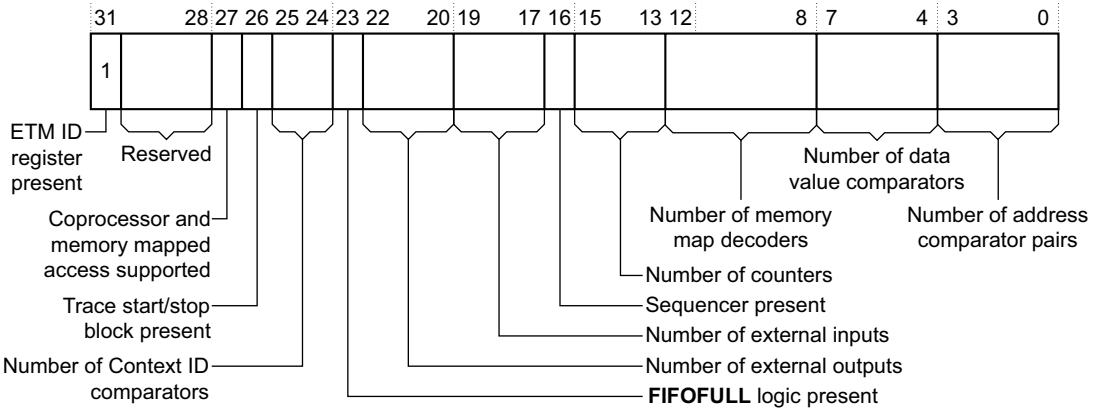
ETM Control Register		Data address tracing	Data value Tracing	CPRT tracing	Data-only mode
Bits [20:18]	Bits [3:1]				
b11X	b111	Full implementation - all data tracing features are implemented			
b00X	b100	Implemented	Not implemented	Not implemented	Not implemented
b01X	b011	Not implemented	Implemented	Implemented	Not implemented
b000	b000	Not implemented	Not implemented	Not implemented	Not implemented

3.5.2 ETM Configuration Code Register

The ETM Configuration Code Register enables the debugger to read the IMPLEMENTATION DEFINED configuration of the ETM, giving the number of each type of resource. Where a value indicates the number of instances of a particular resource, zero indicates that there are no implemented resources of that resource type. The register is:

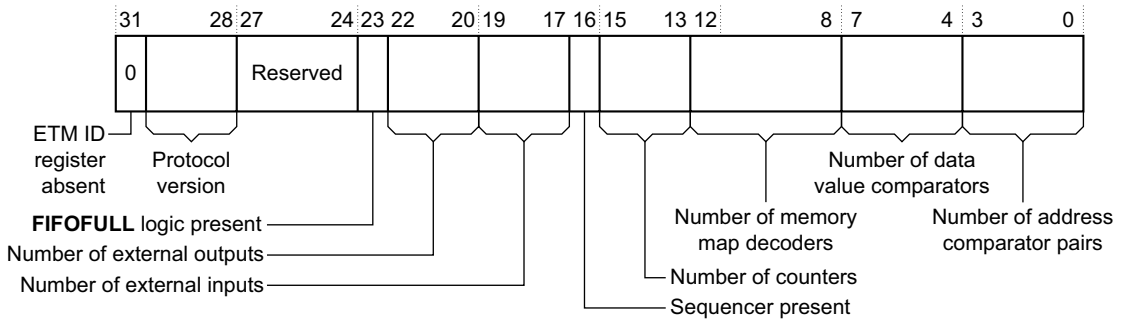
- register `0x01`, at offset `0x04` in a memory-mapped implementation
- a read-only register.

Figure 3-5 shows the bit assignments for the ETM Configuration Code Register, for architecture version 3.1 and later, and Figure 3-6 on page 3-30 shows the bit assignments for architecture versions 1.x.



Shown for ETM architecture v3.1. See the register bit assignments table for differences in other architecture versions

Figure 3-5 ETM Configuration Code Register bit assignments, from architecture v3.1



Shown for ETM architecture v1.x. See the register bit assignments table for differences in other architecture versions

Figure 3-6 ETM Configuration Code Register bit assignments for architecture v1.x

Table 3-10 shows the bit assignments for the Configuration Code Register, and describes how these are different for different versions of the ETM architecture:

Table 3-10 ETM Configuration Code Register bit assignments

Bit	Max. value	Version[a]	Description
[31]	1	All	When set to 1, this bit indicates that the ETM ID Register, `0x79`, is present and defines the ETM architecture version in use. When set to 0, this bit indicates that the ETM ID Register is not present. For details, see *ETM ID Register, ETMv2.0 and later* on page 3-71.
[30:28]	-	v2.0	Reserved. For ETMv2.0 and later the ETM architecture version is given in the ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71.
	7	v1.x only	Protocol version (when ETM ID Register not present).
[27]	1	v3.1	Coprocessor or memory-mapped access to registers supported. See *Programming and reading ETM registers* on page 3-3.
[26]	1	v2.0	When set to 1, the trace start/stop block is present. In ETMv1.2 and ETMv1.3, the trace start/stop block is always present and this bit is Read-As-Zero.
[25:24]	3	v2.0	Number of Context ID comparators.

Table 3-10 ETM Configuration Code Register bit assignments (continued)

Bit	Max. value	Version[a]	Description
[23]	1	v1.0	When set to 1, the **FIFOFULL** logic is present. This bit is used in conjunction with bit [8] of the System Configuration Register, register `0x05`, of the processor core connected to the ETM. **Note** You can use **FIFOFULL** only if it is supported by both your ETM and your system. Some cores do not support **FIFOFULL**, so it cannot be used by the system. If this bit is 0, the FIFOFULL Region Register, `0x0A`, is not implemented and is Reserved, RAZ. In this case, the Processor stall bit, bit [7], of the ETM Control Register might ignore writes.
[22:20]	4	v1.0	Number of external outputs. Supplied by the ASIC in ETMv3.1 and later.
[19:17]	4	v1.0	Number of external inputs. Supplied by the ASIC in ETMv3.1 and later.
[16]	1	v1.0	When set to b1the sequencer is present.
[15:13]	4	v1.0	Number of counters.
[12:8]	16	v1.0	Number of memory map decoder inputs. If this bit is 0, the ViewData Control 2 Register, `0x0E`, is not implemented and is Reserved, RAZ.
[7:4]	8	v1.0	Number of data value comparators. From ETMv3.3, this field is zero if data address comparisons are not supported. See *No data address comparator option, ETMv3.3 and later* on page 2-7 for more information.
[3:0]	8	v1.0	Number of pairs of address comparators.

a. The first ETM architecture version that defines the field, or (where the use of a field is different in different versions) the first architecture version to which the description applies.

3.5.3 Trigger Event Register

The Trigger Event Register defines the event that controls the trigger. It is:

- register `0x02`, at offset `0x08` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-7 on page 3-32 shows the bit assignments for the Trigger Event Register:

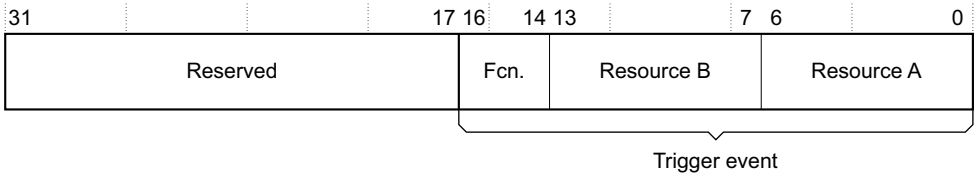


Figure 3-7 Trigger Event Register bit assignments

Table 3-11 shows the bit assignments for the Trigger Event Register:

Table 3-11 Trigger Event Register bit assignments

Bit	Defined in ETM architecture versions	Description
[31:17]	-	Reserved
[16:0]	v1.0 and later	Trigger event

The section *Using ETM event resources* on page 3-108 describes how you define a trigger event.

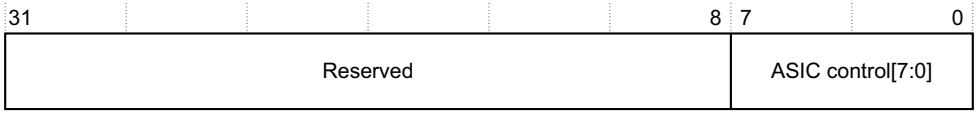
3.5.4 ASIC Control Register

The ASIC Control Register is used to control ASIC logic, such as the static configuration of MMDs. It was previously called the *Memory Map Decoder* (MMD) Register. It is:

- register `0x03`, at offset `0x0C` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.
- an optional register in implementations where no MMDs are supported.

Support of this register is IMPLEMENTATION DEFINED, and the size of the register is also IMPLEMENTATION DEFINED but is usually eight bits. Tools cannot detect whether this register is implemented, or its size. Writing to this register has no effect if it is not implemented.

Figure 3-8 shows the bit assignments for the ASIC Control Register:



The size of the ASIC control field is implementation defined. Register is shown for an 8-bit implementation.

Figure 3-8 ASIC Control Register bit assignments

Table 3-12 shows the bit assignments for the ASIC Control Register:

Table 3-12 ASIC Control Register bit assignments

Bits	Defined in ETM architecture versions	Description
[31:n+1]	-	Reserved
[n:0]	v1.0 and later	ASIC control

The value of this register is output by the ETM to the ASIC logic over a dedicated bus. It can be used for many purposes, but its intended use is to refine memory map decoders. See *Memory map decoder (MMD)* on page 2-8.

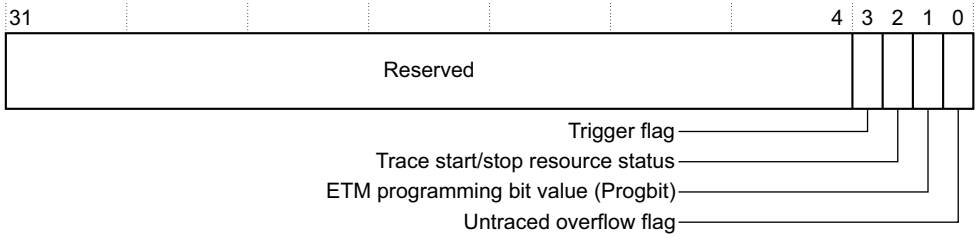
Even where MMDs are not supported by the ETM, it is sometimes desirable for the tools to be able to communicate with the ASIC in a general manner, in addition to the special-purpose bits defined in the ETM Control Register. You can use the ASIC control register for this communication.

3.5.5 ETM Status Register, ETMv1.1 and later

The ETM Status Register provides information about the current status of the trace and trigger logic. It is:

- register `0x04`, at offset `0x10` in a memory-mapped implementation
- a read-only register in ETM architecture versions 3.0 and earlier
- a read/write register in ETM architecture versions 3.1 and later.

Figure 3-9 shows the bit assignments for the ETM Status Register, for ETM architecture version 3.1 and later:



Shown for ETM architecture v3.1. See the register bit assignments table for differences in other architecture versions.

Figure 3-9 ETM Status Register bit assignments for architecture v3.1

Table 3-13 on page 3-34 shows the bit assignments for the ETM Status Register, and describes the differences between different ETM architecture versions

Table 3-13 ETM Status Register bit assignments

Bit	Type[a]	Version[b]	Description
[31:4]	-	-	Reserved.
[3]	R/W	v3.1	Trigger bit. Set when the trigger occurs, and prevents the trigger from being output until the ETM is next programmed. This bit exists in all architecture versions, but can only be accessed in ETMv3.1 and later as described in *ETM Programming bit and associated state* on page 3-19.
[2]	RO	v1.2 to v3.0	Holds the current status of the trace start/stop resource. If set to 1, it indicates that a *trace on* address has been matched, without a corresponding *trace off* address match.
	R/W	v3.1	
[1]	RO	v1.2	The current effective value of the ETM Programming bit (ETM Control Register bit [10]). You must wait for this bit to go to 1 before you start to program the ETM as described in *ETM Programming bit and associated state* on page 3-19. If you read other bits in the ETM Status Register while this bit is 0, some instructions might not have taken effect. It is recommended that you set the ETM Programming bit and wait for this bit to go to 1 before reading the overflow bit. In ETMv3.2 and later this bit remains 0 if there is any data in the FIFO. This ensures that the FIFO is empty before the ETM programming is changed.
[0]	RO	v1.1	If set to 1, there is an overflow that has not yet been traced. This bit is cleared to 0 when either: • trace is restarted. • the ETM Power Down bit, bit [0] of the ETM Control Register, `0x00`, is set to 1. **Note** Setting or clearing the ETM Programming bit does not cause this bit to be cleared to 0.

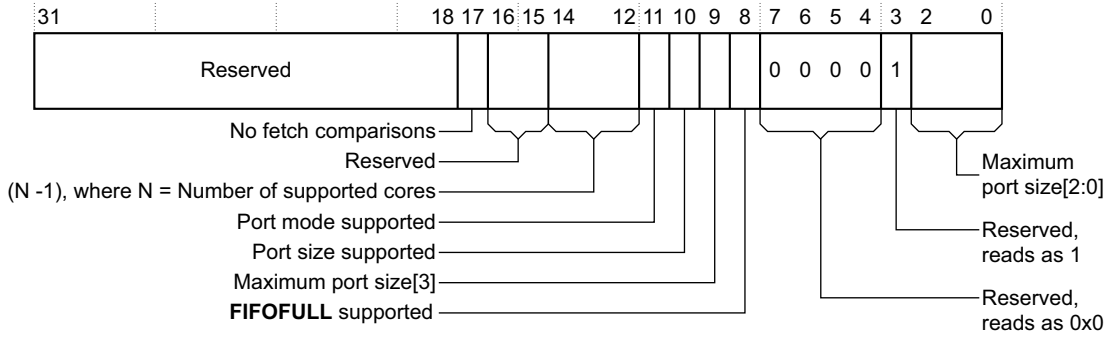
a. In architecture versions before 3.1, all fields in the register are RO.
b. The first ETM architecture version that defines the field, or (where the use of a field is different in different versions) the first architecture version to which the description applies.

3.5.6 System Configuration Register, ETMv1.2 and later

The System Configuration Register shows the ETM features supported by the ASIC. The contents of this register are based on inputs provided by the ASIC. It is:

- register `0x05`, at offset `0x14` in a memory-mapped implementation
- only available in ETMv1.2 or later
- a read-only register.

Figure 3-10 shows the bit assignments for the System Configuration Register, for ETM architecture version 3.2:



Shown for ETM architecture v3.2 See the register bit assignments table for differences in other architecture versions

Figure 3-10 System Configuration Register bit assignments for architecture v3.2

Table 3-14 shows the bit assignments for the System Configuration Register, and describes the differences between different ETM architecture versions.

Table 3-14 System Configuration Register bit assignments

Bit	Version[a]	Description
[31:18]	-	Reserved.
[17]	v2.1	No Fetch comparisons. Address comparators are not capable of performing fetch-stage comparisons. Setting bits [2:0] of an Address Access Type Register to b000 (instruction fetch) causes the comparator to have UNPREDICTABLE behavior.
[16:15]	-	Reserved.
[14:12]	v3.2	Number of supported cores minus 1. The value given here is the maximum value that can be written to bits [27:25] of the ETM Control Register, `0x00`. These bits must be b000 if JTAG access is supported.
[11]	v3.0	Port mode supported. Set to 1 if the currently selected port mode is supported internally or externally.

Table 3-14 System Configuration Register bit assignments (continued)

Bit	Version[a]	Description
[10]	v3.0	Port size supported. Set to 1 if the currently selected port size is supported internally or externally for the currently selected port mode. Enables more complex port sizes to be supported.
[9]	v3.0	Maximum port size[3]. This bit is used in conjunction with bits [2:0].
[8]	v1.3	If set to 1, **FIFOFULL** is supported. This bit is used in conjunction with bit [23] of the ETM Configuration Register, `0x01`. **Note** You can use **FIFOFULL** only if it is supported by both your ETM and your system. Some cores do not support **FIFOFULL**, so it cannot be used by the system.
[7]	v2.x and earlier	If set to 1, demultiplexed trace data format is supported.
	v3.0 and later	Reserved, reads as zero. Use bit [11] instead.
[6]	v2.x and earlier	If set to 1, multiplexed trace data format is supported.
	v3.0 and later	Reserved, reads as zero. Use bit [11] instead.
[5]	v2.x and earlier	If set to 1, normal trace data format is supported.
	v3.0 and later	Reserved, reads as zero. Use bit [11] instead.
[4]	v2.x and earlier	If set to 1, full-rate clocking is supported.
	v3.0 and later	Reserved, reads as zero. Full-rate clocking is no longer supported.

Table 3-14 System Configuration Register bit assignments (continued)

Bit	Version[a]	Description
[3]	v2.x and earlier	If set to 1, half-rate clocking is supported.
	v3.0 and later	Reserved, Read-As-One. All modes use half-rate clocking.
[2:0]	v1.2 and later	Maximum port size[2:0]. This bit is used in conjunction with bit [9]. The value given here is the maximum size supported by both the ETM and the ASIC. Smaller sizes might or might not be supported. Check bit [10] for precise details of supported modes. See bits [6:4] in *ETM Control Register bit assignments* on page 3-21.

a. The first ETM architecture version that defines the field, or (where the use of a field is different in different versions) the first architecture version to which the description applies.

3.5.7 TraceEnable registers

The **TraceEnable** trace filtering signal is described in *TraceEnable and filtering the instruction trace* on page 2-20. Four registers are used to configure **TraceEnable**, and these are described in the following sections:

- *Trace Start/Stop Resource Control Register, ETMv1.2 and later*
- *TraceEnable Event Register* on page 3-41
- *TraceEnable Control 1 Register* on page 3-39.
- *TraceEnable Control 2 Register, ETMv1.2 and higher* on page 3-38.

These registers are write-only, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

For an example of **TraceEnable** control register encoding, see *An example TraceEnable configuration* on page 3-115.

Trace Start/Stop Resource Control Register, ETMv1.2 and later

The Trace Start/Stop Resource Control Register specifies the single address comparators that hold the trace start and stop addresses. It is:

- register `0x06`, at offset `0x18` in a memory-mapped implementation
- only available in ETMv1.2 or later
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-11 on page 3-38 shows bit assignments for the Trace Start/Stop Resource Control Register:

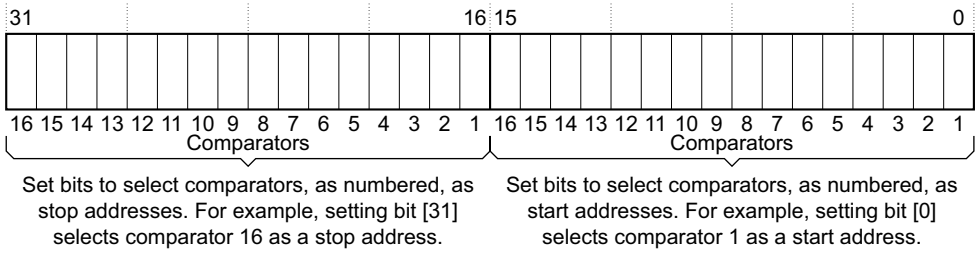


Figure 3-11 Trace Start/Stop Resource Control Register bit assignments

Table 3-15 shows the bit assignments for the Trace Start/Stop Resource Control Register:

Table 3-15 Trace Start/Stop Resource Control Register bit assignments

Bit	Version[a]	Description
[31:16]	v1.2	When a bit is set to 1, it selects a single address comparator 16-1 as stop addresses. For example, bit [16] set to 1 selects single address comparator 1 as a stop address.
[15:0]	v1.2	When a bit is set to 1, it selects a single address comparator 16-1 as start addresses. For example, bit [0] set to 1 selects single address comparator 1 as a start address.

a. The first ETM architecture version that defines the field.

TraceEnable Control Registers

The **TraceEnable** control registers hold the include/exclude resource controls. In ETMv1.2 and higher there are two **TraceEnable** control registers. Earlier versions of the ETM have only one **TraceEnable** control register.

TraceEnable Control 2 Register, ETMv1.2 and higher

The TraceEnable Control 2 Register specifies the single address comparators that hold the addresses used for include/exclude control. It is:

- register `0x07`, at offset `0x1C` in a memory-mapped implementation
- only available in ETMv1.2 or later
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-12 on page 3-39 shows the bit assignments for the TraceEnable Control 2 Register:

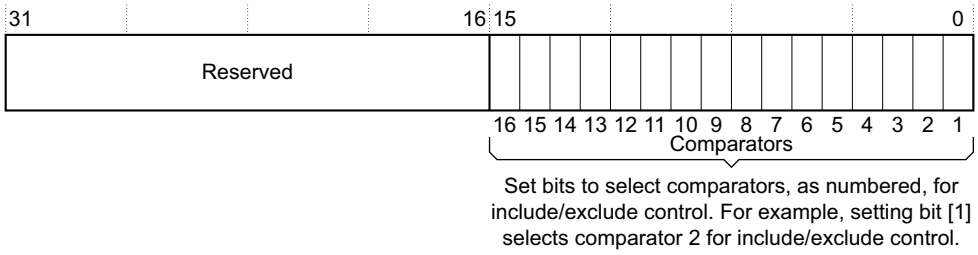


Figure 3-12 TraceEnable Control 2 Register bit assignments

Table 3-16 shows the bit assignments for the TraceEnable Control 2 Register:

Table 3-16 TraceEnable Control 2 Register bit assignments

Bit	Version[a]	Description
[31:16]	-	Reserved.
[15:0]	v1.2	When a bit is set to 1, it selects a single address comparator 16-1 for include/exclude control. For example, bit [0] set to 1 selects single address comparator 1.

a. The first ETM architecture version that defines the field.

Whether the specified comparators hold include or exclude addresses depends on the setting of the exclude/include flag in the TraceEnable Control 1 register.

TraceEnable Control 1 Register

The TraceEnable Control 1 Register:

- enables the start/stop logic
- determines whether the resources specified in the TraceEnable Control 1 and TraceEnable Control 2 Registers are used for include or exclude control
- specifies the address range comparators used for include/exclude control
- specifies the memory map decodes (MMDs) used for include/exclude control.

It is:

- register `0x09`, at offset `0x24` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-13 on page 3-40 shows the bit assignments for the TraceEnable Control 1 Register:

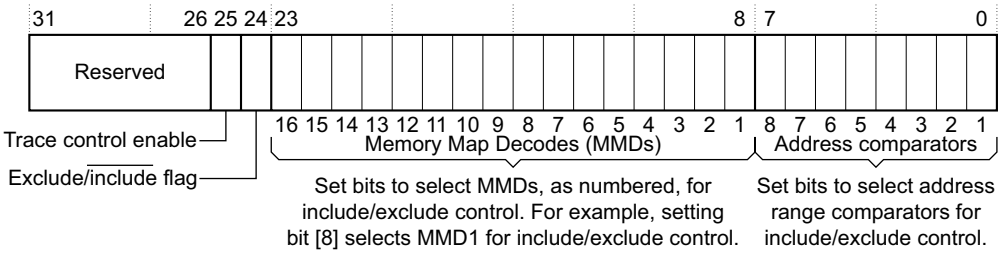


Figure 3-13 TraceEnable Control 1 Register bit assignments

Table 3-17 shows the bit assignments for the TraceEnable Control 1 Register:

Table 3-17 TraceEnable Control 1 Register bit assignments

Bit	Version[a]	Description
[31:26]	-	Reserved.
[25]	v1.2	Trace start/stop enable. The possible values of this bit are: **0** Tracing is unaffected by the trace start/stop logic. **1** Tracing is controlled by the trace on and off addresses configured for the trace start/stop logic, see *The trace start/stop block* on page 2-23. The trace start/stop resource (resource `0x5F`) is unaffected by the value of this bit.
[24]	v1.0	Include/exclude control. The possible values of this bit are: **0** Include. The specified resources indicate the regions where tracing can occur. When outside this region tracing is prevented. **1** Exclude. The resources, specified in bits [23:0] and in the TraceEnable Control 2 Register, indicate regions to be excluded from the trace. When outside an exclude region, tracing can occur.
[23:8]	v1.0	When a bit is set to 1, it selects memory map decode 16-1 for include/exclude control. For example, bit [8] set to 1 selects MMD 1.
[7:0]	v1.0	When a bit is set to 1, it selects address range comparator 8-1 for include/exclude control. For example, bit [0] set to 1 selects address range comparator 1.

a. The first ETM architecture version that defines the field.

TraceEnable Event Register

The TraceEnable Event Register defines the **TraceEnable** enabling event. It is:

- register `0x08`, at offset `0x20` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-14 shows the bit assignments for the TraceEnable Event Register:

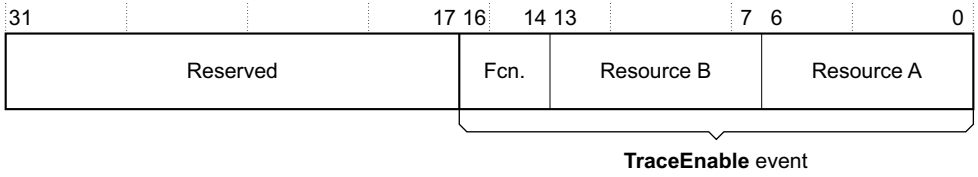


Figure 3-14 TraceEnable Event Register bit assignments

Table 3-18 shows the bit assignments for the TraceEnable Event Register:

Table 3-18 TraceEnable Event Register bit assignments

Bit	Defined in ETM architecture versions	Description
[31:17]	-	Reserved
[16:0]	v1.0 and later	**TraceEnable** event

Using ETM event resources on page 3-108 describes how you define a **TraceEnable** event.

Tracing all memory

To trace all memory:

- set bit [24] in register `0x09`, the TraceEnable Control 1 register, to 1
- set all other bits in register `0x09`, the TraceEnable Control 1 register, to 0
- set all bits in register `0x07`, the TraceEnable Control 2 register, to 0
- set register `0x08`, the TraceEnable Event register, to `0x6F` (TRUE).

This has the effect of excluding nothing, that is, tracing everything.

3.5.8 FIFO overflow registers (FIFOFULL control)

There are two FIFO overflow registers, described in the following sections:

- *FIFOFULL Region Register* on page 3-42
- *FIFOFULL Level Register* on page 3-43.

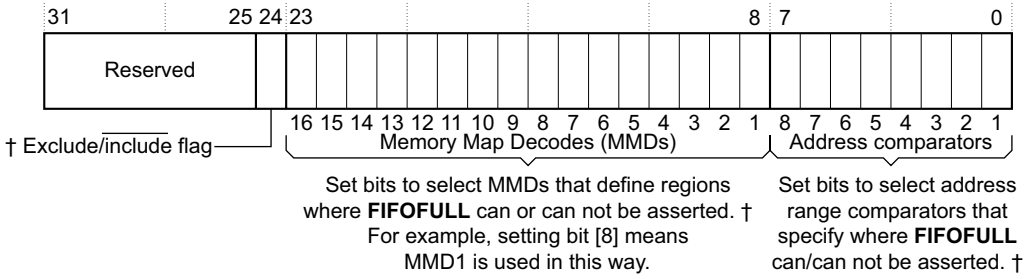
FIFOFULL Region Register

The FIFOFULL Region Register defines the regions where **FIFOFULL** can be asserted, specifying the MMDs and address comparators used for **FIFOFULL** region control. It is:

- register `0x0A`, at offset `0x28` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

The FIFOFULL Region Register does not affect data suppression. See *Data suppressed packet* on page 7-51 for more information about data suppression.

Figure 3-15 shows the bit assignments for the FIFOFULL Region Register:



† The value of bit [24] determines whether **FIFOFULL** can or can not be asserted in the specified regions.

Figure 3-15 FIFOFULL Region Register bit assignments

Table 3-19 shows the bit assignments for the FIFOFULL Region Register:

Table 3-19 FIFOFULL Region Register bit assignments

Bit	Version[a]	Description
[31:25]	-	Reserved.
[24]	v1.0	Include/exclude control. The possible values of this bit are: **0** Include. The resources specified in bits [23:0] indicate the regions where **FIFOFULL** can be asserted. When outside these regions, **FIFOFULL** cannot be asserted. **1** Exclude. The resources specified in bits [23:0] indicate the regions where **FIFOFULL** cannot be asserted. When outside these regions **FIFOFULL** can be asserted.
[23:8]	v1.0	When a bit is set to 1, it selects memory map decode 16-1 as defining regions where **FIFOFULL** can or cannot be asserted, depending on the setting of bit [24]. For example, bit [8] set to 1 selects MMD 1.

Table 3-19 FIFOFULL Region Register bit assignments (continued)

Bit	Version[a]	Description
[7:0]	v1.0	When a bit is set to 1, it selects address range comparator 8-1 for specifying regions where **FIFOFULL** can or cannot be asserted, depending on the setting of bit [24]. For example, bit [0] set to 1 selects address range comparator 1.

a. The first ETM architecture version that defines the field.

Note

To enable **FIFOFULL** anywhere in memory, set bit [24] in register `0x0A` to 1 and set all other bits to 0.

FIFOFULL Level Register

The FIFOFULL Level Register holds the level below which the FIFO is considered full, although its function varies for different ETM architectures. From ETMv3.0 the value in this register also controls the point at which data trace suppression occurs.

The FIFOFULL Level Register is:

- register `0x0B`, at offset `0x2C` in a memory-mapped implementation
- a write-only register in ETM architecture versions 1.x
- a read/write register in ETM architecture versions 2.0 and later.

Figure 3-16 shows the bit assignments for the FIFOFULL Level Register:

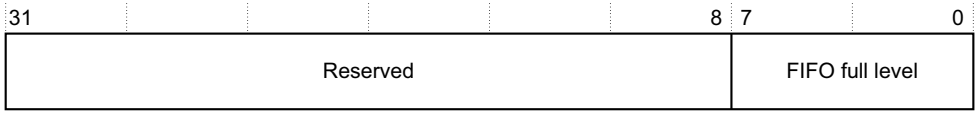


Figure 3-16 FIFOFULL Level Register bit assignments

Table 3-20 shows the bit assignments for the FIFOFULL Level Register:

Table 3-20 FIFOFULL Level Register bit assignments

Bit	Type	Version[a]	Description
[31:8]	-	-	Reserved.
[7:0]	WO	v1.0	The number of bytes left in the FIFO, below which the **FIFOFULL** or SuppressData signal is asserted. For example, setting this value to 15 causes data trace suppression or processor stalling, if enabled, when there are less than 15 free bytes in the FIFO.
	R/W	v2.0	

a. The first ETM architecture version to which the *Type* value applies.

The maximum valid value for this register is the size of the FIFO. This causes **FIFOFULL** to be asserted whenever the FIFO is not empty. Behavior is UNPREDICTABLE if the value 0 is written to this register and Stall processor or Suppress data (ETMv3 only) is selected in the ETM Control Register, `0x00`:

ETMv1.x If a value larger than the FIFO size is written to the FIFOFULL Level Register, the behavior is UNPREDICTABLE. It is not possible to determine the FIFO size from the programmer's model.

ETMv2.x Depending on the implementation, your ETM might not observe the FIFOFULL Level Register. If it does not, the ETM assumes that the **FIFOFULL** level is always set to its maximum value (see *Processor stalling, FIFOFULL* on page 2-31). This enables it to assert **FIFOFULL** earlier because no comparison with the FIFO level is required. In this case the FIFOFULL Level Register ignores writes, and returns the FIFO size when read.

If a value larger than the FIFO size is written to the FIFOFULL Level Register, the FIFO size itself is selected, and is the value returned when the register is read.

You must determine:

- the size of the FIFO
- whether the ETM ignores the FIFOFULL Level Register.

To do this you must perform the following sequence of operations:

1. Write the value `0xFFFFFFFF` to the register.
2. Read the register. The value returned is the FIFO size.
3. Write the value `0x00000001` to the register.
4. Read the register.

   If `0x00000001` is returned, the ETM observes this register.

   If the same value is returned as in step 2, the ETM ignores this register.

ETMv3.0 and later

For cores that choose to implement it, data suppression is offered in addition to or instead of **FIFOFULL**. The FIFOFULL Level Register is used for both. The modes supported are listed in Table 3-21 on page 3-45. See *Data suppressed packet* on page 7-51 for more information on data suppression.

If a value larger than the FIFO size is written to the FIFOFULL Level Register, the FIFO size itself is selected, and is the value returned when the register is read.

- bit [23] in the ETM Configuration Code Register indicates when **FIFOFULL** is supported by the ETM, see *ETM Configuration Code Register* on page 3-29
- bit [8] in the System Configuration Register indicates when **FIFOFULL** is supported by the core, see *System Configuration Register, ETMv1.2 and later* on page 3-35.

FIFOFULL can only be used when both these bits are set.

Table 3-21 Supported FIFOFULL and data suppression modes in ETMv3.0 and later

Stall processor[a]	Suppress data[b]	Description
0	0	No overflow avoidance.
1	0	**FIFOFULL** is asserted when the number of free bytes in the FIFO is less than the value in the FIFOFULL Level Register, subject to the **FIFOFULL** region if present.
0	1	Data suppression occurs if the data causes the number of free bytes in the FIFO to go below the value in the FIFOFULL Level Register. This is not subject to the **FIFOFULL** region.
1	1	UNPREDICTABLE.

a. Controlled by bit [7] of the ETM Control Register, see *ETM Control Register* on page 3-20.
b. Controlled by bit [18] of the ETM Control Register, see *ETM Control Register* on page 3-20.

3.5.9 ViewData registers

The **ViewData** trace filtering signal is described in *ViewData and filtering the data trace* on page 2-26. Four **ViewData** registers are used, and these are described in the following sections:

- *ViewData Event Register*
- *ViewData Control Registers* on page 3-46.

These registers are write-only, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

For an example of using a ViewData Control Register, see *An example ViewData configuration* on page 3-113.

Note

From ETMv3.3, it is IMPLEMENTATION DEFINED whether various data tracing options are implemented. The implementation options are described in *Data tracing options, ETMv3.3 and later* on page 7-54. If an implementation does not include any of data address tracing, data value tracing or CPRT tracing, the ViewData registers are not implemented, and the ViewData area of the register map reads as zero.

ViewData Event Register

The ViewData Event Register defines the **ViewData** enabling event. It is:

- register `0x0C`, at offset `0x30` in a memory-mapped implementation

- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-17 shows the bit assignments for the ViewData Event Register:

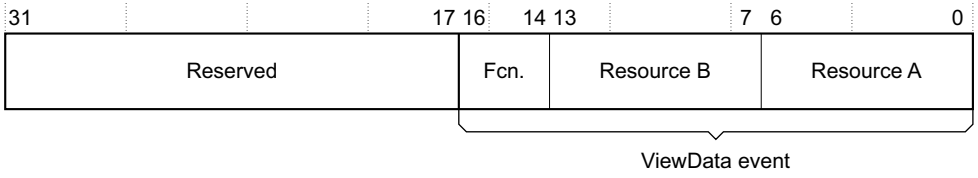


Figure 3-17 ViewData Event Register bit assignments

Table 3-22 shows the bit assignments for the ViewData Event Register:

Table 3-22 ViewData Event Register bit assignments

Bit	Defined in ETM architecture versions	Description
[31:17]	-	Reserved
[16:0]	v1.0 and later.	**ViewData** event

The section *Using ETM event resources* on page 3-108 describes how you define a **ViewData** event.

ViewData Control Registers

ViewData requires three include/exclude control registers, described in:

- *ViewData Control 1 Register*
- *ViewData Control 2 Register* on page 3-47
- *ViewData Control 3 Register* on page 3-48.

ViewData Control 1 Register

The ViewData Control 1 Register specifies the single address comparators that provide include and exclude addresses for **ViewData** operation. It is:

- register `0x0D`, at offset `0x34` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-18 on page 3-47 shows the bit assignments for the ViewData Control 1 Register:

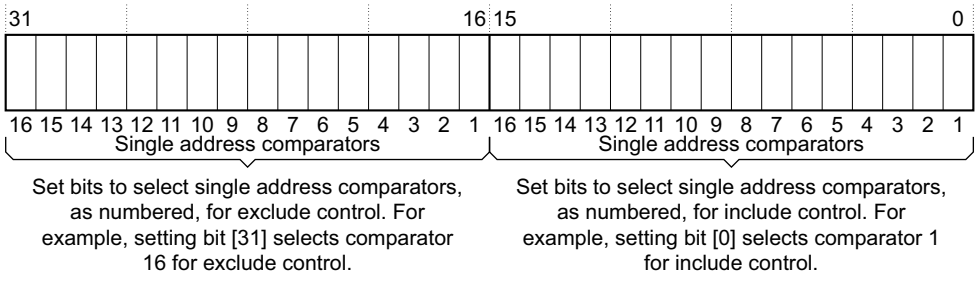


Figure 3-18 ViewData Control 1 Register bit assignments

Table 3-23 shows the bit assignments for the ViewData Control 1 Register:

Table 3-23 ViewData Control 1 Register bit assignments

Bit	Version[a]	Description
[31:16]	v1.0	When a bit is set to 1, it selects single address comparator 16 to 1 for exclude control. For example, bit [16] set to 1 selects single address comparator 1.
[15:0]	v1.0	When a bit is set to 1, it selects single address comparator 16 to 1 for include control. For example, bit [0] set to 1 selects single address comparator 1.

a. The first ETM architecture version that defines the field.

ViewData Control 2 Register

The ViewData Control 2 Register specifies the Memory Map Decodes (MMDs) that provide include and exclude control of **ViewData** operation. It is:

- register `0x0E`, at offset `0x38` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-19 on page 3-48 shows the bit assignments for the ViewData Control 2 Register:

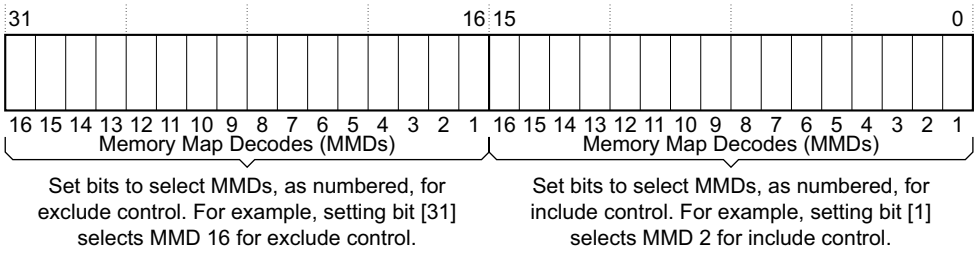


Figure 3-19 ViewData Control 2 Register bit assignments

Table 3-24 shows the bit assignments for the ViewData Control 2 Register:

Table 3-24 ViewData Control 2 Register bit assignments

Bit	Version[a]	Description
[31:16]	v1.0	When a bit is set to 1, it elects memory map decode 16 to 1 for exclude control. For example, bit [16] set to 1 selects MMD 1.
[15:0]	v1.0	When a bit is set to 1, it selects memory map decode 16 to 1 for include control. For example, bit [0] set to 1 selects MMD 1.

a. The first ETM architecture version that defines the field.

ViewData Control 3 Register

The ViewData Control 3 Register specifies the address range comparators that hold include and exclude address ranges for **ViewData** operation, and selects exclude-only operation if required. It is:

- register `0x0F`, at offset `0x3C` in a memory-mapped implementation
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-20 on page 3-49 shows the bit assignments for the ViewData Control 3 Register:

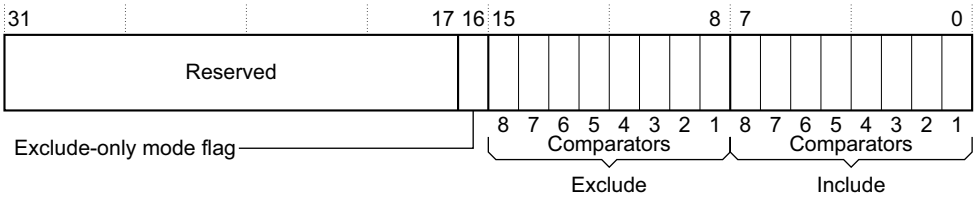


Figure 3-20 ViewData Control 3 Register bit assignments

Table 3-25 shows the bit assignments for the ViewData Control 3 Register:

Table 3-25 ViewData Control 3 Register bit assignments

Bit	Version[a]	Description
[31:17]	-	Reserved.
[16]	v1.0	Exclude-only control. The possible values of this bit are: **0** Mixed mode. **ViewData** operates in a mixed mode, and both include and exclude resources can be programmed. **1** Exclude-only mode. **ViewData** is programmed only in an excluding mode. If none of the excluding resources match, tracing can occur.
[15:8]	v1.0	When a bit is set to 1, it selects address range comparator 8-1 for exclude control. For example, bit [8] set to 1 selects address range comparator 1.
[7:0]	v1.0	When a bit is set to 1, it selects address range comparator 8-1 for include control. For example, bit [0] set to 1 selects address range comparator 1.

a. The first ETM architecture version that defines the field.

Programming the ViewData logic

An example of programming the **ViewData** logic is given in *An example ViewData configuration* on page 3-113.

Note

To enable **ViewData** throughout memory:

- set bit [16] of register `0x0F` to 1, exclude only
- set all other bits of registers `0x0D`, `0x0E`, and `0x0F` to 0
- set bits [6:0] of register `0x0C` to `0x6F`, permanently enabled, see *Defining events* on page 3-110.

3.5.10 Address comparator registers

Two registers are defined for each of the single address comparators. These registers are described in the following sections:

- *Address Comparator Value Registers*
- *Address Access Type Registers* on page 3-51.

These registers are write-only, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

You can associate each pair of address comparator registers with a data value comparator (see *Data value comparator registers* on page 3-54). If you do this, a match is triggered only when both the address and the data value match.

Note

- If data address comparators are used in **TraceEnable** exclude regions, the ETM **TraceEnable** behavior is UNPREDICTABLE. See the Caution in *TraceEnable and filtering the instruction trace* on page 2-20 for more information.
- From ETMv3.3, an ETM implementation might not support data address comparisons. See *No data address comparator option, ETMv3.3 and later* on page 2-7 for more information.

When a pair of address comparator registers is used to define an address range, the upper Address Comparator Value Register must always contain an address that is greater than the lower Address Comparator Value Register. Otherwise, the behavior of the Address Range Comparators is UNPREDICTABLE.

When you configure a comparator for instruction address comparisons, by setting bit [2] of the Address Access Type Register to 0, you must set the Data Value Comparison field, bits [6:5] of the Address Access Type Register, to b00. Comparator behavior is UNPREDICTABLE if you set this field to any other value.

The use of the address comparator registers is described in *Address comparators* on page 2-36.

Address Comparator Value Registers

An Address Comparator Value Register holds an address to compare against. There are 16 Address Comparator Registers, they are:

- registers `0x10` to `0x1F`, at offsets `0x40` to `0x7C` (in increments of 4) in a memory-mapped implementation
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-21 on page 3-51 shows the bit assignments for an Address Comparator Value Register:

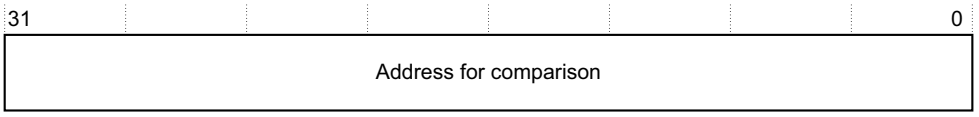


Figure 3-21 Address Comparator Value Registers, bit assignments

Table 3-25 on page 3-49 shows the bit assignments for an Address Comparator Value Register:

Table 3-26 Address Comparator Value Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:0]	v1.0 and later	Address value

Each of the Address Comparator Value Registers has the same bit assignments.

Address Access Type Registers

An Address Access Type Register specifies the type of access, for example instruction or data, and other comparator configuration information, for an address comparison. There are 16 Address Access Type Registers, each corresponding to one of the sixteen Address Comparator Value Registers, see *Address Comparator Value Registers* on page 3-50. The Address Access Type Registers are:

- registers `0x20` to `0x2F`, at offsets `0x80` to `0xBC` (in increments of 4) in a memory-mapped implementation
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-22 shows the bit assignments for an Address Access Type Register:

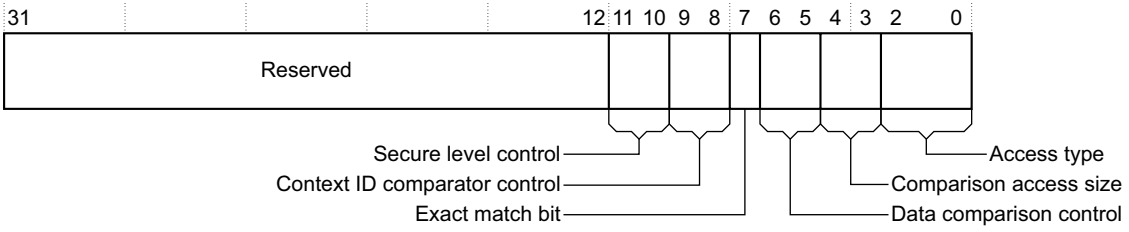


Figure 3-22 Address Access Type Registers, bit assignments

Table 3-25 on page 3-49 shows the bit assignments for an Address Access Type Register:

Table 3-27 Address Access Type Registers, bit assignments

Bit	Version[a]	Description
[31:12]	-	Reserved
[11:10]	v3.2	Security level control. The permitted values of this field are: **b00** Security level ignored. **b01** Match only if in Non-secure state. **b10** Match only if in Secure state. The value of b11 is reserved and must not be used. This field is available only if the connected processor implements the Security Extensions. If the Security Extensions are not implemented writes to these bits are ignored and, if the Address Access Type Register is read/write, they Read-As-Zero.
[9:8]	v2.0	Context ID comparator control. The permitted values of this field are: **b00** Ignore Context ID comparator. **b01** Address comparator matches only if Context ID comparator value 1 matches. **b10** Address comparator matches only if Context ID comparator value 2 matches. **b11** Address comparator matches only if Context ID comparator value 3 matches.
[7]	v2.0	Exact match bit. Specifies comparator behavior when exceptions, aborts, and load misses occur. See *Exact matching, ETMv2.0 and later* on page 2-44.
[6:5]	v1.0	Data value comparison control. The permitted values of this field are: **b00** No data value comparison is made. **b01** Comparator can match only if data value matches. **b11** Comparator can match only if data value does not match. The value of b10 is reserved and must not be used. **Note** The b11 encoding was introduced in ETM architecture version 1.2. Previously this value was reserved. For details of the effect of this field on data value comparison, see *Exact matching for data address comparisons* on page 2-46.

Table 3-27 Address Access Type Registers, bit assignments (continued)

Bit	Version[a]	Description
[4:3]	v1.0	Comparison access size. The permitted values of this field are: **b00** Java instruction (from ETM architecture version 1.3 only) or byte data. **b01** Thumb instruction or halfword data. **b11** ARM instruction or word data. The value of b10 is reserved and must not be used. For more information, see *Comparator access size* on page 2-36.
[2:0]	v1.0	Access type. The permitted values of this field are: **b000**[b] Instruction fetch. **b001** Instruction execute. **b010** Instruction executed and passed condition code test. **b011** Instruction executed and failed condition code test. **b100** Data load or store. **b101** Data load. **b110** Data store. The value of b111 is reserved and must not be used. **Note** • The b010 and b011 encodings were introduced in ETM architecture version 1.2. Previously these values were reserved. • From ETMv3.3, if data address comparisons are not supported, writing b100, b101 or b110 to this field causes UNPREDICTABLE behavior. See *No data address comparator option, ETMv3.3 and later* on page 2-7 for more information.

a. The first ETM architecture version that defines the field.
b. Unsupported if bit [17] of the System Configuration Register, `0x05` is set. See *System Configuration Register, ETMv1.2 and later* on page 3-35.

Each of the Address Access Type Registers has the same bit assignments.

Note

Some ETM documentation describes the Address Access Type Registers as Address Comparator Access Type Registers.

Access types for address range comparators

If you are using two address comparators as an address range comparator, the access type must be identical for each, otherwise the behavior of the comparator is UNPREDICTABLE. The only exceptions to this are:

- Bits [6:5] must be set only for the first comparator in the pair. These bits control data value comparisons.
- The special case where the range includes the address `0xFFFFFFFF`. See *Selecting a range to include address 0xFFFFFFFF*.

Note

This information is also given in *Address comparators* on page 2-36.

Selecting a range to include address 0xFFFFFFFF

Ranges are defined to be exclusive of the upper address, so if you specify an upper address of `0xFFFFFFFF`, only addresses up to and including `0xFFFFFFFE` match. To specify a data address to include `0xFFFFFFFF`, configure the upper address comparator as follows:

- Value Register = `0xFFFFFFFF`
- set Access Type Register bits [4:3] (size mask) to b11.

This is the only case where the size mask can be different between the two address comparators of an address range comparator.

For more information, see *Address range comparators* on page 2-6.

3.5.11 Data value comparator registers

Two registers are defined for each data value comparator. They are described in the following sections:

- *Data value comparator value registers* on page 3-55
- *Data value comparator mask registers* on page 3-56.

Up to eight data value comparators can be present.

Note

From ETMv3.3, whether an ETM macrocell supports data address comparisons is IMPLEMENTATION DEFINED. If data address comparisons are not implemented then the data value comparator registers are not implemented and Read-As-Zero. See *No data address comparator option, ETMv3.3 and later* on page 2-7 for more information.

The data value comparator registers are defined as even-numbered registers, as shown in Table 3-28:

Table 3-28 Summary of the data value comparator registers

Data Value Comparator	Data Comparator Value Register		Data Comparator Mask Register	
	Number[a]	Offset[c]	Number[b]	Offset[c]
1	0x30	0xC0	0x40	0x100
2	0x32	0xC8	0x42	0x108
3	0x34	0xD0	0x44	0x110
4	0x36	0xD8	0x46	0x118
5	0x38	0xE0	0x48	0x120
6	0x3A	0xE8	0x4A	0x128
7	0x3C	0xF0	0x4C	0x130
8	0x3E	0xF8	0x4E	0x138

a. Registers numbered 0x31, 0x33, 0x35, 0x37, 0x39, 0x3B, 0x3D and 0x3F are reserved.
b. Registers numbered 0x41, 0x43, 0x45, 0x47, 0x49, 0x4B, 0x4D and 0x4F are reserved.
c. Register offset when implemented in a memory-mapped scheme. The offset is always 4 x (Register number).

Note

You can use the data value comparator only to observe data. This means that you can only use the data value comparator with load/store accesses. If the data value comparator is enabled and you configure the address comparator to match against instruction addresses, the behavior is UNPREDICTABLE.

The use of the data value comparator registers is described in *Address comparators* on page 2-36.

Data value comparator value registers

The registers that hold the values to be used for data value comparisons are called the Data Comparator Value Registers.

A Data Comparator Value Register holds a 32-bit data value for comparison. There are eight Data Comparator Value Registers. The Data Comparator Value Registers are:

- the even numbered registers from register 0x30 to register 0x3E, as shown in Table 3-28
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-23 shows the bit assignments for a Data Comparator Value Register:

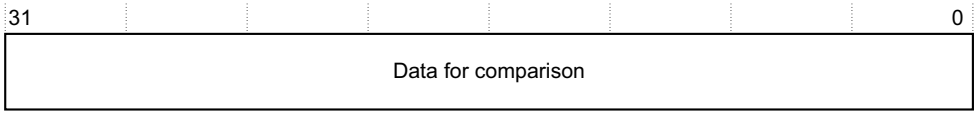


Figure 3-23 Data Comparator Value Registers, bit assignments

Table 3-29 shows the bit assignments for a Data Comparator Value Register:

Table 3-29 Data Comparator Value Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:0]	v1.0 and later	Data value

Each of the Data Comparator Value Registers has the same bit assignments.

Data value comparator mask registers

The registers that hold the masks to be used for data value comparisons are called the Data Comparator Mask Registers.

A Data Comparator Mask Register holds the 32-bit data mask, for use with the data value held in the associated Data Comparator Value Register. There are eight Data Comparator Mask Registers, corresponding to the eight Data Comparator Value Registers. The Data Comparator Mask Registers are:

- the even numbered registers from register `0x40` to register `0x4E`, as shown in Table 3-28 on page 3-55
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-24 shows the bit assignments for a Data Comparator Mask Register:

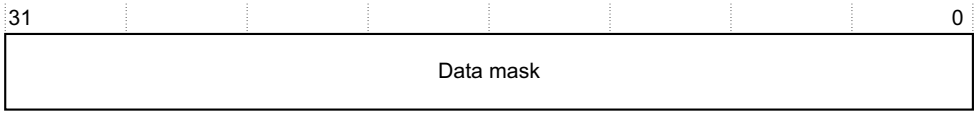


Figure 3-24 Data Comparator Mask Registers, bit assignments

Table 3-30 shows the bit assignments for a Data Comparator Mask Register:

Table 3-30 Data Comparator Mask Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:0]	v1.0 and later	Data mask

Each of the Data Comparator Mask Registers has the same bit assignments.

When a data mask bit is set to 1, the corresponding bit in the Value Register is disregarded in the comparison and must be zero.

Alignment considerations

See *Operation of data value comparators* on page 2-56 for information about alignment considerations when programming the Data Comparator Value and Data Comparator Mask Registers. These considerations are different for different ETM versions, and a summary of the alignment considerations for different ETM versions is given in *Summary of alignment and endianness considerations for different ETM versions* on page 2-61.

Associating data value comparators with address comparators

Each data value comparator is permanently associated with a particular address comparator. Data value comparators are assigned sequentially to odd-numbered address comparators. You cannot have more data value comparators than address comparator pairs.

The Data Comparator Value Register and the Data Comparator Mask Register addresses directly correspond to equivalent Address Comparator Value Register addresses. For example, in a system that uses four pairs of address comparators and two data value comparators (a medium-sized configuration), the address mapping is as shown in Table 3-31.

Table 3-31 Example comparator register associations for a medium-sized configuration

Address comparator		Data value comparator		
Number	Value Register[a]	Present?	Value Register[a]	Mask Register[a]
8	0x17	No	-	-
7	0x16	No	-	-
6	0x15	No	-	-
5	0x14	No	-	-
4	0x13	No	-	-

Table 3-31 Example comparator register associations for a medium-sized configuration (continued)

Address comparator		Data value comparator		
Number	**Value Register[a]**	**Present?**	**Value Register[a]**	**Mask Register[a]**
3	0x12	Yes	0x32	0x42
2	0x11	No	-	-
1	0x10	Yes	0x30	0x40

a. Register numbers are listed. Where registers are accessed in a memory-mapped scheme, the register offset is always 4 x (Register number).

Note

Early versions of this specification permitted more data value comparators than address comparator pairs. The extra data value comparators can be allocated to even-numbered address comparators when all odd-numbered address comparators had a data value comparator allocated, up to a maximum of eight. However, this was never implemented in any of the supported standard configurations and is no longer permitted.

3.5.12 Counter registers

There are between zero and four 16-bit counters. Four registers are used to define the operation of each counter. The following sections describe the counter registers:

- *Counter Reload Value Registers* on page 3-59
- *Counter Enable Registers* on page 3-59
- *Counter Reload Event Registers* on page 3-61
- *Counter Value Registers* on page 3-61.

Table 3-32 summarizes the Counter registers, and gives the address offset for each of the registers:

Table 3-32 Summary of Counter registers

	Counter Registers:			
Counter	**Reload Value[a]**	**Enable[a]**	**Reload Event[a]**	**Value[a]**
1	0x50	0x54	0x58	0x5C
2	0x51	0x55	0x59	0x5D
3	0x52	0x56	0x5A	0x5E
4	0x53	0x57	0x5B	0x5F

a. Register numbers are listed. Where registers are accessed in a memory-mapped scheme, the register offset is always 4 x (Register number).

See *Counters* on page 2-10 for more information on counter registers.

Counter Reload Value Registers

A Counter Reload Value Register specifies the starting value of the associated counter. The counter is automatically loaded with this value when this register is programmed and when the ETM Programming bit is set. It is then reloaded with this value whenever the counter reload event, specified by the Counter Reload Event Register, is active. The Counter Reload Value Registers are:

- registers `0x50` to `0x53`, at offsets `0x140`, `0x144`, `0x148`, and `0x14C` in a memory-mapped implementation
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-25 shows the bit assignments for a Counter Reload Value Register:

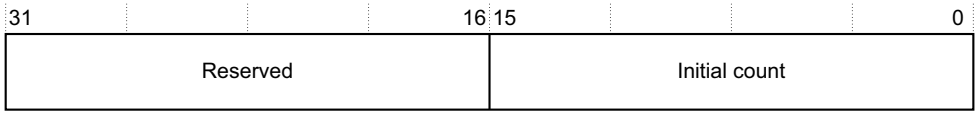


Figure 3-25 Counter Reload Value Registers, bit assignments

Table 3-33 shows the bit assignments for a Counter Reload Value Register:

Table 3-33 Counter Reload Value Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:16]	-	Reserved
[15:0]	v1.0 and later	Initial count

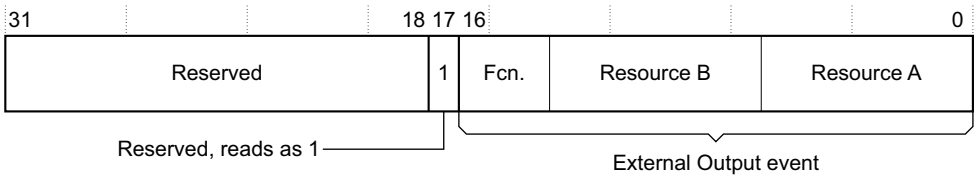
Each of the Counter Reload Value Registers has the same bit assignments.

Counter Enable Registers

A Counter Enable Register defines an event that enables the associated counter. It can also be used to configure the counter for continuous operation. The Counter Enable Registers are:

- registers `0x54` to `0x57`, at offsets `0x150`, `0x154`, `0x158`, and `0x15C` in a memory-mapped implementation
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-26 shows the bit assignments for a Counter Enable Register, for ETM version 2.0 or later:



Shown for ETM architecture v2 and later. See the register bit assignments table for differences in architecture v1.x.

Figure 3-26 Counter Enable Registers, bit assignments

Table 3-34 shows the bit assignments for a Counter Enable Register, and describes the differences in different ETM architecture versions:

Table 3-34 Counter Enable Registers, bit assignments

Bit	Version[a]	Description
[31:18]	-	Reserved.
[17]	v1.x only	Count enable source in ETMv1.x. When set to 0, the counter is continuously enabled and decrements every cycle regardless of the count enable event. When set to 1, the count enable event is used to enable the counter. It is recommended that bit [17] is always set to 1 and that the count enable event is used to control counter operation, using `0x6F` (TRUE) if a free running counter is required. **Note** This bit is not supported in ETMv2.0 and later, and is always set to 1 in these ETM architecture versions.
[16:0]	v1.0	Count enable event. To configure a continuous counter, program these bits for external resource 15. (For the encoding, see *Resource identification* on page 3-108). External resource 15 is hard-wired to be always active.

a. The first ETM architecture version that defines the field, or (where the use of a field is different in different versions) the first architecture version to which the description applies.

Each of the Counter Enable Registers has the same bit assignments.

The section *Using ETM event resources* on page 3-108 describes how you define a counter enable event.

Counter Reload Event Registers

A Counter Reload Event Register defines an event that causes the associated counter to be reloaded with the value held in its Counter Reload Value Register. The Counter Reload Event Registers are:

- registers `0x58` to `0x5B`, at offsets `0x160`, `0x164`, `0x168`, and `0x16C` in a memory-mapped implementation
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-27 shows the bit assignments for a Counter Reload Event Register:

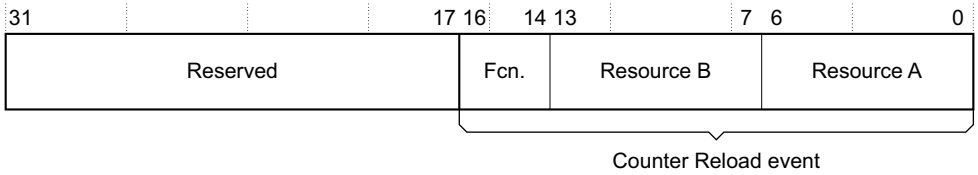


Figure 3-27 Counter Reload Event Registers, bit assignments

Table 3-35 shows the bit assignments for a Counter Reload Event Register:

Table 3-35 Counter Reload Event Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:17]	-	Reserved
[16:0]	v1.0 and later	Counter reload event

Each of the Counter Reload Event Registers has the same bit assignments.

Using ETM event resources on page 3-108 describes how you define a counter reload event.

Counter Value Registers

A Counter Value Register holds the current value of the associated counter. The Counter Value registers are:

- registers `0x5C` to `0x5F`, at offsets `0x170`, `0x174`, `0x178` and `0x17C` in a memory-mapped implementation
- read-only registers in ETM architecture versions 3.0 and earlier
- read/write registers in ETM architecture versions 3.1 and later.

From ETM architecture version 3.1, when the Programming bit is set you can write to an ETM Counter Value Register to set the current value of the counter.

Figure 3-28 on page 3-62 shows the bit assignments for a Counter Value Register:

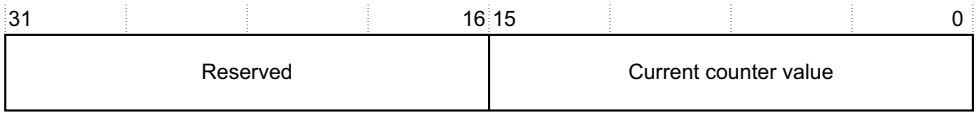


Figure 3-28 Counter Value Registers, bit assignments

Table 3-36 shows the bit assignments for a Counter Value Register:

Table 3-36 Counter Value Registers, bit assignments

Bit	Type	Version[a]	Description
[31:16]	-	-	Reserved.
[15:0]	RO	v1.0 up to v3.0	Current counter value. See *ETM Programming bit and associated state* on page 3-19 for information on programming this register.
	R/W	v3.1	

a. ETM architecture versions to which the *Type* description applies.

Each of the Counter Value Registers has the same bit assignments.

3.5.13 Sequencer registers

An ETM implementation can include a sequencer. If it does, you control the sequencer by defining the events that cause the sequencer to move between the different states.

Each sequencer state transition event has its own register, and these registers are programed to control the state transitions. An additional register holds the current state of the sequencer. The sequencer registers are summarized in Table 3-37 on page 3-63, with the register number and address offset of each register.

Programing any of the Sequencer State Transition Event Registers when the ETM Programming bit is set resets the sequencer to State 1.

When programming the sequencer, you must program a valid encoding into all the State Transition Event Registers, otherwise the behavior of the sequencer is UNPREDICTABLE. For example, if you want the sequencer only to involve transitions between states 1 and 2, you must program registers `0x60` and `0x61` to control transitions between these two states. You must:

- program registers `0x62` and `0x65` to ensure that state 3 is never entered
- program registers `0x63` and `0x64`, typically with the value `0x0000406F`, to ensure these transitions never occur.

Table 3-37 Sequencer register allocation

Register Number	Offset[b]	Type	Version[a]	Description
0x60	0x180	WO[c]	v1.0	State 1 to State 2 Transition Event Register[d]
0x61	0x184	WO[c]	v1.0	State 2 to State 1 Transition Event Register[d]
0x62	0x188	WO[c]	v1.0	State 2 to State 3 Transition Event Register[d]
0x63	0x18C	WO[c]	v1.0	State 3 to State 1 Transition Event Register[d]
0x64	0x190	WO[c]	v1.0	State 3 to State 2 Transition Event Register[d]
0x65	0x194	WO[c]	v1.0	State 1 to State 3 Transition Event Register[d]
0x66	0x198	-	-	Reserved
0x67	0x19C	RO	v1.0	Current Sequencer State Register, see *Current Sequencer State Register* on page 3-64
		R/W	v3.1	

a. The first ETM architecture version that defines the field, or (where the field *Type* is different in different versions) the first architecture version to which the *Type* description applies.
b. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4 x (Register number).
c. In ETMv3.1 and later, these bits are read-write if bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.
d. Each of the Sequencer State Transition Event Registers has the same bit assignments. See *Sequencer State Transition Event Registers* for details.

Sequencer State Transition Event Registers

A Sequencer State Transition Event Register defines the event that causes the associated sequencer state transition.

- the register numbers, corresponding sequencer transitions, and offset addresses in a memory-mapped implementation, are listed in Table 3-32 on page 3-61
- the registers are write-only, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-29 on page 3-64 shows the bit assignments for a Sequencer State Transition Event Register:

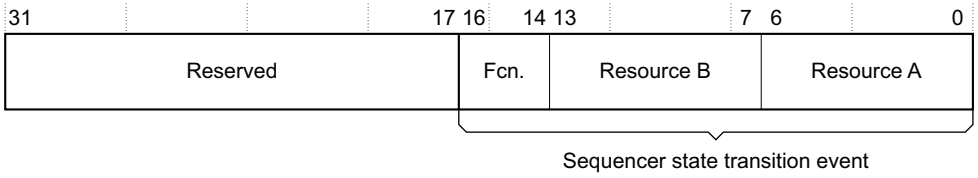


Figure 3-29 Sequencer State Transition Event Registers, bit assignments

Table 3-38 list the bit assignments for a Sequencer State Transition Event Register:

Table 3-38 Sequencer State Transition Event Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:17]	-	Reserved
[16:0]	v1.0 and later	State transition event

Each of the Sequencer State Transition Event Registers has the same bit assignments.

Using ETM event resources on page 3-108 describes how you define a sequencer state transition event.

Current Sequencer State Register

The Current Sequencer State Register holds the current state of the sequencer. It is:

- register `0x67`, at offset `0x19c` in a memory-mapped implementation
- read-only in ETM architecture versions 3.0 and earlier
- read/write in ETM architecture versions 3.1 and later.

In ETM architecture version 3.1 and later, when the Programming bit is set, you can write to this register to force the sequencer to a particular state.

Note

The state field is two bits. However, a value of b11 is UNPREDICTABLE, and in implementations where the register is read/write you must not write a value of b11 to this field.

Figure 3-29 shows the bit assignments for the Current Sequencer State Register:

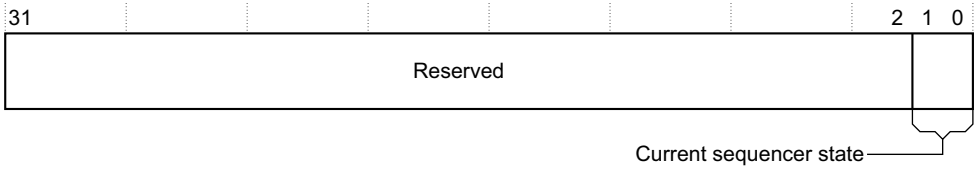


Figure 3-30 Current Sequencer State Register bit assignments

Table 3-39 shows the bit assignments for the Current Sequencer State Register:

Table 3-39 Current Sequencer State Register bit assignments

Bit	Defined in ETM architecture versions	Description
[31:2]	-	Reserved
[1:0]	v1.0 and later	The permitted values of this field are: **b00** Sequencer currently in state 1. **b01** Sequencer currently in state 2. **b10** Sequencer currently in state 3. The value of b11 is reserved and must not be used.

See *ETM Programming bit and associated state* on page 3-19 for information on programming this register.

3.5.14 External Output Event Registers

Each of the four External Output Event Registers defines the event that controls the corresponding external output. The registers are:

- registers `0x68-0x6B`, at offsets `0x1A0`, `0x1A4`, `0x1A8`, and `0x1AC` in a memory-mapped implementation
- write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-31 shows the bit assignments for an External Output Event Register.

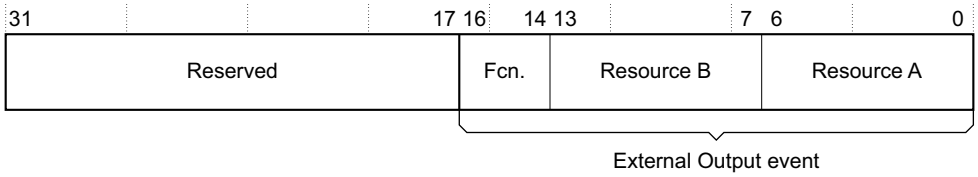


Figure 3-31 External Output Event Registers, bit assignments

Table 3-40 shows the bit assignments for the External Output Event Registers:

Table 3-40 External Output Event Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:17]	-	Reserved
[16:0]	v1.0 and later	External output event

Each of the External Output Event Registers has the same bit assignments.

The section *Using ETM event resources* on page 3-108 describes how you define a counter reload event.

3.5.15 Context ID comparator registers, ETMv2.0 and later

Four registers are used to define expected Context ID values. These registers are summarized in Table 3-41:

Table 3-41 Summary of the Context ID comparator registers

Description	Register number	Offset[a]
Context ID Comparator Value 1 Register	0x6C	0x1B0
Context ID Comparator Value 2 Register	0x6D	0x1B4
Context ID Comparator Value 3 Register	0x6E	0x1B8
Context ID Comparator Mask Register	0x6F	0x1BC

a. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4 x (Register number).

These registers are write-only, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

These registers are described in:

- *Context ID Comparator Value Registers*
- *Context ID Comparator Mask Register* on page 3-67.

Context ID Comparator Value Registers

There are three Context ID Comparator Value Registers. Each register holds a 32-bit Context ID value. These registers:

- have register numbers, and offset addresses in a memory-mapped implementation, listed in Table 3-41
- are only available from ETMv2.0
- are write-only registers, although in architecture versions 3.1 and later they are read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-32 on page 3-67 shows the bit assignments for a Context ID Comparator Value Register:

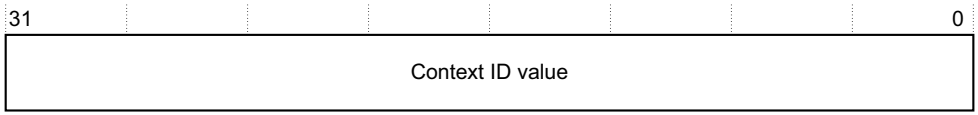


Figure 3-32 Context ID Comparator Value Registers, bit assignments

Table 3-42 shows the bit assignments for a Context ID Comparator Value Register:

Table 3-42 Context ID Comparator Value Registers, bit assignments

Bit	Defined in ETM architecture versions	Description
[31:0]	v2.0 and later	Context ID value

Each of the Context ID Comparator Value Registers has the same bit assignments.

Context ID Comparator Mask Register

The Context ID Comparator Mask Register holds the 32-bit Context ID mask. It is:

- register `0x6F`, at offset `0x1BC` in a memory-mapped implementation
- only available in ETMv2.0 or later
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-33 shows the bit assignments for the Context ID Comparator Mask Register:

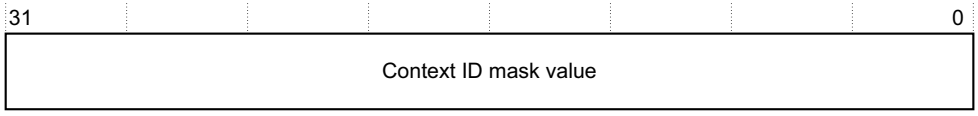


Figure 3-33 Context ID Comparator Mask Register bit assignments

Table 3-43 shows the bit assignments for the Context ID Comparator Mask Register:

Table 3-43 Context ID Comparator Mask Register bit assignments

Bit	Defined in ETM architecture versions	Description
[31:0]	v2.0 and later	Context ID mask value

The same mask is used for each Context ID comparator. When a Context ID mask bit is set to 1, the corresponding bit in the Value Register is disregarded in the comparison and must be zero.

This register is present only if at least one Context ID comparator is present.

3.5.16 IMPLEMENTATION SPECIFIC registers

Register numbers `0x70-0x77` in the register map are reserved for the future implementation of up to eight application-specific registers. Even when an ETM does not implement these registers, IMPLEMENTATION SPECIFIC Register 0, register number `0x70`, must be partially defined, so that a debugger can implement a general mechanism for detecting IMPLEMENTATION SPECIFIC extensions.

This register area is write-only, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

IMPLEMENTATION SPECIFIC Register 0

IMPLEMENTATION SPECIFIC Register 0 is used to show the presence of any IMPLEMENTATION SPECIFIC features, and to enable any features that are provided. In all current implementations it indicates that no IMPLEMENTATION SPECIFIC features are provided. The register is:

- register `0x70`, at offset `0x1C0` in a memory-mapped implementation
- only available in ETMv2.0 or later
- a write-only register, although in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-34 shows the bit assignments for the IMPLEMENTATION SPECIFIC Register 0:

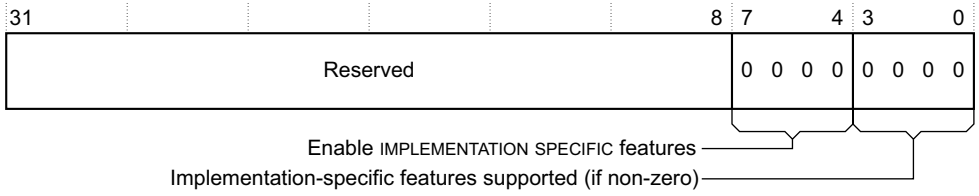


Figure 3-34 IMPLEMENTATION SPECIFIC Register 0 bit assignments

Table 3-44 shows the bit assignments for the IMPLEMENTATION SPECIFIC Register 0:

Table 3-44 IMPLEMENTATION SPECIFIC Register 0 bit assignments

Bits	Version[a]	Type	Description
[31:8]	-	-	Reserved.
[7:4]	v2.0	R/W	Enable IMPLEMENTATION SPECIFIC extensions. The ETM must behave as if the IMPLEMENTATION SPECIFIC extensions are not implemented when these bits are b0000. The behavior of the ETM is IMPLEMENTATION DEFINED when these bits are set to any value other than b0000. On an ETM reset these bits are cleared to b0000.
[3:0]	v2.0	RO	If this field is b0000 then no IMPLEMENTATION SPECIFIC extensions are supported. Other values are for use only as permitted in writing by ARM Limited.

a. The first ETM architecture version that defines the field.

Note

Trace debug tools might require application-specific modifications to support any added functionality.

3.5.17 Synchronization Frequency Register, ETMv2.0 and later

The Synchronization Frequency Register holds the trace synchronization frequency value. It is:

- register `0x78`, at offset `0x1E0` in a memory-mapped implementation
- only available in ETMv2.0 or later
- a write-only register, although:
  - in architecture versions 3.1 and later it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76
  - in architecture version 3.4 and later, it can be implemented as a read-only register, see *Finding the access type, ETMv3.4 and later* on page 3-70.

Figure 3-35 on page 3-70 shows the bit assignments for the Synchronization Frequency Register, with the default value of the register:

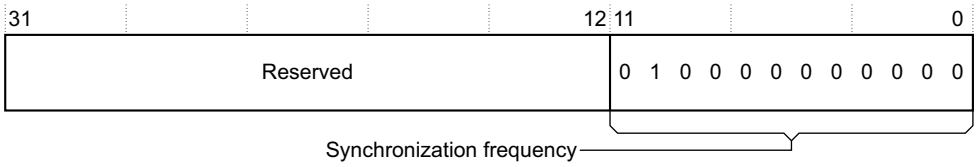


Figure 3-35 Synchronization Frequency Register bit assignments

Table 3-45 shows the bit assignments for the Synchronization Frequency Register:

Table 3-45 Synchronization Frequency Register bit assignments

Bit	Defined in ETM architecture versions	Description
[31:12]	-	Reserved
[11:0]	v2.0 and later	Synchronization frequency. Default value is 1024.

In ETMv2.0 and later, when a *Trace FIFO Offset* (TFO) has occurred, the TFO counter is reset to the value that is programmed into the Synchronization Frequency Register. This value is the time between synchronization points in the trace (the tools can start decompressing only at synchronization points). Depending on the protocol version, the time is measured in cycles or bytes. The default value is 1024.

Note

This value must be set to a value greater than the size of the FIFO.

An implementation might not implement the bottom bits of this register, because of limitations in the accuracy of the synchronization frequency. In this case, a value read from this register might be different from the value written to it.

From ETMv3.4, an ETM implementation can implement a fixed synchronization frequency of 1024. In this case the Synchronization Frequency Register is implemented as a read-only register, that always returns the value 1024 (`0x00000400`) on reads. For more information see *Finding the access type, ETMv3.4 and later*.

This register is used to control TFOs in ETMv2. See *Trace FIFO offsets* on page 6-14. This register is used to control A-sync, I-sync, and D-sync in ETMv3. See *Synchronization* on page 7-65.

Finding the access type, ETMv3.4 and later

From ETMv3.4, the Synchronization Frequency Register can be implemented as either:

- a write-only register that is read/write when bit [11] of the Configuration Code Extension Register is set to 1, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76
- a read-only register, if the Synchronization Frequency is fixed at 1024.

To find out how the register has been implemented, in ETMv3.4 or later:

1. Make sure that bit [11] of the Configuration Code Extension Register is set to 1. This means that, if the implementation permits the Synchronization Frequency to be changed, you can write to the Synchronization Frequency Register.
2. Read the value of the Synchronization Frequency Register. You might have to restore this value later.
3. Write the value `0xFFFFFFFF` to the Synchronization Frequency Register.
4. Read the value of the Synchronization Frequency Register again:
   - If this value is `0x00000400` then the register is implemented as read-only.
   - If the value is not `0x00000400` then the register is implemented as read/write, write only if bit [11] of the Configuration Code Extension Register is set to 0.

     Note

     When the register is implemented as read/write, it is still unlikely that this second read of the register will return `0xFFFFFFFF`, because:
     — bits [31:12] of the register are reserved and might read-as-zero
     — the bottom bits of the register might not be implemented.

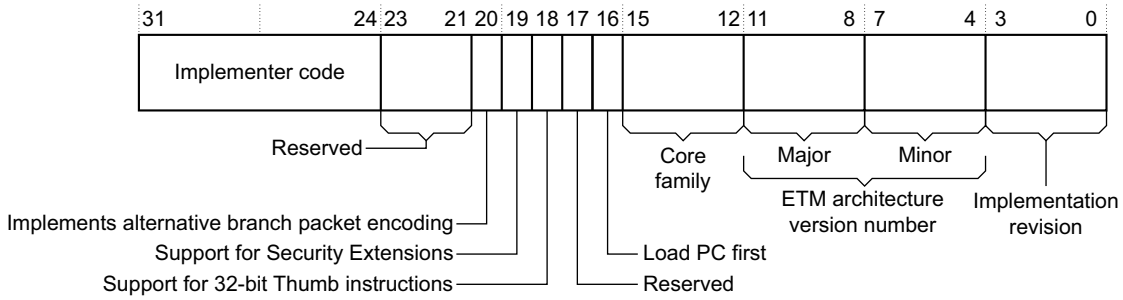
     Your check *must not* expect the read at stage 4 to match the value written at stage 3.
5. If the Synchronization Frequency Register is implemented as read/write, write the value from stage 3 back to the register.

3.5.18 ETM ID Register, ETMv2.0 and later

The ETM ID Register holds the ETM architecture variant, and precisely defines the programmer's model for the ETM. It is:

- register `0x79`, at offset `0x1E4` in a memory-mapped implementation
- only available in ETMv2.0 or later
- a read-only register
- only valid when bit 31 in the ETM Configuration Code Register is set, see *ETM Configuration Code Register* on page 3-29
  — in other cases the architecture version is given in the ETM Configuration Code Register.

Figure 3-36 on page 3-72 shows the bit assignments for the ETM ID Register, for ETM architecture version 3.4:



Shown for ETM architecture v3.4. See the register bit assignments table for differences in other architecture versions

Figure 3-36 ETM ID Register bit assignments, for ETM architecture v3.4

Table 3-46 shows the bit assignments for the ETM ID Register, and describes the differences between different ETM architecture versions:

Table 3-46 ETM ID Register bit assignments

Bit	Version[a]	Description
[31:24]	v2.0	Implementer code. The following codes are defined[b], all other values are reserved by ARM Limited: `0x41` ASCII code for A, indicating ARM Limited. `0x44` ASCII code for D, indicating Digital Equipment Corporation. `0x4D` ASCII code for M, indicating Motorola, Freescale Semiconductor Inc. `0x51` ASCII code for Q, indicating QUALCOMM Inc. `0x56` ASCII code for V, indicating Marvell Semiconductor Inc. `0x69` ASCII code for i, indicating Intel Corporation.
[23:21]	-	Reserved.
[20]	v3.4	Branch packet encoding implemented. The possible values of this bit are: **0** The ETM implements the original branch packet encoding, see *Branch address compression, original scheme* on page 7-16. **1** The ETM implements the alternative branch packet encoding, see *Branch packet formats with the alternative address compression scheme* on page 7-18.
[19]	v3.2	Support for Security Extensions. The possible values of this bit are: **0** The ETM behaves as if the processor is in Secure state at all times. **1** The ARM architecture Security Extensions are implemented by the processor.

Table 3-46 ETM ID Register bit assignments (continued)

Bit	Version[a]	Description
[18]	v3.2	Support for 32-bit Thumb instructions. The possible values of this bit are: **0** A 32-bit Thumb instruction is traced as two instructions, and exceptions might occur between these two instructions. **1** A 32-bit Thumb instruction is traced as a single instruction. See *32-bit Thumb instructions* on page 4-13 for more information.
[17]	-	Reserved.
[16]	v2.1	Load PC first. If this bit is set to 1, `LSM`s with the PC in the list load the PC first, followed by the other registers in the normal order. This can be decompressed by using the following procedure: 1. Calculate the number of items transferred by the `LSM` by looking at the code image. 2. As each item is read, assign an address equal to 4 greater than the previous one as normal. 3. When the number of items read equals the total number of items transferred, subtract (4 * number of items) from each address other than the first. **Note** This means that a branch address can be traced before the remaining data values of an instruction. While this has never been prohibited in the protocol, care must be taken to ensure that this case is correctly handled.
[15:12]	v2.0	Core family. The meaning of this field depends on the value of the Implementer code.The following apply if Implementer code = `0x41`, for ARM Limited: **b0000** ARM7 core. **b0001** ARM9 core. **b0010** ARM10 core. **b0011** ARM11 core **b1111** Core family is defined elsewhere, see *The Core family field* on page 3-76 for more information. When the Implementer code = `0x41`, all other values are reserved by ARM Limited. For any other Implementer code the permitted values of this field are defined by the implementer.

Table 3-46 ETM ID Register bit assignments (continued)

Bit	Version[a]	Description
[11:8]	v2.0	Major ETM architecture version number, see *The ETM architecture version*. Possible values of this field are: **b0000** ETMv1. **b0001** ETMv2. **b0010** ETMv3. All other values are reserved.
[7:4]	v2.0	Minor ETM architecture version number, see *The ETM architecture version*.
[3:0]	v2.0	Implementation revision.

a. The first ETM architecture version that defines the field.
b. The Implementer code list applies to processors as well as ETMs. This list does not indicate the implementers of ETMs.

The ETM architecture version

In the ETM ID Register, the ETM architecture version is encoded as `ETMvX.Y`, where:

- `(X-1)` = the value encoded in register bits [11:8]
- `Y` = the value encoded in register bits [7:4].

For protocol versions up to 3, previous versions of this specification referred to protocol numbers and made no reference to ETMv2. To enable independent evolution of ETMs in different product families and to provide better information to tools using the ETM, protocol numbers are replaced with major and minor architecture version numbers. If a protocol number is referred to as a characteristic of an ETM implementation, the major architecture version of that implementation is 1.

An ID Register value of 0 indicates that the ETM is not present. This can be returned by the coprocessor interface in cores supporting ARM architecture v6 and later.

Table 3-47 shows the ETM ID Register values for ETMs described in this specification and implemented by ARM Limited.

Table 3-47 ID values for different ETM variants

Implementation	ID value	Architecture version	Protocol number (deprecated)
ETM not present	`0x00000000`[a]	-	-
ETM7 Rev 0	`0x41000010`[b]	ETMv1.1	1
ETM7 Rev 1	`0x41000020`[b]	ETMv1.2	2
ETM7 Rev 1a	`0x41000021`[b]	ETMv1.2	4

Table 3-47 ID values for different ETM variants (continued)

Implementation	ID value	Architecture version	Protocol number (deprecated)
ETM9 Rev 0	0x41001000[b]	ETMv1.0	0
ETM9 Rev 0a	0x41001010[b]	ETMv1.1	1
ETM9 Rev 1	0x41001020[b]	ETMv1.2	2
ETM9 Rev 2	0x41001030[b]	ETMv1.3	3
ETM9 Rev 2a	0x41001031[b]	ETMv1.3	5
ETM9 r2p2	0x41001032[b]	ETMv1.3	7
CoreSight ETM9 r0p0	0x41001220	ETMv3.2	-
ETM10 Rev 0	0x41002100	ETMv2.0	-
ETM10RV Rev 0	0x41002200	ETMv3.0	-
ETM11RV r0p0	0x41003210[c]	ETMv3.1	-
ETM11RV r0p1	0x41013211	ETMv3.1	-
CoreSight ETM11 r0p0	0x41013220 0x41053220 0x41093220	ETMv3.2	-
CoreSight ETM-R4	0x4104F230	ETMv3.3	-
CoreSight ETM-A8	0x410CF230	ETMv3.3	-
CoreSight ETM-M3	0x4114F240	ETMv3.4	-

a. Returned from CP14 access.
b. These ETMs do not have an ETM ID Register. Bit [31] of the ETM Configuration Code Register is 0 and the minor architecture version number is given in that register (see Table 3-10 on page 3-30). The value provided here is for illustration.
c. Bit [16], Load PC first, is not set to 1 in this revision, but should be. LSMs with the PC in the list load the PC first.

Note

Tools must determine the programmer's model from the major and minor architecture version numbers alone where possible. The Core family field must not be used to determine aspects of ETM behavior.

The Core family field

From ETMv3, where the Implementer code field, bits [31:24], is `0x41`, ARM Limited recommends that debug tools do not use the Core family field, bits [15:12], to discover information about the connected core. Instead, the tools must interrogate the core directly, for example by reading its identification registers. See the *Technical Reference Manual* of the appropriate core for more information.

From ETMv3.3, macrocells implemented by ARM Limited normally return 4'b1111 in this field, meaning that the core family is identified elsewhere.

Note

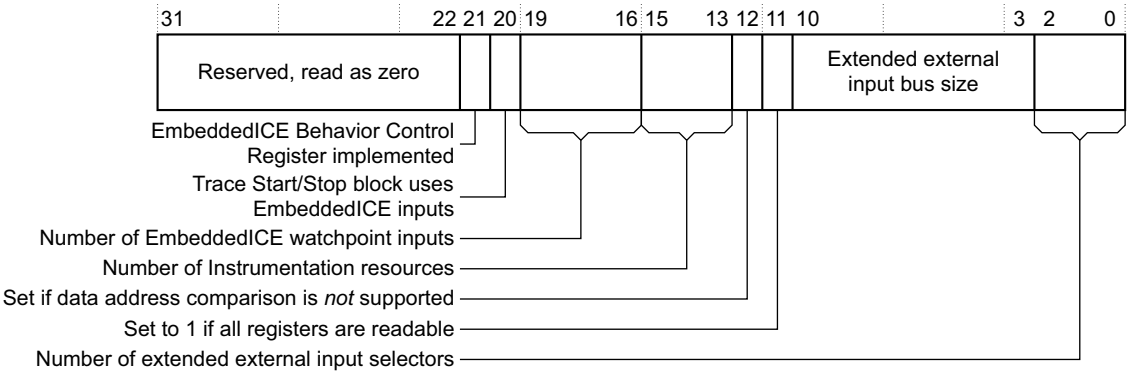
To find the core family, you can read the JTAG IDCODE of the core, if this feature is implemented. See the *Technical Reference Manual* for the core for more information.

3.5.19 Configuration Code Extension Register, ETMv3.1 and later

The Configuration Code Extension Register holds additional bits for ETM configuration code. This register describes the extended external inputs. It is:

- register `0x7A`, at offset `0x1E8` in a memory-mapped implementation
- only available in ETMv3.1 or later
- a read-only register.

Figure 3-36 on page 3-72 shows the bit assignments for the Configuration Code Extension Register, for ETMv3.4 and later:



Shown for ETM architecture v3.4. See the register bit assignments table for differences in other architecture versions.

Figure 3-37 Configuration Code Extension Register bit assignments, ETMv3.4 and later

Table 3-48 shows the bit assignments for the Configuration Code Extension Register:

Table 3-48 Configuration Code Extension Register bit assignments

Bits	Version[a]	Description
[31:22]	-	Reserved. Read-As-Zero.
[21]	v3.4	EmbeddedICE Behavior Control Register implemented. This bit is 1 if the register is implemented, and 0 if it is not implemented.
[20]	v3.4	Trace Start/Stop block can use EmbeddedICE watchpoint inputs. This bit is 1 if the Trace Start/Stop block can used these inputs, and is 0 otherwise.
[19:16]	v3.4	Number of EmbeddedICE watchpoint inputs implemented. This field can take any value from b0000 (0 inputs) to b1000 (8 inputs).
[15:13]	v3.3	Number of Instrumentation resources supported. The maximum value of this field is b100, for four Instrumentation resources. For more information see *Instrumentation resources, from ETMv3.3* on page 2-63.
[12]	v3.3	Set to 1 if data address comparisons are not supported. For more information see *No data address comparator option, ETMv3.3 and later* on page 2-7.
[11]	v3.1	Set to 1 if all registers are readable.
[10:3]	v3.1	Size of extended external input bus. This field must be 0 if bits [2:0] are 0.
[2:0]	v3.1	Number of extended external input selectors.

a. The first ETM architecture version that defines the field.

3.5.20 Extended External Input Selection Register, ETMv3.1 and later

The Extended External Input Selection Register specifies the extended external inputs, see *External inputs* on page 2-11 for more information. It is:

- register `0x7B`, at offset `0x1EC` in a memory-mapped implementation
- only available in ETMv3.1 or later
- a write-only register, although it is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-38 on page 3-78 shows the bit assignments for the Extended External Input Selection Register:

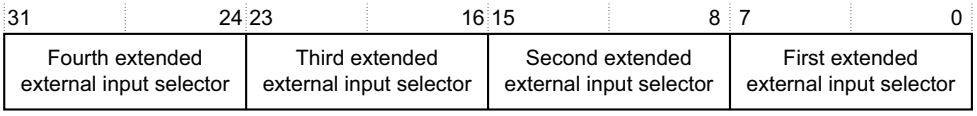


Figure 3-38 Extended External Input Selection Register bit assignments

Table 3-49 shows the bit assignments for the Extended External Input Selection Register:

Table 3-49 Extended External Input Selection Register bit assignments

Bits	Version[a]	Description
[31:24]	v3.1	Fourth extended external input selector.
[23:16]	v3.1	Third extended external input selector.
[15:8]	v3.1	Second extended external input selector.
[7:0]	v3.1	First extended external input selector.

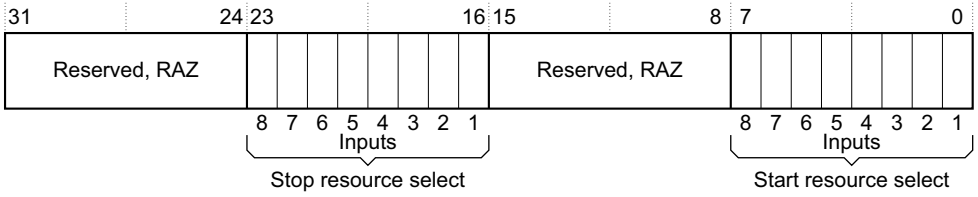
a. The first ETM architecture version that defines the field.

3.5.21 Trace Start/Stop EmbeddedICE Control Register, ETMv3.4 and later

The Trace Start/Stop EmbeddedICE Control Register specifies the EmbeddedICE watchpoint comparator inputs that are used as trace start and stop resources. It is:

- register `0x7C`, at offset `0x1F0` in a memory-mapped implementation
- only available in ETMv3.4 or later
- a write-only register, that is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-39 shows the bit assignments for the Trace Start/Stop EmbeddedICE Control Register:



Set bits to 1 to select EmbeddedICE watchpoint inputs, as numbered, as **TraceEnable** start or stop resources. For example, setting bit [2] to 1 selects EmbeddedICE watchpoint 3 as a **TraceEnable** start resource.

Figure 3-39 Trace Start/Stop EmbeddedICE Control Register bit assignments

Table 3-50 shows the bit assignments for the Trace Start/Stop EmbeddedICE Control Register:

Table 3-50 Trace Start/Stop EmbeddedICE Control Register bit assignments

Bit	Version[a]	Description
[31:24]	-	Reserved, Read-as-zero.
[23:16]	v3.4	Stop resource selection. Setting a bit in this field to 1 selects the corresponding EmbeddedICE watchpoint input as a **TraceEnable** stop resource. Bit [16] corresponds to input 1, bit [17] to input 2, and this pattern continues up to bit [23] corresponding to input 8.
[15:8]	-	Reserved, Read-as-zero.
[7:0]	v3.4	Start resource selection. Setting a bit in this field to 1 selects the corresponding EmbeddedICE watchpoint input as a **TraceEnable** start resource. Bit [0] corresponds to input 1, bit [1] to input 2, and this pattern continues up to bit [7] corresponding to input 8.

a. The first ETM architecture version that defines the field.

3.5.22 EmbeddedICE Behavior Control Register, ETMv3.4 and later

The EmbeddedICE Behavior Control Register controls the sampling behavior of the EmbeddedICE watchpoint comparator inputs. It is:

- Register `0x7D`, at offset `0x1F4` in a memory-mapped implementation.
- Only available in ETMv3.4 or later.
- An optional register. Bit [21] of the Configuration Code Extension Register is set to 1 if the EmbeddedICE Behavior Control Register is implemented.
- A write-only register, that is read/write when bit [11] of the Configuration Code Extension Register is set, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Figure 3-40 shows the bit assignments for the EmbeddedICE Behavior Control Register:

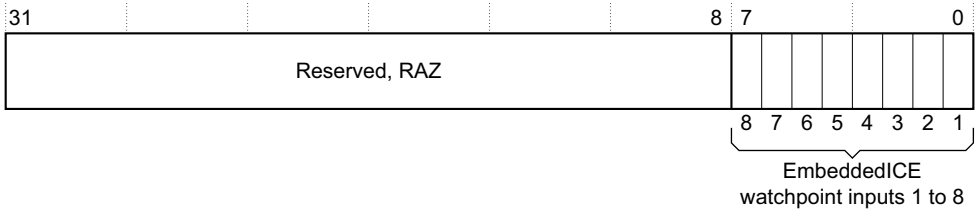


Figure 3-40 EmbeddedICE Behavior Control Register bit assignments

Table 3-51 shows the bit assignments for the EmbeddedICE Behavior Control Register:

Table 3-51 EmbeddedICE Behavior Control Register bit assignments

Bit	Version[a]	Description
[31:8]	-	Reserved, Read-as-zero.
[7:0]	v3.4	EmbeddedICE watchpoint input sampling behavior. Each bit controls the sampling behavior of one of the EmbeddedICE watchpoint inputs: **Bit == 0** When sampled, the corresponding input is pulsed for a single sample. **Bit == 1** When sampled, the corresponding input is latched and held until one cycle before the next sampling point. Bit [0] corresponds to input 1, bit [1] to input 2, and this pattern continues up to bit [7] corresponding to input 8.

a. The first ETM architecture version that defines the field.

For more information about the behavior of the EmbeddedICE watchpoint comparator inputs see *Behavior of EmbeddedICE inputs, from ETMv3.4* on page 7-62.

Note

From ETMv3.4, if the EmbeddedICE Behavior Control Register is not implemented, the EmbeddedICE watchpoint comparator inputs must behave as described in *Default behavior of EmbeddedICE watchpoint inputs* on page 7-63.

3.5.23 CoreSight Trace ID Register, ETMv3.2 and later

The CoreSight Trace ID Register defines the 7-bit Trace ID, for output to the trace bus. It is:

- register `0x80`, at offset `0x200` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read/write register.

This register is used in systems where multiple trace sources are present and tracing simultaneously. For example, where an ETM outputs trace onto the AMBA 3 *Advanced Trace Bus* (ATB), a unique ID is required for each trace source so that the trace can be uniquely identified as coming from a particular trace source. For more information about the AMBA 3 ATB, see the *CoreSight Architecture Specification*.

Figure 3-41 on page 3-81 shows the bit assignments for the CoreSight Trace ID Register:

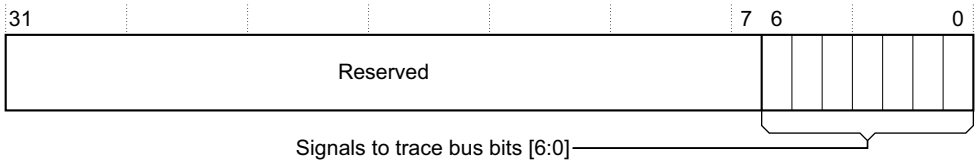


Figure 3-41 CoreSight Trace ID Register bit assignments

Table 3-52 shows the bit assignments for the CoreSight Trace ID Register:

Table 3-52 CoreSight Trace ID Register bit assignments

Bit	Version[a]	Description
[31:7]	-	Reserved.
[6:0]	v3.2	Trace ID to output onto the trace bus. On an ETM reset this field is cleared to `0x00`.

a. The first ETM architecture version that defines the field.

3.5.24 Operating System Save and Restore Registers, ETMv3.3 and later

From ETMv3.3, these registers are provided for saving the entire ETM state of the processor before it is powered down. The use of these registers is described in *Power-down support, ETMv3.3 and later* on page 3-119. The registers are described in the following sections:

- *OS Lock Access Register (OSLAR), ETMv3.3 and later*
- *OS Lock Status Register (OSLSR), ETMv3.3 and later* on page 3-82
- *OS Save and Restore Register (OSSRR), ETMv3.3 and later* on page 3-84.

OS Lock Access Register (OSLAR), ETMv3.3 and later

The OS Lock Access Register (OSLAR) is used to lock access to the ETM trace registers. It is:

- register `0x0C0`, at offset `0x300` in a memory-mapped implementation
- only available in ETMv3.3 or later
- a write-only register, that reads as zero (RAZ).

See Table 3-4 on page 3-16 for details of how the ETM registers are split into trace and management registers.

Figure 3-42 on page 3-82 shows the bit assignments for the OS Lock Access Register (OSLAR):

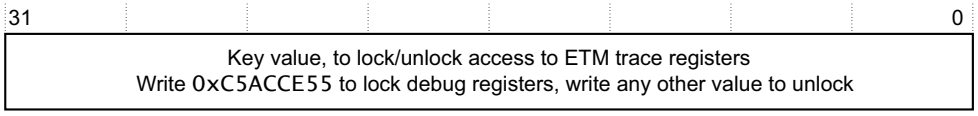


Figure 3-42 OS Lock Access Register (OSLAR) bit assignments

Table 3-53 shows the bit assignments for the OS Lock Access Register (OSLAR):

Table 3-53 OS Lock Access Register (OSLAR) bit assignments

Bit	Version[a]	Description
[31:0]	v3.3	Write `0xC5ACCE55` to this field to lock the ETM trace registers. Write any other value to this field to unlock the ETM trace registers.

a. The first ETM architecture version that defines the field.

When the ETM trace registers are locked, any attempt to access the locked registers returns a slave-generated error response. See *Power-down support, ETMv3.3 and later* on page 3-119 for more information.

Accessing this register, to lock or unlock the ETM trace registers, also resets the internal save/restore counter. See *OS Save and Restore Register (OSSRR), ETMv3.3 and later* on page 3-84 for details of this counter. You must lock the ETM trace registers before you perform an OS save or restore. This prevents any changes to the trace registers during the save or restore process.

The OS Lock Status Register is read-only. To find out whether the ETM trace registers are locked you read the OS Lock Status register, see *OS Lock Status Register (OSLSR), ETMv3.3 and later*.

OS Lock Status Register (OSLSR), ETMv3.3 and later

The OS Lock Status Register (OSLSR) is used to find whether the ETM trace registers are locked. It can also be used to find whether ETM trace register locking is implemented on your ETM macrocell. It is:

- register `0x0C1`, at offset `0x304` in a memory-mapped implementation
- only available in ETMv3.3 or later
- a read-only register.

Figure 3-43 on page 3-83 shows the bit assignments for the OS Lock Status Register (OSLSR):

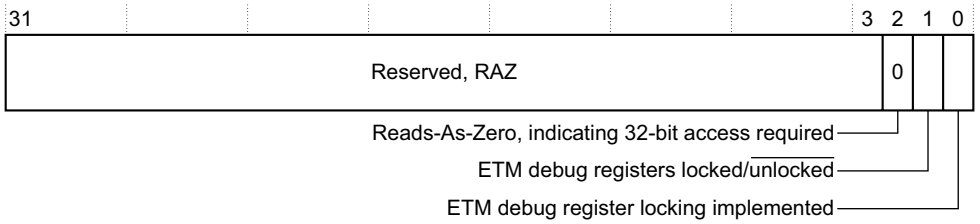


Figure 3-43 OS Lock Status Register (OSLSR) bit assignments

Table 3-54 shows the bit assignments for the OS Lock Status Register (OSLSR):

Table 3-54 OS Lock Status Register (OSLSR) bit assignments

Bit	Version[a]	Description
[31:3]	-	Reserved, Read-As-Zero (RAZ).
[2]	3.3	32-bit access bit. This bit is always Reads-As-Zero, indicating that 32-bit accesses are required to operate the OS Lock Access Register.
[1]	3.3	Locked bit. The possible values of this bit are: **0** ETM trace registers are not locked. **1** ETM trace registers are locked. Any access to these registers returns a slave-generated error response. The reset value of this field corresponds to the value on the **DBGOSLOCKINIT** pin.
[0]	3.3	ETM trace register locking implemented. The possible values of this bit are: **0** OS Lock and OS Save/Restore registers are not implemented. In this case, bits [31:0] of the OS Lock Status register are Read-As-Zero. **1** OS Lock and OS Save/Restore registers are implemented and it is possible to set the OS Lock for this macrocell, to lock the ETM trace registers. The reset value of this field is IMPLEMENTATION DEFINED.

a. The first ETM architecture version that defines the field.

If a read of the OS Lock Status Register returns zero, OS Locking is not implemented.

Note

Because UNDEFINED ETM registers Read-As-Zero, this test can be used on ETM versions earlier than ETMv3.3.

See *Power-down support, ETMv3.3 and later* on page 3-119 for more information about OS Locking.

OS Save and Restore Register (OSSRR), ETMv3.3 and later

The OS Save and Restore Register (OSSRR) is used to save or restore the complete ETM trace register state of the macrocell. It is:

- register `0x0C2`, at offset `0x308` in a memory-mapped implementation
- only available in ETMv3.3 or later
- a read/write register.

Figure 3-44 shows the bit assignments for the OS Save and Restore Register (OSSRR):

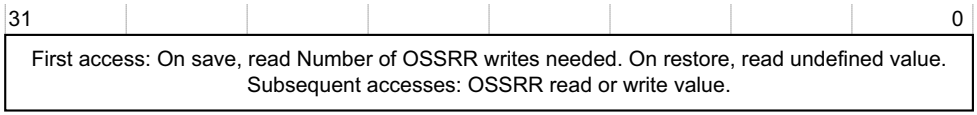


Figure 3-44 OS Save and Restore Register (OSSRR) bit assignments

Table 3-55 shows the bit assignments for the OS Save and Restore Register (OSSRR):

Table 3-55 OS Save and Restore Register (OSSRR) bit assignments

Bit	Version[a]	Description
[31:0]	v3.3	The first access to the register must be a read. On this access: • on an OS save, this field returns the number of additional read accesses required to save the ETM trace register settings • on an OS restore, this field returns an UNKNOWN value, or an IMPLEMENTATION DEFINED value. **Note** On an OS restore, this read must be performed even if it returns an UNKNOWN value. On subsequent accesses the field holds the ETM trace register value being saved or restored.

a. The first ETM architecture version that defines the field.

This register works in conjunction with an internal sequence counter to enable you to save or restore the contents of the ETM trace registers. See Table 3-4 on page 3-16 for details of how the ETM registers are split into trace and management registers.

Before accessing this register, you must write the key value, `0xC5ACCE55`, to the OS Lock Access register. This write resets the OS save/restore internal sequence counter. Locking the ETM trace registers in this way prevents any change to their contents during the save/restore process.

When you have locked access to the ETM trace registers you must read the OSSRR. The significance of the result returned depends on whether you are performing an OS save or an OS restore:

OS save The value returned is the number of additional OSSRR accesses required to save or restore the ETM trace registers. After reading this value, you must:

- save this value, for use for the OS restore
- perform this number of reads of the OSSRR, saving the returned values to save the status of the ETM trace registers.
- to restore the ETM trace registers you must perform this number of writes to the OSSRR.

OS restore The value returned is UNKNOWN, or might be IMPLEMENTATION DEFINED. After reading this value, you must:

- discard this value
- use the number of accesses value saved from the OS save operation to know how many accesses are required to restore the status of the ETM trace registers
- perform this number of writes to the OSSRR, writing back the status information saved in the OS save operation.

The number of accesses required, and the order and interpretation of the save and restore data, is IMPLEMENTATION DEFINED. However, the restore sequence must observe the writable status of all bits of the registers being restored.

Behavior is UNPREDICTABLE if:

- you access the OSSRR when the ETM trace registers are not locked
- after writing `0xC5ACCE55` to the OS Lock Access register, your first access to the OSSRR is a write
- after your first read of the OSSRR, you mix read and write accesses to the OSSRR
- after your first read of the OSSRR, you perform more read or writes to the OSSRR than are required to save or restore the ETM trace registers.

For more information on using the OSSRR, see *Power-down support, ETMv3.3 and later* on page 3-119.

3.5.25 Device Power-Down Status Register (PDSR)

The PDSR is used to indicate the power-down status of the ETM. It is:

- register `0xC5`, at offset `0x314` in a memory-mapped implementation
- only available in ETMv3.3 or later
- a read-only register.

Figure 3-45 on page 3-86 shows the bit assignments for the PDSR.

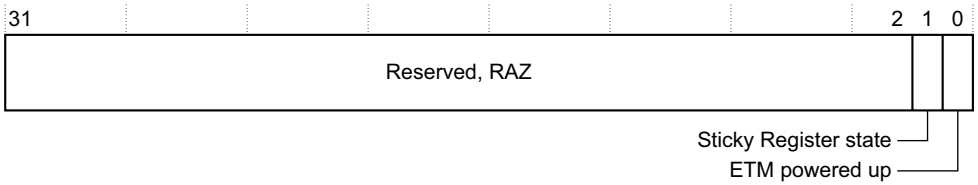


Figure 3-45 PDSR bit assignments

Table 3-56 shows the bit assignments for the PDSR.

Table 3-56 PDSR bit assignments

Bit	Version[a]	Description
[31:2]	-	Reserved, Read-As-Zero (RAZ).
[1]	3.3	Sticky Register state bit. The possible values of this bit are: **0** ETM Trace Registers have not been powered down since this register was last read. **1** ETM Trace Registers have been powered down since this register was last read, and have lost their state. This bit is cleared to 0 on reading this register if the debug power domain of the ETM is currently powered up (bit [0] is HIGH). When the debug power domain of the ETM is powered down, this bit is set to 1. Reads of this register when the debug power domain is powered down return 1 for this bit, and do not change the value of this bit. Reads of this register when the debug power domain is powered up return the current value of this bit, and then clear this bit to 0. If the Software Lock mechanism is locked and the PDSR read is made through the memory mapped interface, this bit is not cleared. When this bit is set, accesses to any ETM Trace Registers return an error response.
[0]	3.3	ETM powered up bit. The value of this bit indicates whether the ETM Trace Registers are accessible. The possible values are: **0** ETM Trace Registers cannot be accessed. **1** ETM Trace Registers can be accessed. Usually, an external input to the ETM, driven by the system power controller, controls the value of this bit. When this bit is set to 0, accesses to any ETM Trace Registers return an error response.

a. The first ETM architecture version that defines the field.

Table 3-57 shows the different encodings of the PDSR.

Table 3-57 PDSR encodings

Bit [1] Sticky Register state	Bit [0] ETM powered up	Meaning
0	0	ETM Trace Registers are inaccessible. No state has been lost.
0	1	ETM Trace Registers are accessible.
1	0	ETM Trace Registers are powered down, inaccessible, and their state has been lost.
1	1	ETM Trace Registers are powered up. However, their state has been lost because of a power down.

This register is always implemented in:

- ETMs that support multiple power domains where the power to the ETM Trace Registers can be removed
- situations where access to the ETM Trace Registers is limited because of other design limitations.

If the ETM only occupies a single power domain, this register might always read as `0x00000001`, indicating that the ETM is powered up and accessible. In this case, if the ETM is not powered up:

- no ETM registers are accessible
- the PDSR does not indicate whether the ETM state has been lost.

3.5.26 Integration Mode Control Register (ITCTRL), ETMv3.2 and later

The Integration Mode Control Register (ITCTRL) is used to enable topology detection or to check integration testing. It is:

- register `0x3C0`, at offset `0xF00` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read/write register.

Figure 3-46 shows the bit assignments for the Integration Mode Control Register:

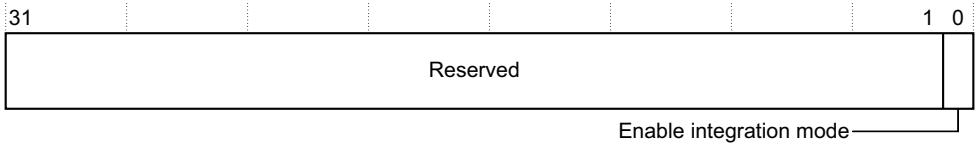


Figure 3-46 Integration Mode Control Register bit assignments

Table 3-58 shows the bit assignments for the Integration Mode Control Register:

Table 3-58 Integration Mode Control Register bit assignments

Bit	Version[a]	Description
[31:1]	-	Reserved.
[0]	v3.2	When this bit is set to 1, the device enters an integration mode to enable Topology Detection or Integration Testing to be checked. On an ETM reset this bit is cleared to 0.

a. The first ETM architecture version that defines the field.

3.5.27 Claim tag registers, ETMv3.2 and later

The two claim tag registers are described in the following sections:

- *Claim Tag Set Register (CLAIMSET)*
- *Claim Tag Clear Register (CLAIMCLR)* on page 3-89.

The claim tag can be used to coordinate software and debugger access to ETM functionality.

Claim Tag Set Register (CLAIMSET)

The Claim Tag Set Register (CLAIMSET) is used to set bits in the claim tag and find the number of bits supported by the claim tag. It is:

- register `0x3E8`, at offset `0xFA0` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read/write register.

Figure 3-47 shows the bit assignments for the Claim Tag Set Register:

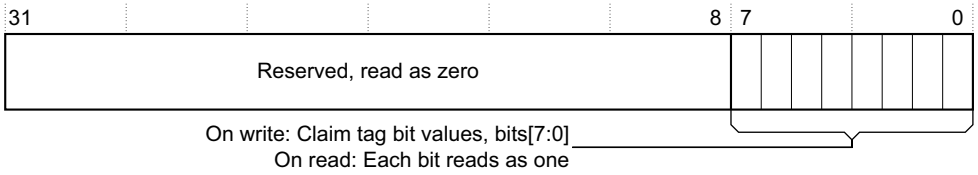


Figure 3-47 Claim Tag Set Register bit assignments

Table 3-59 shows the bit assignments for the Claim Tag Set Register:

Table 3-59 Claim Tag Set Register bit assignments

Bit	Version[a]	Description
[31:8]	-	Reserved.
[7:0]	v3.2	On reads, returns `0xFF`. On writes, a 1 in a bit position causes the corresponding bit in the claim tag value to be set.

a. The first ETM architecture version that defines the field.

Claim Tag Clear Register (CLAIMCLR)

The Claim Tag Clear Register (CLAIMCLR) is used to clear bits in the claim tag to 0, and to find the current value of the claim tag. It is:

- register `0x3E9`, at offset `0xFA4` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read/write register.

Figure 3-48 shows the bit assignments for the Claim Tag Clear Register:

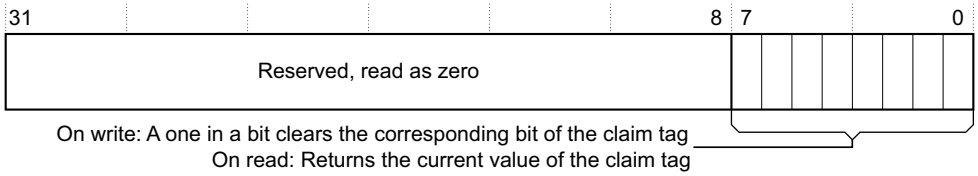


Figure 3-48 Claim Tag Clear Register bit assignments

Table 3-60 shows the bit assignments for the Claim Tag Clear Register:

Table 3-60 Claim Tag Clear Register bit assignments

Bit	Version[a]	Description
[31:8]	-	Reserved.
[7:0]	v3.2	On reads, returns the current claim tag value. On writes, a 1 in a bit position causes the corresponding bit in the claim tag value to be cleared to 0. On an ETM reset this field is cleared to 0x00.

a. The first ETM architecture version that defines the field.

3.5.28 Lock registers, ETMv3.2 and later

From ETM architecture version 3.2, the lock registers control memory-mapped software access to all other registers, including the ETM Control Register. When the ETM is locked using this feature, memory-mapped software writes are ignored. JTAG accesses, coprocessor accesses, memory-mapped debugger accesses, and all reads are unaffected.

Note

- Any implementation of the ETM architecture that implements memory mapped access to ETM registers must implement the lock access mechanism.
- When the lock access mechanism is implemented, an ETM reset locks the ETM.

This feature can be used to prevent accidental modification of the ETM registers by software being debugged. For example, software that accidentally initializes unwanted areas of memory might disable the ETM, making it impossible to trace such software. To prevent this, on-chip software that accesses the ETM must access the ETM registers as follows:

1. Unlock the ETM by writing `0xC5ACCE55` to the Lock Access Register.
2. Access the other ETM registers.
3. Lock the ETM by writing any other value, for example `0x0`, to the Lock Access Register.

Lock Access Register (LAR or LOCKACCESS)

The Lock Access Register (LAR or LOCKACCESS) is used to lock and unlock access to all other ETM registers. It is:

- register `0x3EC`, at offset `0xFB0` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a write-only register.

Figure 3-49 shows the bit assignments for the Lock Access Register:

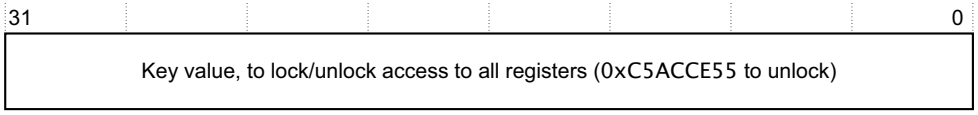


Figure 3-49 Lock Access Register bit assignments

Table 3-61 on page 3-91 shows the bit assignments for the Lock Access Register.

Table 3-61 Lock Access Register bit assignments

Bit	Version[a]	Description
[31:0]	v3.2	A write of `0xC5ACCE55` unlocks the ETM. A write of any other value locks the ETM. Writes to this register from an interface that ignores the lock registers are ignored.

a. The first ETM architecture version that defines the field.

Lock Status Register (LSR or LOCKSTATUS)

The Lock Status Register (LSR or LOCKSTATUS) has two uses:

- If you read this register from any interface, you can check bit [0] to find out whether the lock registers are implemented for the interface you are using.
- If you read this register from an interface for which lock registers *are* implemented, you can check bit [1] to find out whether the registers are currently locked.

The Lock Status Register is:

- register `0x3ED`, at offset `0xFB4` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-50 shows the bit assignments for the Lock Status Register:

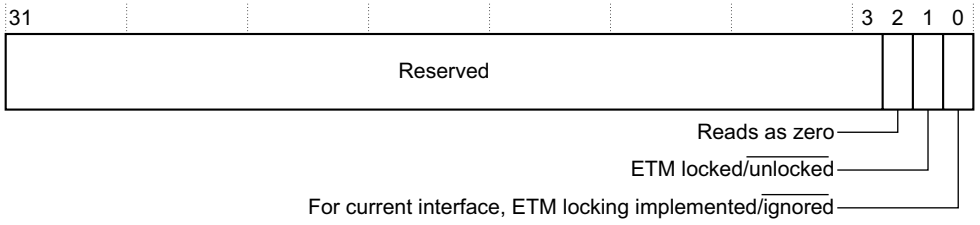


Figure 3-50 Lock Status Register bit assignments

Table 3-62 on page 3-92 shows the bit assignments for the Lock Status Register.

Table 3-62 Lock Status Register bit assignments

Bit	Version[a]	Description
[31:3]	-	Reserved.
[2]	v3.2	Reads as b0. Indicates that the Lock Access Register is 32 bits.
[1]	v3.2	Indicates if the ETM is locked. The possible values of this bit are: **0** Writes are permitted. **1** ETM locked. Writes are ignored. If this register is accessed from an interface where the lock registers are ignored, this field reads as 0 regardless of whether the ETM is locked.
[0]	v3.2	Indicates if the lock registers are implemented for this interface. The possible values of this bit are: **0** This access is from an interface that ignores the lock registers. **1** This access is from an interface that requires the ETM to be unlocked.

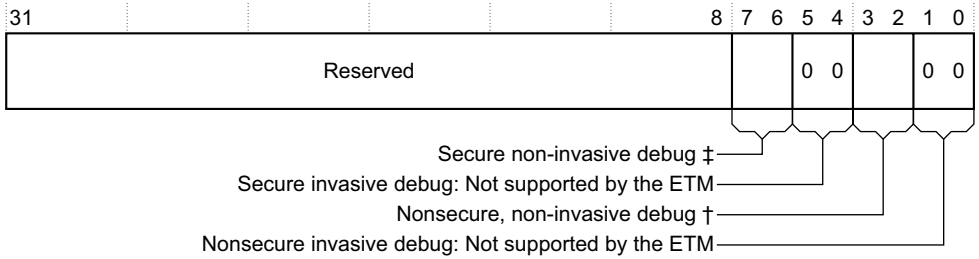
a. The first ETM architecture version that defines the field.

3.5.29 Authentication Status Register (AUTHSTATUS), ETMv3.2 and later

The Authentication Status Register (AUTHSTATUS) reports the level of tracing currently permitted by the authentication signals provided to the ETM. It is:

- register 0x3EE, at offset 0xFB8 in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-51 shows the bit assignments for the Authentication Status Register:



‡ It is Implementation-defined whether an ETM implements this field

† This field is only implemented when the processor supports TrustZone security extensions

Figure 3-51 Authentication Status Register bit assignments

Table 3-63 shows the bit assignments for the Authentication Status Register:

Table 3-63 Authentication Status Register bit assignments

Bit	Version[a]	Description
[31:8]	-	Reserved, RAZ.
[7:6][b]	v3.2	Permission for Secure non-invasive debug. See *Implementation of the Secure non-invasive debug field* for more information.
[5:4]	v3.2	Reads as b00, Secure invasive debug not supported by the ETM.
[3:2]	v3.2	Permission for Non-secure non-invasive debug. This field is only implemented if the processor core implemented with the ETM implements the Security Extensions. When this field is implemented the possible values of the field are: **b10** Non-secure non-invasive debug disabled. **b11** Non-secure non-invasive debug enabled. This field is a logical OR of the **NIDEN** and **DBGEN** signals. It takes the value b11 when the OR is TRUE, and b10 when the OR is FALSE. If the core does not support the Security Extensions, bits [3:2] are reserved, RAZ.
[1:0]	v3.2	Reads as b00, Non-secure invasive debug not supported by the ETM.

a. The first ETM architecture version that defines the field.
b. It is IMPLEMENTATION DEFINED whether an ETM implements the Secure non-invasive debug field. If the field is not implemented then bits [7:6] are Reserved, RAZ.

Implementation of the Secure non-invasive debug field

It is IMPLEMENTATION DEFINED whether an ETM implements the Secure non-invasive debug field. If this field is implemented, its behavior depends on whether the core implemented with the ETM supports the Security Extensions. If the core does support the Security Extensions, then the behavior depends on which of the following applies:

- the processor controls what trace is prohibited
- the ETM controls what trace is prohibited.

This behavior is shown in Table 3-64 on page 3-94.

Table 3-64 Implementation of the Secure non-invasive debug field

Core includes Security Extensions?	Control of Secure tracing	Behavior of Secure non-invasive debug field, bits [7:6]
No	-[a]	The processor is assumed to operate in a Secure state. The possible values of the field are: **b10** Secure non-invasive debug disabled, **b11** Secure non-invasive debug enabled. The value of this field is a logical OR of the **NIDEN** and **DBGEN** signals. It takes the value b11 when the OR is TRUE, and b10 when the OR is FALSE.
Yes	Not controlled by ETM	The field reads as b00, indicating that the ETM does not control when trace is prohibited.
Yes	Controlled by ETM	The possible values of the field are: **b10** Secure non-invasive debug disabled, **b11** Secure non-invasive debug enabled. The value of this field is a logical result of: (**SPNIDEN** OR **SPIDEN**) AND (**NIDEN** OR **DBGEN**) It takes the value b11 when the logical result is TRUE, and b10 when it is FALSE. Figure 3-52 shows the logic used to obtain the value of this field.

a. Not applicable when the core does not support the Security Extensions.

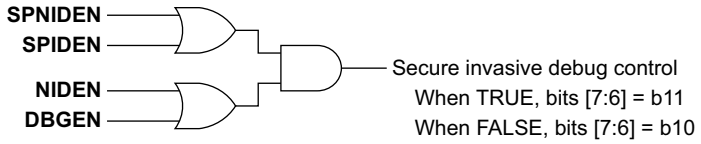


Figure 3-52 Secure non-invasive debug enable logic when controlled by the ETM

3.5.30 Device Configuration Register (DEVID), ETMv3.2 and later

The Device Configuration Register (DEVID) returns an IMPLEMENTATION DEFINED CoreSight component capabilities field. It is:

- register `0x3F2`, at offset `0xFC8` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register
- present in all CoreSight components
- IMPLEMENTATION DEFINED for each designer and part number.

The data width of the Device Configuration Register is IMPLEMENTATION DEFINED.

Figure 3-53 shows the bit assignments for the Device Configuration Register:

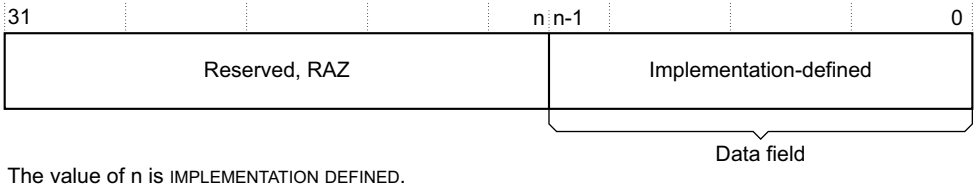


Figure 3-53 Device Configuration Register bit assignments

Table 3-65 shows the bit assignments for the Device Type Register:

Table 3-65 Device Configuration Register bit assignments

Bit	Version[a]	Description
[31:n][b]	-	Reserved. Read-as-zero.
[n-1:0]	v3.2 and later	Component capabilities. Bit assignments in this field are IMPLEMENTATION DEFINED.

a. The first ETM architecture version that defines the field.
b. The value of n is IMPLEMENTATION DEFINED.

The Device Configuration Register is a required register in any CoreSight component. The data field in this register is used to indicate the capabilities of the component. The width of the data field, and the meaning of the bits in the data field, are IMPLEMENTATION DEFINED. All unused bits must Read-As-Zero.

If a component is configurable, ARM Limited recommends that this register is used to indicate any changes to the standard configuration.

3.5.31 Device Type Register (DEVTYPE), ETMv3.2 and later

The Device Type Register (DEVTYPE) returns the CoreSight device type of the component. It is:

- register `0x3F3`, at offset `0xFCC` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register
- present in all CoreSight components.

Figure 3-54 shows the bit assignments for the Device Type Register:

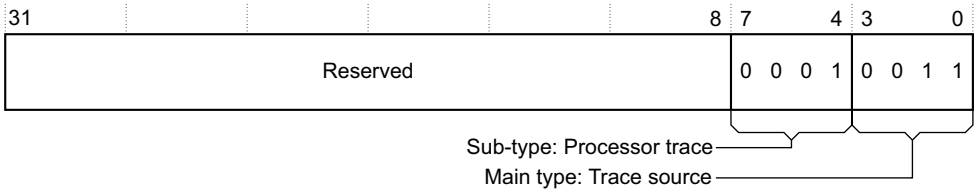


Figure 3-54 Device Type Register bit assignments

Table 3-66 shows the bit assignments for the Device Type Register:

Table 3-66 Device Type Register bit assignments

Bit	Defined in ETM architecture versions	Description
[31:8]	-	Reserved
[7:4]	v3.2 and later	`0x1` Sub type, processor trace
[3:0]	v3.2 and later	`0x3` Main type, trace source

3.5.32 Peripheral identification registers, ETMv3.2 and later

The peripheral identification registers provide standard information required by all CoreSight components. They are a set of eight registers, listed in register number order in Table 3-67:

Table 3-67 Summary of the peripheral identification registers

Description	Register number	Offset[a]
Peripheral ID4	0x3F4	0xFD0
Peripheral ID5	0x3F5	0xFD4
Peripheral ID6	0x3F6	0xFD8
Peripheral ID7	0x3F7	0xFDC
Peripheral ID0	0x3F8	0xFE0
Peripheral ID1	0x3F9	0xFE4
Peripheral ID2	0x3FA	0xFE8
Peripheral ID3	0x3FB	0xFEC

a. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4 x (Register number).

Only bits [7:0] of each Peripheral ID Register are used, with bits [31:8] reserved. Together, the eight Peripheral ID Registers can be considered to define a single 64-bit Peripheral ID, as shown in Figure 3-55.

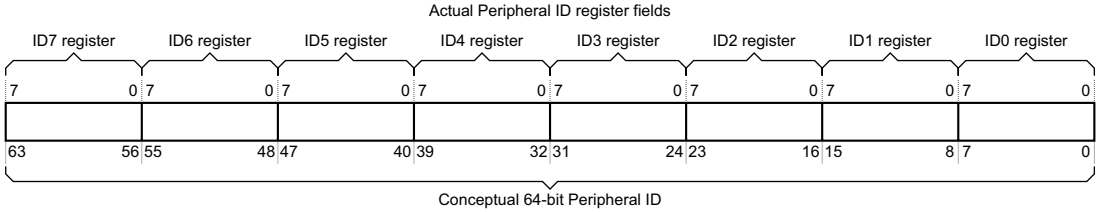


Figure 3-55 Mapping between the Peripheral ID Registers and the Peripheral ID value

Figure 3-56 on page 3-98 shows the standard Peripheral ID fields in the single conceptual Peripheral ID.

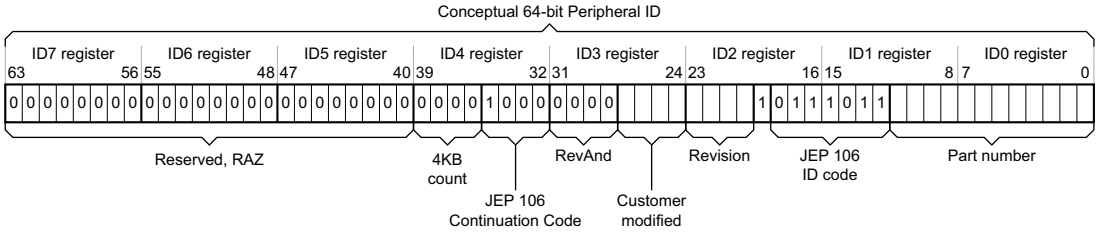


Figure 3-56 Peripheral ID fields

Table 3-68 shows the standard Peripheral ID fields, and shows where this information is held in the Peripheral ID Registers.

Table 3-68 Register fields for the peripheral identification registers

Name	Size	Description	See Register
4KB Count	4 bits	Log_2 of the number of 4KB blocks occupied by the device. ETM implementations occupy a single 4KB block, so this field is always `0x0`.	Peripheral ID4
JEP 106 code	4+7 bits	Identifies the designer of the device. This consists of a 4-bit continuation code and a 7-bit identity code. In all current ETMs the continuation code is `0x4` and the identity code is `0x3B`, indicating ARM.	Peripheral ID1, Peripheral ID2, Peripheral ID4
Part Number	12 bits	Part number for the device.	Peripheral ID0, Peripheral ID1
Revision	4 bits	Revision of the peripheral. Identical to bits [3:0] of the ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71.	Peripheral ID2
RevAnd	4 bits	Indicates a late modification to the device, usually as a result of an Engineering Change Order. This field is `0x0` in all current implementations.	Peripheral ID3
Customer modified	4 bits	Indicates an endorsed modification to the device.	Peripheral ID3

For more information about these fields, see the *CoreSight Architecture Specification*.

The fields present in each register are indicated in the following sections. Registers are described in register name order (ID0 to ID7). Table 3-67 on page 3-97 shows the register numbers and offset addresses of these registers, that do not run in register name order.

Peripheral ID0 Register

The Peripheral ID0 Register holds peripheral identification information. It is:

- register `0x3F8`, at offset `0xFE0` in a memory-mapped implementation

- only available in ETMv3.2 or later
- a read-only register.

Figure 3-57 shows the bit assignments for the Peripheral ID0 Register:

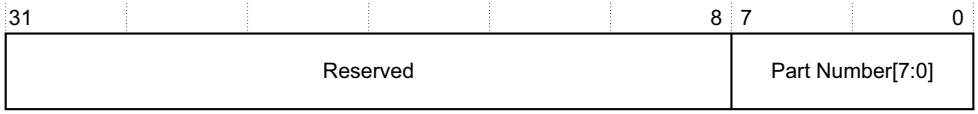


Figure 3-57 Peripheral ID0 Register bit assignments

Table 3-69 shows the bit assignments for the Peripheral ID0 Register:

Table 3-69 Peripheral ID0 Register bit assignments

Bits	Defined in ETM architecture versions	Description[a]
[31:8]	-	Reserved
[7:0]	v3.2 and later	Part Number[7:0]

a. See Table 3-68 on page 3-98 for more information about the register fields.

Peripheral ID1 Register

The Peripheral ID1 Register holds peripheral identification information. It is:

- register `0x3F9`, at offset `0xFE4` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-58 shows the bit assignments for the Peripheral ID1 Register:

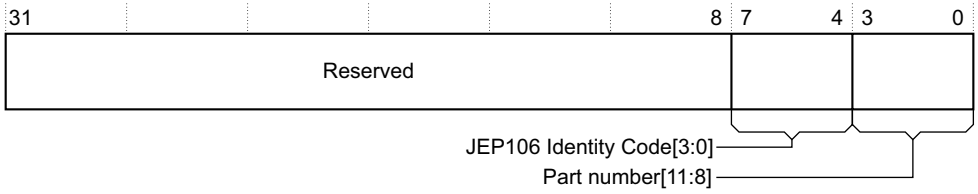


Figure 3-58 Peripheral ID1 Register bit assignments

Table 3-70 shows the bit assignments for the Peripheral ID1 Register:

Table 3-70 Peripheral ID1 Register bit assignments

Bits	Defined in ETM architecture versions	Description[a]
[31:8]	-	Reserved
[7:4]	v3.2 and later	JEP106 Identity Code[3:0]
[3:0]	v3.2 and later	Part Number[11:8]

a. See Table 3-68 on page 3-98 for more information about the register fields.

Peripheral ID2 Register

The Peripheral ID2 Register holds peripheral identification information. It is:

- register `0x3FA`, at offset `0xFE8` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-59 shows the bit assignments for the Peripheral ID2 Register:

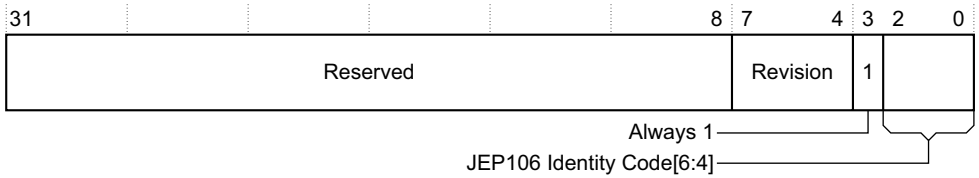


Figure 3-59 Peripheral ID2 Register bit assignments

Table 3-71 shows the bit assignments for the Peripheral ID2 Register:

Table 3-71 Peripheral ID2 Register bit assignments

Bits	Defined in ETM architecture versions	Description[a]
[31:8]	-	Reserved
[7:4]	v3.2 and later	Revision
[3]	v3.2 and later	Always 1
[2:0]	v3.2 and later	JEP106 Identity Code[6:4]

a. See Table 3-68 on page 3-98 for more information about the register fields.

Peripheral ID3 Register

The Peripheral ID3 Register holds peripheral identification information. It is:

- register `0x3FB`, at offset `0xFEC` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-60 shows the bit assignments for the Peripheral ID3 Register:

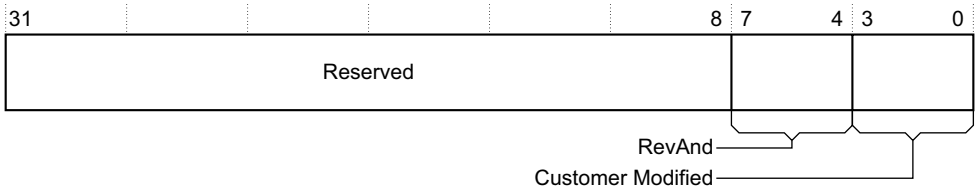


Figure 3-60 Peripheral ID3 Register bit assignments

Table 3-72 shows the bit assignments for the Peripheral ID3 Register:

Table 3-72 Peripheral ID3 Register bit assignments

Bits	Defined in ETM architecture versions	Description[a]
[31:8]	-	Reserved
[7:4]	v3.2 and later	RevAnd
[3:0]	v3.2 and later	Customer Modified

a. See Table 3-68 on page 3-98 for more information about the register fields.

Peripheral ID4 Register

The Peripheral ID4 Register holds peripheral identification information. It is:

- register `0x3F4`, at offset `0xFD0` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-61 shows the bit assignments for the Peripheral ID4 Register:

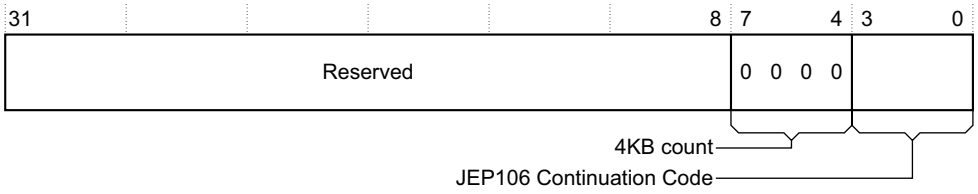


Figure 3-61 Peripheral ID4 Register bit assignments

Table 3-73 shows the bit assignments for the Peripheral ID4 Register:

Table 3-73 Peripheral ID4 Register bit assignments

Bits	Defined in ETM architecture versions	Description[a]
[31:8]	-	Reserved
[7:4]	v3.2 and later	4KB count
[3:0]	v3.2 and later	JEP106 Continuation Code

a. See Table 3-68 on page 3-98 for more information about the register fields.

Peripheral ID5 to Peripheral ID7 Registers

In all current implementations, no information is held in the Peripheral ID5, Peripheral ID6 and Peripheral ID7 Registers. Figure 3-62 shows the bit assignments for these registers:

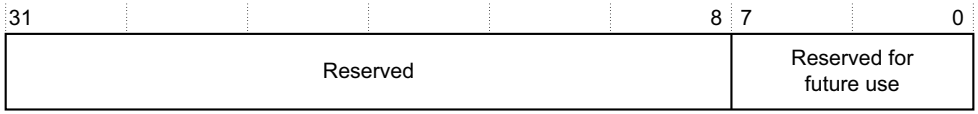


Figure 3-62 Peripheral ID5 to Peripheral ID7 Registers, bit assignments

Table 3-74 shows the bit assignments for the Peripheral ID5, Peripheral ID6 and Peripheral ID7 Registers:

Table 3-74 Peripheral ID5 to Peripheral ID7 Registers, bit assignments

Bits	Defined in ETM architecture versions	Description
[31:8]	-	Reserved
[7:0]	v3.2 and later	Reserved

The register addresses and memory offsets for these registers are shown in Table 3-67 on page 3-97.

3.5.33 Component identification registers, ETMv3.2 and later

There are four read-only Component Identification Registers, ComponentID3 to ComponentID0. Table 3-75 shows these registers:

Table 3-75 Summary of the component identification registers

Register	Register number	Offset[a]
Component ID0	0x3FC	0xFF0
Component ID1	0x3FD	0xFF4
Component ID2	0x3FE	0xFF8
Component ID3	0x3FF	0xFFC

a. Register offset where the registers are accessed in a memory-mapped scheme. The register offset is always 4x (Register number).

The component identification registers identify the ETM as a CoreSight component. For more information, see the *CoreSight Architecture Specification*.

Only bits [7:0] of each register are used. The concept of a single 32-bit component ID, obtained from the four Component Identification Registers, is shown in Figure 3-63:

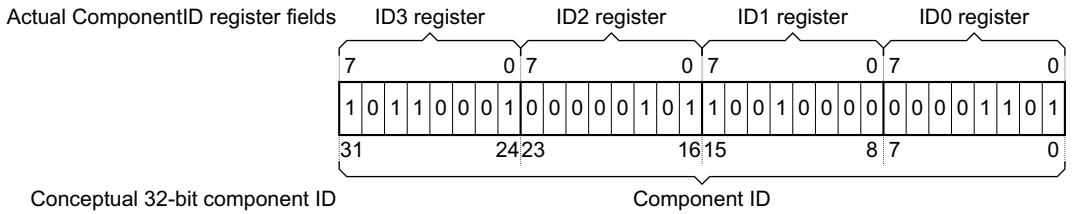


Figure 3-63 Mapping between the Component ID Registers and the Component ID value

Component ID0 Register

The Component ID0 Register holds byte 0 of the CoreSight preamble information. It is:

- register 0x3FC, at offset 0xFF0 in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-64 on page 3-105 shows the bit assignments for the Component ID0 Register:

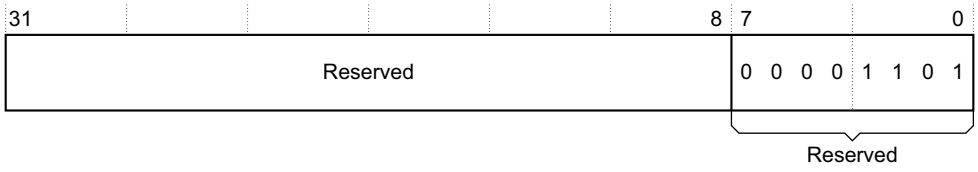


Figure 3-64 Component ID0 Register bit assignments

Table 3-76 shows the bit assignments for the Component ID0 Register:

Table 3-76 Component ID0 Register bit assignments

Bits	Defined in ETM architecture versions	Value	Description
[31:8]	-	-	Reserved
[7:0]	v3.2 and later	0x0D	Reserved

Component ID1 Register

The Component ID1 Register holds byte 1 of the CoreSight preamble information. It is:

- register 0x3FD, at offset 0xFF4 in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-65 shows the bit assignments for the Component ID1 Register:

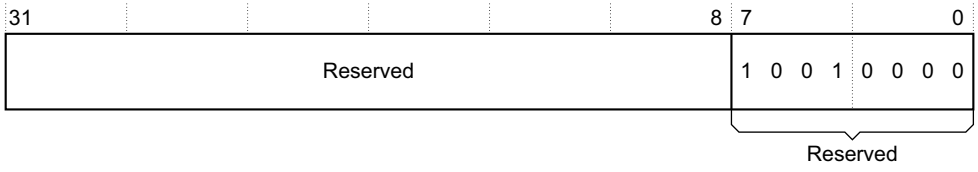


Figure 3-65 Component ID1 Register bit assignments

Table 3-77 shows the bit assignments for the Component ID1 Register:

Table 3-77 Component ID1 Register bit assignments

Bits	Defined in ETM architecture versions	Value	Description
[7:0]	v3.2 and later	0x90	Reserved

Component ID2 Register

The Component ID2 Register holds byte 2 of the CoreSight preamble information. It is:

- register `0x3FE`, at offset `0xFF8` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-66 shows the bit assignments for the Component ID2 Register:

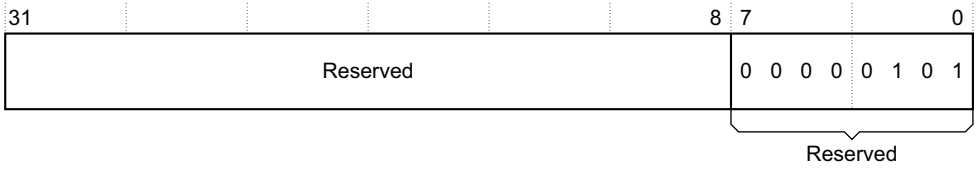


Figure 3-66 Component ID2 Register bit assignments

Table 3-78 shows the bit assignments for the Component ID2 Register:

Table 3-78 Component ID2 Register bit assignments

Bits	Defined in ETM architecture versions	Value	Description
[31:8]	-	-	Reserved
[7:0]	v3.2 and later	`0x05`	Reserved

Component ID3 Register

The Component ID3 Register holds byte 3 of the CoreSight preamble information. It is:

- register `0x3FF`, at offset `0xFFC` in a memory-mapped implementation
- only available in ETMv3.2 or later
- a read-only register.

Figure 3-67 shows the bit assignments for the Component ID3 Register:

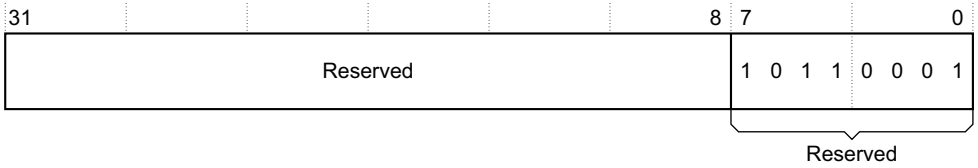


Figure 3-67 Component ID3 Register bit assignments

Table 3-79 shows the bit assignments for the Component ID3 Register:

Table 3-79 Component ID3 Register bit assignments

Bits	Defined in ETM architecture versions	Value	Description
[31:8]	-	-	Reserved
[7:0]	v3.2 and later	0xB1	Reserved

3.6 Using ETM event resources

This section explains how to use ETM event resources. It describes:

- *Resource identification*
- *Boolean combinations for defining events* on page 3-110
- *Examples of event and resource programming* on page 3-112.

3.6.1 Resource identification

A resource identifier is seven bits:

- three bits for the resource type
- four bits for the index.

Resource encoding

The resource encoding is given in Table 3-80.

Table 3-80 Resource encodings

Bit	Defined in ETM architecture versions	Description
[6:4]	v1.0	Resource type
[3:0]	v1.0	Resource index

Table 3-81 defines the available resource types and shows the bit encodings used to identify them.

Table 3-81 Resource identification encoding

Resource type (bits [6:4])	Index range (bits [3:0])	Description of resource type
b000	0-15	Single address comparator 1-16.
b001	0-7	Address range comparator 1-8. Represents the range between two single address comparators.
b001	8-11	Instrumentation resource 1-4. Software-controlled resources, see *Instrumentation resources, from ETMv3.3* on page 2-63. Only available in ETMv3.3 and later.

Table 3-81 Resource identification encoding (continued)

Resource type (bits [6:4])	Index range (bits [3:0])	Description of resource type
b010	0-7	EmbeddedICE module watchpoint comparators 1-8. It is IMPLEMENTATION DEFINED whether an ETM supports EmbeddedICE watchpoint comparators. If it does: • In ETMv3.3 and earlier, two watchpoint comparators are implemented, using index values 0 and 1. • In ETMv3.4 and later, the number of watchpoint comparators is specified in the Configuration Code Extension Register. The maximum number is eight, and the comparators use index values from 0 up to a maximum of 7.
b011	0-15	Memory map decodes 1-16.
b100	0-3	Counter 1-4 at zero.
b101	0-2	Sequencer in states 1-3.
	3-7	Reserved.
	8-10	Context ID comparator 1-3, ETMv2.0 and later.
	11-14	Reserved.
	15	Trace start/stop resource, ETMv2.0 and later.
b110	0-3	External inputs 1-4.
	4-7	Reserved.
	8-11	Extended external input selectors 1-4, ETMv3.1 and later.
	12	Reserved.
	13	Core is in Non-secure state.
	14	Trace prohibited by core.
	15	Hard-wired input (always true).
b111	-	Reserved.

Note

- Valid range encodings between 0 and 15 refer to resource IDs 1-16.
- When a particular resource is active, this means that its output is a logical 1.

3.6.2 Boolean combinations for defining events

ETM event logic on page 2-15 introduced the ETM concept of an *event* as a logical combination of two event resources, used to control the basic transitions in the ETM. This section summarizes where you require event descriptions to program the ETM registers, and then describes how you define an ETM event.

Where are events used?

The following sections use event definitions, as described in *Defining events*:

- *Trigger Event Register* on page 3-31
- *TraceEnable Event Register* on page 3-41
- *ViewData Event Register* on page 3-45
- *Counter Enable Registers* on page 3-59
- *Counter Reload Event Registers* on page 3-61
- *Sequencer State Transition Event Registers* on page 3-63
- *External Output Event Registers* on page 3-65.

Defining events

If A is defined as the first resource match and B as the second match, an event is defined as a function of A and B. The functions and their bit encodings are listed in Table 3-82.

Table 3-82 Boolean function encoding for events

Encoding	Function
b000	A
b001	NOT(A)
b010	A AND B
b011	NOT(A) AND B
b100	NOT(A) AND NOT(B)
b101	A OR B
b110	NOT(A) OR B
b111	NOT(A) OR NOT(B)

A and B are identified with two 7-bit fields. See *Resource identification* on page 3-108 for the exact resource encoding.

An event is encoded in three fields using 17 bits in total, as shown in Table 3-83. Two fields encode the two event resources (see Table 3-81 on page 3-108 and Table 3-80 on page 3-108) and the third field specifies the Boolean operation to be applied to them (see Table 3-82 on page 3-110).

Table 3-83 Event encoding

Bit	Description
[16:14]	Boolean function
[13:7]	Resource B
[6:0]	Resource A

Event and resource encoding is shown in Figure 3-68.

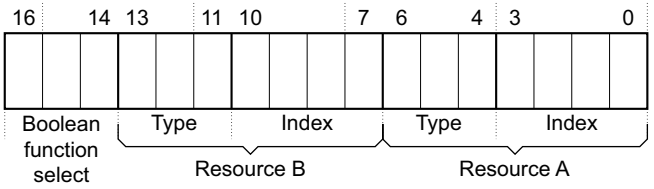


Figure 3-68 Event and resource encoding

Note

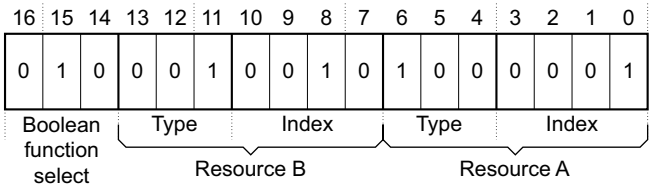
To permanently enable or disable an event, you must specify:

- Resource A as the hard-wired input (type b110, index 15)
- the boolean function as either A (enable) or Not (A) (disable).

3.6.3 Examples of event and resource programming

Example 3-1 shows how to encode an event to occur when in address range 3 and when counter 2 reaches zero.

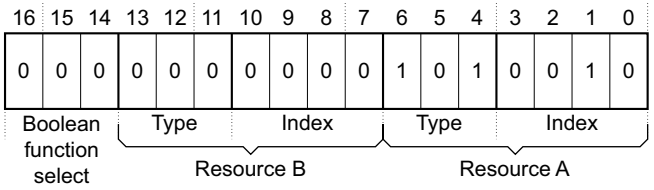
Example 3-1 Encoding an event based on a combination of resources



- bits [16:14] select the Boolean A AND B function, b010
- Resource B is defined as an address range comparator, b001, for range 3, b0010
- Resource A is counter, b100, number 2, b0001.

Example 3-2 shows how to encode an event to occur when sequencer state 3 is reached.

Example 3-2 Encoding an event based on a single resource



- bits [16:14] select the Boolean A function, b000
- Resource A is defined as sequencer, b101, state 3, b0010.

The event is active when the Boolean expression is TRUE.

Because the selected Boolean function does not use Resource B, the value of the Resource B register field is ignored.

3.7 Example ViewData and TraceEnable configurations

The registers used to program **ViewData**, **TraceEnable**, and **FIFOFULL** operate like bit masks that enable individual resources to be activated. Each bit determines whether the function is sensitive to the applicable resource. This section contains two configuration examples:

- *An example ViewData configuration*.
- *An example TraceEnable configuration* on page 3-115.

3.7.1 An example ViewData configuration

Suppose that you want to configure **ViewData** to be asserted only when the following conditions apply:

- the sequencer is in state 3
- the address is in address range 1.

Suppose also that you want to ensure that **ViewData** is not asserted when the address is equal to the values set in address comparators 3 or 4.

The simplified diagram of the **ViewData** configuration required is shown in Figure 3-69.

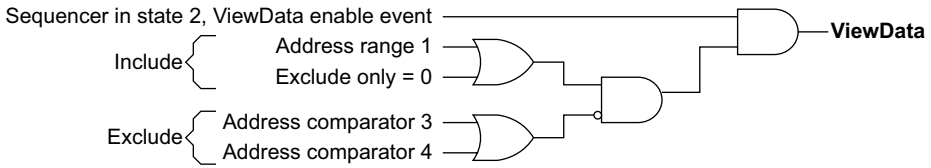


Figure 3-69 Example ViewData configuration

To configure this, proceed as follows:

1. Program the ViewData Event Register as shown in Figure 3-70.

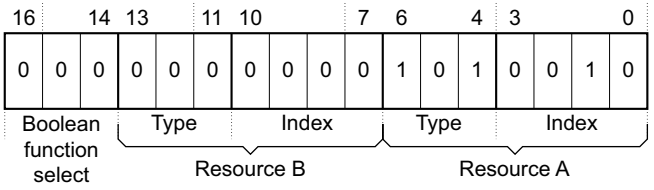


Figure 3-70 ViewData Event Register example

   The ViewData Event Register encodes the **ViewData** enable event to be active when the sequencer (bits [6:4] = b101) is in state 3.

2. Program the ViewData Control 1 Register as shown in Figure 3-71 on page 3-114.

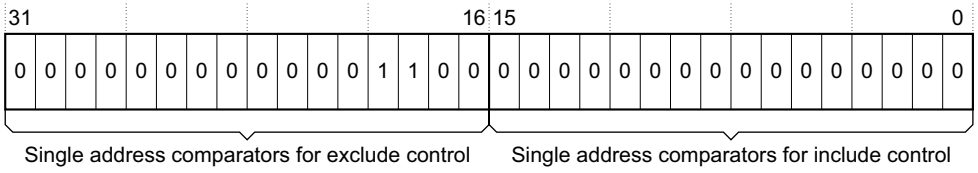


Figure 3-71 ViewData Control 1 Register example

The ViewData Control 1 Register encodes the address comparators that are included and excluded. In this case the encoding shows no include resources. Address comparators 3 and 4 (bits [18:19]) are excluded.

3. Program the ViewData Control 3 register as shown in Figure 3-72.

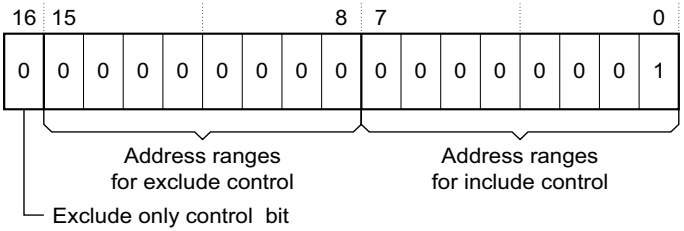


Figure 3-72 ViewData Control 3 Register example

The ViewData Control 3 Register encodes the address ranges that are included and excluded, along with the exclude only control. In this case the exclude only bit, bit [16], must be cleared to 0. Bit [0] is set to 1 to show that address range 1 is included.

The ViewData Control 2 Register encodes the memory map decodes that are included and excluded. There are no MMDs in this example, so you must program this register to zero.

Assume that the following settings have been made for the appropriate resources:

- address range 1 set to `0x8000-0x8100`
- address comparator 3 set to `0x8074`
- address comparator 4 set to `0x8090`.

The result is that the **ViewData** event is activated over addresses `0x8000-0x8100`, but not `0x8074` or `0x8090`, whenever sequencer state 2 is active. This is shown in Figure 3-73 on page 3-115.

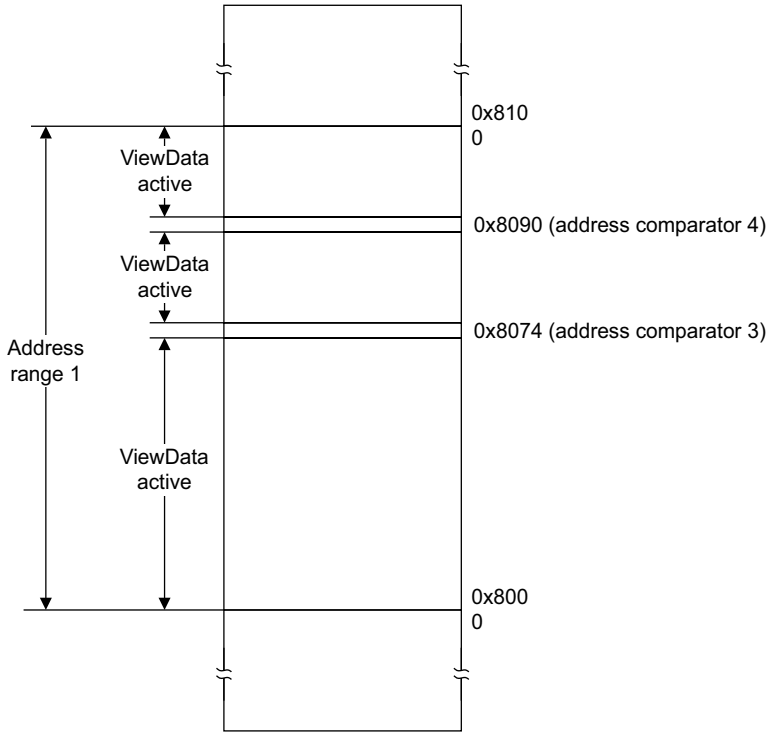


Figure 3-73 Example ViewData composite range

3.7.2 An example TraceEnable configuration

This example is applicable to ETMv1.2 or later.

Suppose that you want to configure the activation of **TraceEnable** to turn tracing on when function X is called, and off when it ends. At the same time, you want to ensure that calls to the C library, and code at a fixed address range, for example, calls to a subfunction, must not be traced.

The simplified diagram of the **TraceEnable** configuration required is shown in Figure 3-74 on page 3-116.

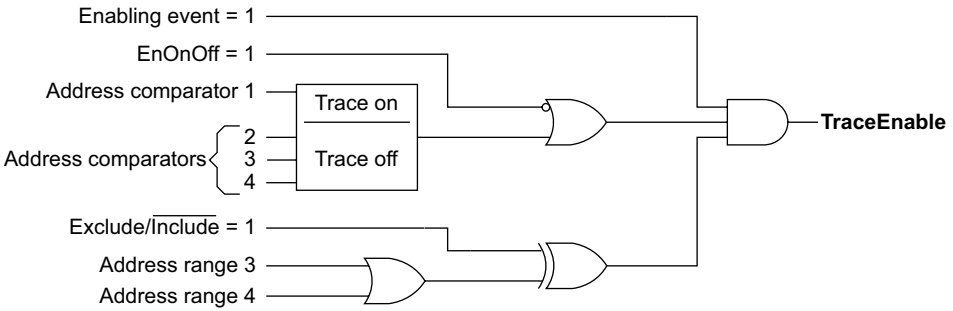


Figure 3-74 Example TraceEnable configuration

The contents of the range and comparison inputs required are listed in Table 3-84.

Table 3-84 Example comparator inputs

Comparator	Contents
Address comparator 1	Function X entry point
Address comparator 2	First function X exit point
Address comparator 3	Second function X exit point
Address comparator 4	Third function X exit point
Address range 3	Sub-function address range to exclude
Address range 4	C library code address range to exclude

To configure **TraceEnable** for this example, proceed as follows:

1. Program the TraceEnable Event Register as shown in Figure 3-75.

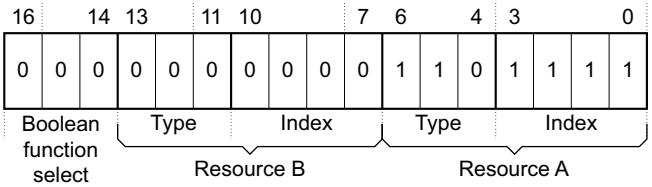


Figure 3-75 TraceEnable Event Register example

The encoding in Figure 3-75 selects Boolean function A, together with external input 16. This permanently enables the **TraceEnable** event.

2. Program the TraceEnable Control 1 Register as shown in Figure 3-76 on page 3-117.

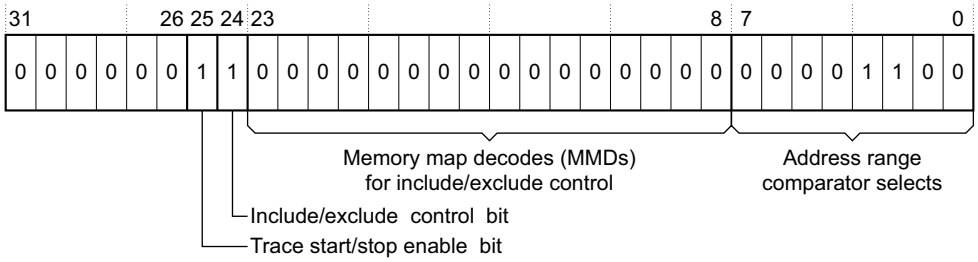


Figure 3-76 TraceEnable Control 1 Register example

Table 3-85 describes the values for this setting of the TraceEnable Control 1 Register.

Table 3-85 TraceEnable Control 1 Register example values

Bit	For this example:	
	Value	Description
[25]	1	Trace start/stop enable. Tracing is controlled by trace on and off addresses.
[24]	1	Include/exclude control. Exclude. The resources specified in bits [23:0] and in the TraceEnable Control 2 Register indicate regions to be excluded from the trace. When outside an exclude region, tracing can occur.
[23:8]	0x0000	Selects no memory map decodes for include/exclude control.
[7:4]	b0000	These address range comparators are not selected.
[3:2]	b11	Address range comparators 3 and 4 are selected.
[1:0]	b00	These address range comparators are not selected.

Bit [25], the tracing on/off bit, of the TraceEnable Control 1 Register is set on, to show that on/off addresses have been specified. Bit [24], the include/exclude bit, is set to exclude, to show that the referenced resources are excluded. Bits [23:8] show that no memory map decodes are referenced. Address range comparators 3 and 4 are indicated by bit [2] and bit [3].

3. Program the Trace Start/Stop Resource Control Register as shown in Figure 3-77:

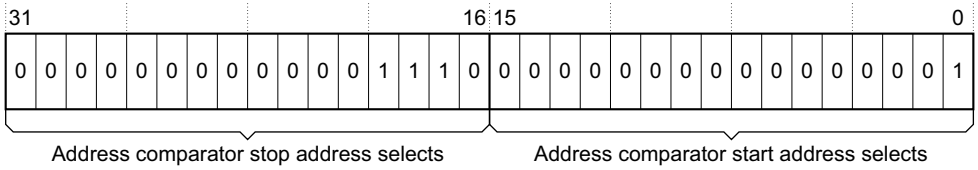


Figure 3-77 Trace Start/Stop Resource Control Register example

The Trace Start/Stop Resource Control Register selects address comparator 1 to turn trace on, and address comparators 2, 3, and 4 to turn trace off.

No single addresses are to be excluded in this example, so you must program the TraceEnable Control 2 Register to zero.

3.8 Power-down support, ETMv3.3 and later

ETMv3.3 introduces power-down support for the macrocell. This support enables the entire ETM state of the macrocell to be saved before it is powered down, and restored when it is powered up again.

Note

The ETM state is held in the ETM trace registers.

The main features of the ETMv3.3 power-down support are:

- The ETM registers are split between ETM trace registers and ETM management registers. It is the ETM trace registers that you can save and restore, to provide power-down support. This classification of the ETM registers is described in *ETM Trace and ETM Management registers, from ETMv3.3* on page 3-16, and is repeated in this section.
- You can lock the ETM trace registers, to prevent any change to their contents.
  - The registers are locked or unlocked by writing to the OS Lock Access register, see *OS Lock Access Register (OSLAR), ETMv3.3 and later* on page 3-81.
  - You can find the current lock status of the ETM trace registers by reading the ETM OS Lock Status register, see *OS Lock Status Register (OSLSR), ETMv3.3 and later* on page 3-82.

  More information about the use of these registers is given in this section.
- You can save and restore the ETM trace registers, using the OS Save and Restore Register. See *OS Save and Restore Register (OSSRR), ETMv3.3 and later* on page 3-84. The registers are saved as, or restored using, a continuous block of data. You do not have to save or restore the registers individually. The registers can only be saved or restored when they have been locked by writing the key value to the OS Lock Access Register.
- The ETM implementation includes the device Power-Down Status Register (PDSR), that indicates when the ETM Trace Registers are powered up. See *Device Power-Down Status Register (PDSR)* on page 3-85. This register also indicates whether the state of the ETM Trace Registers has been lost because of a power-down, and prevents access to the ETM Trace Registers until after the debugger has read the PDSR.

It is IMPLEMENTATION DEFINED whether this power-down support is included in an ETM implementation. However, you can always read the OS Lock Status Register to find whether this feature is implemented.

When power down support is implemented, a JTAG interface to the ETM registers is not permitted. Access to the ETM registers from an external debugger must use the ARM Debug Interface v5. For more information see the *ARM Debug Interface v5 Architecture Specification*.

In a system that supports multiple power domains, the ETM might be split into two domains:

- One domain, typically called the Core domain, is where all the ETM Trace Registers are located. Typically, this domain is where most of the ETM resources and trace generation are found, because normally these must run at the same clock speed as the processor.

- The other domain, typically called the Debug domain, is where the programming interface, ETM Management Registers, and trace output logic are located.

This power domain split enables the processor and Core domain of the ETM to be dynamically powered down while permitting a debugger to maintain communication with the ETM, and to determine that part of the ETM that is powered down.

Typically, the ETM Core domain is the same power domain as the processor. However, some implementations might separate the ETM Core domain from the processor power domain to enable the ETM Core domain to be powered down when the ETM is not in use.

In a typical CoreSight system, the ETM Debug domain is the same power domain as the other debug and trace components. This permits an implementation to power down all the debug and trace logic when not in use.

3.8.1 The process of saving and restoring the macrocell ETM state

This section describes the sequence of operations required to save the macrocell ETM state before a power-down, and to restore the state after the macrocell has been re-powered.

Note

The programming examples given in Example 3-3 on page 3-121 and Example 3-5 on page 3-123 assume an ordered memory model. For more information see *Synchronization of ETM register updates* on page 3-8.

1. You must unlock the CoreSight Lock, if implemented, before performing the register save sequence. Read the *Lock Status Register* (LSR) to determine:
   - if the Lock is present
   - the current status of the lock, if it is present.

   If the Lock is currently locked, then you must unlock it by writing `0xC5ACCE55` to the *Lock Access Register* (LAR).
2. Read the *Power-Down Status Register* (PDSR) to clear the Sticky Register State bit to 0.
3. Before you can save the ETM state you must lock the ETM trace registers. You do this by writing the key, `0xC5ACCE55`, to the OS Lock Access Register. See *OS Lock Access Register (OSLAR), ETMv3.3 and later* on page 3-81.
4. Next, you must read the OS Save and Restore register, see *OS Save and Restore Register (OSSRR), ETMv3.3 and later* on page 3-84. The value returned by this read is the number of additional reads of the OSSRR that are required to save or restore the ETM state. You must save this value for use in the OS restore process.
5. Now you must read the OSSRR the number of times indicated at stage 4, and save the returned values.

   Example 3-3 on page 3-121 shows performing stages 3 to 5 in a memory-mapped implementation of the ETM trace registers.

Example 3-3 OS ETM state save sequence, memory-mapped registers implementation

```
; On entry:
   ;   R0 points to a block to save the ETM registers in.
   ;   CoreSight lock has already been unlocked, if necessary

   SaveETMregisters
       PUSH    {R4, LR}
       MOV     R4, R0                          ; Save pointer

       ; (1) Set OS Lock Access Register (OSLAR).
       BL      GetETMRegisterBase              ; Returns base in R0
       LDR     R1, =0xC5ACCE55
       STR     R1, [R0, #0x300]                ; Write OSLAR

       ; (2) Get the number of words to save ...
       LDR     R1, [R0, #0x308]                ; OSSRR returns size on first read
       ; (3) ... and save this value for the restore
       STR     R1, [R4], #4                    ; Push save count on to the save stack

       ; (4) Loop reading words from the OSSRR.
       CMP     R1, #0                          ; Check for zero
   SaveETMregisters_Loop
       ITTT    NE
       LDRNE   R2, [R0, #0x308]                ; Load a word of data
       STRNE   R2, [R4], #4                    ; Push on to the save stack
       SUBSNE  R1, R1, #1
       BNE     SaveETMregisters_Loop

       ; (5) Return the pointer to first word not written to.
       ;     Leave the OSLAR set as from now on we do not want any changes.
       MOV     R0, R4
       POP     {R4, PC}
```

Example 3-4 shows the same operations in an implementation that provides coprocessor access to the ETM trace registers.

Example 3-4 OS ETM state save sequence, with coprocessor access to ETM registers

```
; On entry:
   ;   R0 points to a block to save the ETM registers in.
   ;   CoreSight lock has already been unlocked, if necessary

   SaveETMregisters
       ; (1) Set OS Lock Access Register (OSLAR).
       LDR     R1, =0xC5ACCE55
       MCR     p14, 1, R1, c1, c0, 4           ; Write OSLAR
       ISB
```



```
    ; (2) Get the number of words to save ...
    MRC     p14, 1, R1, c1, c2, 4           ; OSSRR returns size on first read
    ; (3) ... and save this value for the restore
    STR     R1, [R0], #4                    ; Push save count onto the save stack

    ; (4) Loop reading words from the OSSRR.
    CMP     R1, #0                          ; Check for zero
SaveETMregisters_Loop
    ITTT    NE
    MRCNE   p14, 1, R2, c1, c2, 4           ; Load a word of data
    STRNE   R2, [R0], #4                    ; Push onto the save stack
    SUBSNE  R1, R1, #1
    BNE     SaveETMregisters_Loop

    ; (5) Return the pointer to first word not written to.
    ; Leave the OSLAR set as from now on we do not want any changes.
    MOV     R0, R4
    BX      LR
```

6. When you have saved the ETM state information you can power down the macrocell.

7. To restore the debug status, power up the macrocell with **DBGOSLOCKINIT** held HIGH. This causes the Locked bit, bit [1], in the OS Lock Status register to reset to 1. See *OS Lock Status Register (OSLSR), ETMv3.3 and later* on page 3-82. The ETM trace registers are locked and cannot be accessed.

8. You must unlock the CoreSight Lock, if implemented, before performing the register restore sequence. Read the LSR to determine:
   - if the Lock is present
   - the current status of the lock, if it is present.

   If the Lock is currently locked, then you must unlock it by writing `0xC5ACCE55` to the LAR.

9. Read the PDSR to clear the Sticky Register State bit to 0.

10. Although the trace registers are locked, you must now write the lock key, `0xC5ACCE55`, to the OS Lock Access Register, to reset the internal counter used for ETM state restore. See *OS Lock Access Register (OSLAR), ETMv3.3 and later* on page 3-81.

11. Next you must read the OS Save and Restore register, see *OS Save and Restore Register (OSSRR), ETMv3.3 and later* on page 3-84. However, the value returned by this read is UNKNOWN, or IMPLEMENTATION DEFINED, so you must discard this value.

12. Find the number of OSSRR writes needed to restore the status of the ETM trace registers, from the information saved during the OS save process.

13. Now you must write to the OSSRR the number of times indicated at stage 12, writing back the values saved during the OS save process. This restores the ETM state.

14. Finally, write the OS Lock Access register, with any value other than the key, to unlock the ETM Trace Registers and permit debug and trace operation to continue.

Example 3-5 shows performing stages 10 to 14 in a memory-mapped implementation of the ETM registers.

Example 3-5 OS ETM state restore sequence, memory-mapped registers implementation

```
; On entry:
   ;   R0 points to a block of saved ETM registers
   ;   first entry in this block is number of writes required for the restore.
   ;   CoreSight lock has already been unlocked, if necessary.

   RestoreETMregisters
       PUSH    {R4, LR}
       MOV     R4, R0                      ; Save pointer

       ; (1) Set the OS Lock Access Register (OSLAR) and reset pointer.
       ; The lock will already be set, but this write is needed to reset the pointer.
       BL      GetETMRegisterBase          ; Returns base in R0
       LDR     R1, =0xC5ACCE55
       STR     R1, [R0, #0x300]            ; Write OSLAR

       ; (2) Perform dummy read of OSSRR ...
       LDR     R1, [R0, #0x308]            ; Dummy read of OSSRR
       ; (3) ... and get the number of words saved, from block of saved data
       LDR     R1, [R4], #4                ; Get register count from the save stack

       ; (4) Loop writing words from the OSSRR.
       CMP     R1, #0                      ; Check for zero
   RestoreETMregisters_Loop
       ITTT    NE
       LDRNE   R2, [R4], #4                ; Load a word from the save stack
       STRNE   R2, [R0, #0x308]            ; Store a word of data
       SUBSNE  R1, R1, #1
       BNE     RestoreETMregisters_Loop

       ; (5) Clear the OS Lock Access Register (OSLAR)
       ; Writing any non-key value clears the lock, so use the zero value in R1.
       STR     R1, [R0, #0x300]            ; Write OSLAR
       DSB
       ISB

       ; (6) Return the pointer to first word not read.
       MOV     R0, R4
       POP     {R4, PC}
```

Example 3-6 on page 3-124 shows the same operations in an implementation that provides coprocessor access to the ETM trace registers.

Example 3-6 OS ETM state restore sequence, with coprocessor access to ETM registers

```
; On entry:
   ;   R0 points to a block of saved ETM registers,
   ;   first entry in this block is number of writes required for the restore.
   ;   CoreSight lock has already been unlocked, if necessary.

   RestoreETMregisters
       ; (1) Set OS Lock Access Register (GOSLAR) and reset pointer.
       ; The lock will already be set, but this write is needed to reset the pointer.
       LDR     R1, =0xC5ACCE55
       MCR     p14, 1, R1, c1, c0, 4           ; Write OSLAR
       ISB

       ; (2) Perform dummy read of OSSRR ...
        MRC    p14, 1, R1, c1, c2, 4          ; Dummy read of OSSRR
       ; (3) ... and get the number of words saved, from block of saved data
       LDR     R1, [R0], #4                    ; Load size from the save stack

       ; (4) Loop writing words from the DBGOSSRR.
       CMP     R1, #0                          ; Check for zero
   RestoreETMregisters_Loop
       ITTT    NE
       LDRNE   R2, [R0], #4                    ; Load a word from the save stack
       MCRNE   p14, 1, R2, c1, c2, 4           ; Store a word of data
       SUBSNE  R1, R1, #1
       BNE     RestoreETMregisters_Loop

       ; (5) Clear the OS Lock Access Register (OSLAR)
       ; Writing any non-key value clears the lock, so use the zero value in R1.
       MCR     p14, 1, R1, c1, c0, 4           ; Write OSLAR
       ISB

       ; (6) Return the pointer to first word not read.
       MOV     R0, R4
       BX      LR
```

Note

It is IMPLEMENTATION DEFINED:

- which ETM trace registers are saved and restored
- the order in which the ETM trace registers are saved and restored
- the number of OSSRR reads or writes required for ETM state save or restore.

The ETM state save and restore process does not preserve:

- the TraceEnable and ViewData states
- the state of any comparators
- the state of the Instrumentation Resources.

These states are all re-calculated after the restore sequence is complete and the OS Lock is cleared.

3.8.2 ETM behavior when the OS Lock is set

The OS Lock is set by writing the lock key of `0xC5ACCE55` to the OS Lock Access Register, see *OS Lock Access Register (OSLAR), ETMv3.3 and later* on page 3-81. When the OS Lock is set all ETM functions are disabled. This means that:

- Tracing becomes inactive. The FIFO is emptied and no more trace is produced.
- The counters, sequencer, Instrumentation resources, and start/stop block are held in their current state.
- The external outputs are forced LOW.
- Bit [10] of the ETM Control Register maintains its current value, see *ETM Control Register* on page 3-20.
- Any attempt to access the ETM Trace Registers causes an error response, see *Access permissions for ETM registers* on page 3-128.
- It is IMPLEMENTATION DEFINED whether the address comparators maintain their sticky state, see *Address comparators* on page 2-36.

You must use the WFI mechanism to ensure that the FIFO is empty before you remove power from the macrocell.

Note

The WFI mechanism must be present on any processor and ETM combination that provides power down support. See the *Technical Reference Manual* (TRM) for your processor and ETM macrocell for more information.

When the OS Lock is cleared, tracing can restart. The counters, sequencer, start/stop block and Instrumentation resources continue operating from their held state. If the implementation maintains the sticky state of the address comparators during OS Lock then the address comparators continue operating with this held sticky state. However, if the ETM has been powered down since the OS Lock was set:

- the state of the counters, sequencer, start/stop block is restored as part of the OS Save/Restore sequence
- the state of the Instrumentation resources, and the sticky state of the address comparators, is lost.

3.8.3 Guidelines for the ETM trace registers to be saved and restored

The split of ETM registers into trace and management registers is shown in Table 3-86:

Table 3-86 Split of ETM register map into Trace and Management registers

Area	Register numbers	Register addresses
ETM Trace Registers	0x000-0x0BF, 0x0C3-0x3BF	0x000-0x2FF, 0x30C-0xEFF
ETM Management Registers	0x0C0-0x0C2, 0x3C0-0x3FF	0x300-0x30B, 0xF00-0xFFF

This information is also given in Table 3-4 on page 3-16 in the section *ETM Trace and ETM Management registers, from ETMv3.3* on page 3-16.

It is IMPLEMENTATION DEFINED which registers are included in the save and restore mechanism. However, the mechanism must include all registers whose contents are lost in a power-down. Table 3-87 gives a list of the ETM registers typically included in the save and restore mechanism.

Table 3-87 Typical list of ETM registers to be saved and restored

Register number	Register offset	Register name
0x000	0x000	ETM Control
0x002	0x008	Trigger Event
0x003	0x00C	ASIC Control
0x004	0x010	ETM Status
0x006	0x018	TraceEnable Start/Stop
0x007	0x01C	TraceEnable Control 2
0x008	0x020	TraceEnable Event
0x009	0x024	TraceEnable Control 1
0x00A	0x028	FIFOFULL Region
0x00B	0x02C	FIFOFULL Level
0x00C	0x030	ViewData Event
0x00D	0x034	ViewData Control 1
0x00E	0x038	ViewData Control 2
0x00F	0x03C	ViewData Control 3

Table 3-87 Typical list of ETM registers to be saved and restored (continued)

Register number	Register offset	Register name
0x010-0x01F	0x040-0x07C	Address Comparator Value 1-16
0x020-0x02F	0x080-0x0BC	Address Access Type 1-16
0x030-0x03F	0x0C0-0x0FC	Data Comparator Value 1-16
0x040-0x04F	0x100-0x13C	Data Comparator Mask 1-16
0x050-0x053	0x140-0x14C	Counter Reload Value 1-4
0x054-0x057	0x150-0x15C	Counter Enable 1-4
0x058-0x05B	0x160-0x16C	Counter Reload Event 1-4
0x05C-0x05F	0x170-0x17C	Counter Value 1-4
0x060-0x065	0x180-0x194	Sequencer Control
0x067	0x19C	Sequencer State
0x068-0x06B	0x1A0-0x1AC	External Output Event 1-4
0x06C-0x06E	0x1B0-0x1B8	Context ID Comparator Value 1-3
0x06F	0x1BC	Context ID Comparator Mask
0x078	0x1E0	Synchronization Frequency
0x07B	0x1EC	Extended External Input Selection
0x080	0x200	CoreSight Trace ID

3.9 Access permissions for ETM registers

An ETM implements controls on accesses to the ETM registers. These controls depend on the register access model implemented by the ETM. The usual access models are summarized in *ETM register access models* on page 3-7.

An ETM might be part of a system that is implemented with multiple power domains. A typical implementation might implement two domains that can be independently powered down, for example:

- the core power domain powers the processor that is being traced, and contains most of the ETM logic, including the trace registers
- a debug power domain contains the trace output logic, including the programming interface or interfaces.

However, the system that includes the ETM might be implemented in a single power domain. An implementation of this type is called a SinglePower system.

The access controls on memory-mapped accesses to ETM registers depend on whether the system is implemented with multiple power domains, or as a SinglePower system.

Note

If your ETM is accessed using an ARM Debug Interface v5, see the access permissions descriptions in the *ARM Debug Interface v5 Architecture Specification*.

The two types of access that can be made to the ETM registers are described in *Access types* on page 3-129. The access permissions that control ETM register accesses, and restrictions on ETM register accesses, are described in:

- *Restrictions on accesses using a direct JTAG connection* on page 3-130
- *Access permissions for memory-mapped accesses* on page 3-130
- *Access permissions for coprocessor accesses* on page 3-136.

These descriptions include a number of specific terms and abbreviations. The meanings of these are described in *Meanings of terms and abbreviations used in this section* on page 3-129.

3.9.1 Access types

This section refers to two types of access:

Debugger accesses

These are accesses from an external debug device. Typically, the debug device accesses the ETM using a JTAG interface, or a similar interface.

The debugger can access the ETM directly through a JTAG interface, or indirectly using a memory-mapped or coprocessor interface, for example using an ARM Debug Interface v5. For details of this interface see the *ARM Debug Interface v5 Architecture Specification*.

Software accesses

These are accesses that originate from an on-chip device, such as a processor. There is a lock access mechanism that, if implemented, can be used to force software accesses to the ETM registers to be ignored.

An ETM can distinguish between debugger accesses and software accesses.

3.9.2 Meanings of terms and abbreviations used in this section

The following terms and abbreviations are used in the tables that summarize the access permissions:

X Don't care. The outcome does not depend on this condition.

0 The condition is FALSE.

1 The condition is TRUE.

Error Slave-generated error response. Writes are Ignored and reads return an UNKNOWN value. For a memory-mapped access, an error is returned through the memory system. For a coprocessor access, an Undefined Instruction exception is taken.

NPoss Not possible. Accessing the trace registers while the processor is powered down is not possible if a single power domain is implemented. The response is system dependent and IMPLEMENTATION DEFINED.

OK The read or write access succeeds. Writes to RO locations are ignored. Reads from RAZ/WO locations return zero.

Und The access is UNDEFINED and causes an Undefined Instruction exception.

Unp The access has UNPREDICTABLE results. Reads return an UNKNOWN value.

WI Writes Ignored. Reads return the register value.

LAR Lock Access Register, see *Lock Access Register (LAR or LOCKACCESS)* on page 3-90. This is one of the Management registers.

OSLAR OS Lock Access Register, see *OS Lock Access Register (OSLAR), ETMv3.3 and later* on page 3-81. This is one of the Management registers.

OSLSR OS Lock Status Register, see *OS Lock Status Register (OSLSR), ETMv3.3 and later* on page 3-82. This is one of the Management registers.

OSSRR OS Save and Restore Register, see *OS Save and Restore Register (OSSRR), ETMv3.3 and later* on page 3-84. This is one of the Management registers.

CS Lock CoreSight Lock. Indicated by the Lock Status Register, see *Lock Status Register (LSR or LOCKSTATUS)* on page 3-91. This is one of the Management registers.

Note

The CoreSight Lock is described as the Software Lock in previous issues of the *ETM Architecture Specification*.

OS Lock Indicated by the OS Lock Status Register, see *OS Lock Status Register (OSLSR), ETMv3.3 and later* on page 3-82.

Sticky state Indicated by the Sticky Powerdown State bit, bit [2], of the PDSR, see *Device Power-Down Status Register (PDSR)* on page 3-85. This is one of the Management registers.

ETM_PD ETM Power Down status. Indicated by the ETM Power Down bit, bit [0], of the ETM Control Register. See *ETM Control Register* on page 3-20.

3.9.3 Restrictions on accesses using a direct JTAG connection

If an implementation includes a direct JTAG connection to the ETM then, when using that connection:

- it is not possible to access the ETM management registers
- it is not possible to access the Lock Access Register, and the state of the CoreSight lock does not affect register accesses
- it is not possible to access the OS Save and Restore mechanism
- if the core power domain is powered down it is not possible to access the ETM trace registers.

Note

An ARM Debug Interface v5 can include a JTAG-like external interface. Such an interface can provide memory-mapped access to the ETM registers. This section does not apply to such an implementation. For more information see the *ARM Debug Interface v5 Architecture Specification*.

3.9.4 Access permissions for memory-mapped accesses

If an implementation includes a memory-mapped interface to the ETM registers then it must also implement the software lock, controlled by the Lock Access Register (LAR).

An ETM does not implement different access models based on User or Privileged access status.

This section summarizes the access permissions that affect accesses to the ETM registers in two different power supply schemes. The access permissions are given in the following sections:

- *Access permissions for separate core and debug power domains*
- *Access permissions for SinglePower systems* on page 3-133.

Access permissions for separate core and debug power domains

Access permissions are different for Debugger accesses and Software accesses. For a system with memory-mapped ETM registers and separate debug and core power domains:

- For Debugger accesses, the effect of the access permission controls are shown in:
  - Table 3-88, for accesses to registers other than the OS Save and Restore registers
  - Table 3-89 on page 3-132, for accesses to the OS Save and Restore registers.
- For Software accesses, the effect of the access permission controls are shown in:
  - Table 3-90 on page 3-132, for accesses to registers other than the OS Save and Restore registers
  - Table 3-91 on page 3-133, for accesses to the OS Save and Restore registers.

See *Access types* on page 3-129 for more information about Debugger and Software accesses.

Table 3-88 Debugger accesses to ETM memory-mapped registers, except the OS Save and Restore registers, separate debug and core power domains[a]

Powered domains		Sticky state	ETM_PD	OS Lock	Registers: [b, c]			
Debug	Core				Trace	LAR	PDSR	Other Mng[d]
No	X	X	X	X	NPoss	NPoss	NPoss	NPoss
Yes	No	X	X	X	Error	OK	OK	OK
Yes	Yes	0	0	0	OK	OK	OK	OK
Yes	Yes	0	X	1	Error	OK	OK	OK
Yes	Yes	0	1	0	OK[e]	OK	OK	OK
Yes	Yes	1	X	X	Error	OK	OK	OK

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.
b. Entries in the table do not apply to Reserved and UNDEFINED locations in the memory map of the registers.
c. See *ETM Trace and ETM Management registers, from ETMv3.3* on page 3-16 for the definitions of Trace and Management registers in the ETM. These definitions are different to the definitions of Debug and Management Registers used in the Debug part of the *ARM Architecture Reference Manual*.
d. The ETM management registers other than the OSLAR, OSLSR, OSSR, LAR, and PDSR.
e. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETM Control Register might be ignored.

Table 3-89 Debugger accesses to ETM memory-mapped OS Save and Restore registers, separate debug and core power domains[a]

Powered domains		Sticky state	ETM_PD	OS Lock	Registers:		
Debug	Core				OSLSR	OSLAR	OSSRR
No	X	X	X	X	NPoss	NPoss	NPoss
Yes	No	X	X	X	OK	Unp	Unp
Yes	Yes	0	0	0	OK	OK	Unp
Yes	Yes	0	X	1	OK	OK	OK
Yes	Yes	0	1	0	OK	OK	Unp
Yes	Yes	1	X	X	OK	OK	Unp

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.

Table 3-90 Software accesses to ETM memory-mapped registers, except the OS Save and Restore registers, separate debug and core power domains[a]

Powered domains		Sticky state	ETM_PD	OS Lock	CS Lock	Registers: [b, c]			
Debug	Core					Trace	LAR	PDSR	Other Mng[d]
No	X	X	X	X	X	NPoss	NPoss	NPoss	NPoss
Yes	No	X	X	X	0	Error	OK	OK	OK
Yes	No	X	X	X	1	Error	OK	OK[e]	WI
Yes	Yes	0	0	0	0	OK	OK	OK	OK
Yes	Yes	0	X	0	1	WI	OK	OK[e]	WI
Yes	Yes	0	X	1	0	Error	OK	OK	OK
Yes	Yes	0	X	1	1	Error	OK	OK[e]	WI
Yes	Yes	0	1	0	0	OK[f]	OK	OK	OK
Yes	Yes	1	X	X	0	Error	OK	OK	OK
Yes	Yes	1	X	X	1	Error	OK	OK[e]	WI

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.
b. Entries in the table do not apply to Reserved and UNDEFINED locations in the memory map of the registers.

c. See *ETM Trace and ETM Management registers, from ETMv3.3* on page 3-16 for the definitions of Trace and Management registers in the ETM. These definitions are different to the definitions of Debug and Management Registers used in the Debug part of the *ARM Architecture Reference Manual*.
d. The ETM management registers other than the OSLAR, OSLSR, OSSR, LAR, and PDSR.
e. When the CoreSight Lock is set, reads from the PDSR do not clear the Sticky Register Status bit to 0.
f. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETM Control Register might be ignored.

Table 3-91 Software accesses to ETM memory-mapped OS Save and Restore registers, separate debug and core power domains[a]

Powered domains		Sticky state	ETM_PD	OS Lock	CS Lock	Registers:		
Debug	Core					OSLSR	OSLAR	OSSRR
No	X	X	X	X	X	NPoss	NPoss	NPoss
Yes	No	X	X	X	X	OK	Unp	Unp
Yes	Yes	0	X	0	0	OK	OK	Unp
Yes	Yes	0	X	0	1	OK	WI	Unp
Yes	Yes	0	X	1	0	OK	OK	OK
Yes	Yes	0	X	1	1	OK	WI	WI
Yes	Yes	1	X	X	0	OK	OK	Unp
Yes	Yes	1	X	X	1	OK	WI	Unp

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.

Access permissions for SinglePower systems

Access permissions are different for Debugger accesses and Software accesses. For a SinglePower system with memory-mapped ETM registers:

- For Debugger accesses, Table 3-92 on page 3-134 and Table 3-93 on page 3-134 show the effect of the access permission controls.
- For Software accesses, Table 3-94 on page 3-135 and Table 3-95 on page 3-135 show the effect of the access permission controls.

See *Access types* on page 3-129 for more information about Debugger and Software accesses.

Table 3-92 Debugger accesses to ETM memory-mapped registers, except the OS Save and Restore registers, for SinglePower system[a]

Power	Sticky state	ETM_PD	OS Lock	Registers: [b, c]			
				Trace	LAR	PDSR	OtherMng[d]
No	X	X	X	NPoss	NPoss	NPoss	NPoss
Yes	0	0	0	OK	OK	OK	OK
Yes	0	X	1	Error	OK	OK	OK
Yes	0	1	0	OK[e]	OK	OK	OK
Yes	1	X	X	Error	OK	OK	OK

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.
b. Entries in the table do not apply to Reserved and UNDEFINED locations in the memory map of the registers.
c. See *ETM Trace and ETM Management registers, from ETMv3.3* on page 3-16 for the definitions of Trace and Management registers in the ETM. These definitions are different to the definitions of Debug and Management Registers used in the Debug part of the *ARM Architecture Reference Manual*.
d. The ETM management registers other than the OSLAR, OSLSR, OSSR, LAR, and PDSR.
e. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETM Control Register might be ignored.

Table 3-93 Debugger accesses to ETM memory-mapped OS Save and Restore registers, for SinglePower system[a]

Power	Sticky state	ETM_PD	OS Lock	Registers:		
				OSLSR	OSLAR	OSSRR
No	X	X	X	NPoss	NPoss	NPoss
Yes	0	0	0	OK	OK	Unp
Yes	0	X	1	OK	OK	OK
Yes	0	1	0	OK	OK	Unp
Yes	1	X	X	OK	OK	Unp

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.

Table 3-94 Software accesses to ETM memory-mapped registers, except the OS Save and Restore registers, for SinglePower system[a]

Power	Sticky state	ETM _PD	Locks		Registers: [b, c]			
			OS	CS	Trace	LAR	PDSR	OtherMng[d]
No	X	X	X	X	NPoss	NPoss	NPoss	NPoss
Yes	0	0	0	0	OK	OK	OK	OK
Yes	0	X	0	1	WI	OK	OK[e]	WI
Yes	0	X	1	0	Error	OK	OK	OK
Yes	0	X	1	1	Error	OK	OK[e]	WI
Yes	0	1	0	0	OK[f]	OK	OK	OK
Yes	1	X	X	0	Error	OK	OK	OK
Yes	1	X	X	1	Error	OK	OK[e]	OK

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.
b. Entries in the table do not apply to Reserved and UNDEFINED locations in the memory map of the registers.
c. See *ETM Trace and ETM Management registers, from ETMv3.3* on page 3-16 for the definitions of Trace and Management registers in the ETM. These definitions are different to the definitions of Debug and Management Registers used in the Debug part of the *ARM Architecture Reference Manual*.
d. The ETM management registers other than the OSLAR, OSLSR, OSSR, LAR, and PDSR.
e. When the CoreSight Lock is set, reads from the PDSR do not clear the Sticky Register Status bit to 0.
f. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETM Control Register might be ignored.

Table 3-95 Software accesses to ETM memory-mapped OS Save and Restore registers, for SinglePower system[a]

Power	Sticky state	ETM _PD	Locks		Registers:		
			OS	CS	OSLSR	OSLAR	OSSRR
No	X	X	X	X	NPoss	NPoss	NPoss
Yes	0	0	0	0	OK	OK	Unp
Yes	0	X	0	1	OK	WI	Unp
Yes	0	X	1	0	OK	OK	OK
Yes	0	X	1	1	OK	WI	WI

Table 3-95 Software accesses to ETM memory-mapped OS Save and Restore registers, for SinglePower system[a] (continued)

Power	Sticky state	ETM _PD	Locks		Registers:		
			OS	CS	OSLSR	OSLAR	OSSRR
Yes	0	1	0	0	OK	OK	Unp
Yes	1	X	X	0	OK	OK	Unp
Yes	1	X	X	1	OK	OK	Unp

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.

3.9.5 Access permissions for coprocessor accesses

The CoreSight lock is not relevant to coprocessor accesses to the ETM registers, and usually the LAR is not implemented in an implementation that provides coprocessor access. This means that the ETM access permissions for coprocessor accesses do not distinguish between debugger and software accesses to the registers.

For both Debugger and Software accesses:

- for a system with separate Debug and Core power domains, Table 3-96 shows the effect of the access permission controls on coprocessor accesses to the ETM registers.
- for a SinglePower system, Table 3-97 on page 3-137 shows the effect of the access permission controls on coprocessor accesses to the ETM registers.

If the LAR is implemented it has the access permissions shown in the *Other Mng* column of Table 3-96 and Table 3-97 on page 3-137.

Table 3-96 Coprocessor accesses to ETM registers, separate debug and core power domains[a]

Powered[b]		Sticky state	ETM _PD	OS Lock	Registers:					
Dbg	Core				Trace	OSLSR	OSLAR	OSSRR	PDSR	Other[c]
No	X	X	X	X	NPoss	NPoss	NPoss	NPoss	NPoss	NPoss
Yes	No	X	X	X	Error	OK	Unp	Unp	OK	OK
Yes	Yes	0	0	0	OK	OK	OK	Unp	OK	OK

Table 3-96 Coprocessor accesses to ETM registers, separate debug and core power domains[a] (continued)

Powered[b]		Sticky state	ETM _PD	OS Lock	Registers:					
Dbg	Core				Trace	OSLSR	OSLAR	OSSRR	PDSR	Other[c]
Yes	Yes	0	X	1	Error	OK	OK	OK	OK	OK
Yes	Yes	0	1	0	OK[d]	OK	OK	Unp	OK	OK
Yes	Yes	1	X	X	Error	OK	OK	Unp	OK	OK

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.
b. Powered domains. Indicates whether the Debug (Dbg) and Core domains are powered.
c. The ETM management registers other than the OSLAR, OSLSR, OSSR, and PDSR. If the LAR is implemented it has the access permissions shown in this column.
d. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETM Control Register might be ignored.

Table 3-97 Coprocessor accesses to ETM registers, for SinglePower system[a]

Powered	Sticky state	ETM _PD	OS Lock	Registers:					
				Trace	OSLSR	OSLAR	OSSRR	PDSR	Other Mng[b]
No	X	X	X	NPoss	NPoss	NPoss	NPoss	NPoss	NPoss
Yes	0	0	0	OK	OK	OK	Unp	OK	OK
Yes	0	X	1	Error	OK	OK	OK	OK	OK
Yes	0	1	0	OK[c]	OK	OK	Unp	OK	OK
Yes	1	X	X	Error	OK	OK	Unp	OK	OK

a. See *Meanings of terms and abbreviations used in this section* on page 3-129 when using this table.
b. The ETM management registers other than the OSLAR, OSLSR, OSSR, and PDSR. If the LAR is implemented it has the access permissions shown in this column.
c. When ETM_PD is 1, register writes to all Trace registers except certain bits of the ETM Control Register might be ignored.

Chapter 4
Signal Protocol Overview

This chapter describes the types of trace information that are output from the ETM trace port. It contains the following sections:

4.1 About trace information

The trace port outputs two different types of trace information:

Instructions Instruction trace shows the flow of execution of the processor. It provides a list of all the instructions that were executed, giving the address of each instruction, indicating which instructions failed their condition codes and which instructions were subject to an exception. This information is highly compressed. Instruction trace is described in *Instruction trace* on page 4-9.

Data Data trace shows the data accesses performed by the processor that occur as a result of the processor executing a load or store operation. For data accesses it is possible to output both the address and the data value. However, you can choose to compress the data trace by only outputting either the address or the data value. Additional compression is performed by later protocols. Data trace is described in *Data trace* on page 4-14.

Note

In this specification, a *packet* is a discrete quantity of trace information comprising one or more bytes. In previous versions of this document, the word packet and byte were used interchangeably.

4.2 Signal protocol variants

There are three variants of the ETM protocol, defined by the major architecture version as follows:

ETMv1 Implemented in ETM7 and ETM9.

ETMv2 Implemented in ETM10.

ETMv3 Implemented in ETM10RV, ETM11RV, and CoreSight ETMs.

For more information, see *ETM versions and variants* on page 1-6.

4.3 Structure of the trace port

The structure is described in:

- *Signals*
- *Multiplexed trace port (ETMv1.x and ETMv2.x only)* on page 4-5
- *Demultiplexed trace port (ETMv1.x and ETMv2.x only)* on page 4-5.

4.3.1 Signals

The signals output from the ETM are described in:

- *ETMv1.x and ETMv2.x signals*
- *ETMv3.x signals* on page 4-5.

ETMv1.x and ETMv2.x signals

The following signals are output from the ETM:

PIPESTAT Pipeline status. These are output on the pipeline status pins, three for ETMv1.x and four for ETMv2.x.

TRACEPKT Trace packets. These are output on an n-pin trace packet port, where n can be 4, 8, or 16 pins.

TRACESYNC A trace synchronization signal (ETMv1.x only).

TRACECLK The same frequency as the processor clock.

The pipeline status signals provide a cycle-by-cycle indication of what is happening in the Execute stage of the processor pipeline. The n-pin trace packet port provides additional information associated with particular pipeline status events. For example, if a change in instruction flow occurs then it is necessary to output the destination address through the n-pin trace packet port. If the processor has executed an instruction that has failed its condition codes (almost all ARM instructions are conditional), no additional data is required through the trace packet port.

Separating the cycle-accurate pipeline status from the trace packets enables the use of an on-chip FIFO for the trace packet information. You can use the FIFO to buffer trace packets, for example when several branches occur in quick succession. You can pass buffered packets out through the port when the processor is executing several sequential instructions that have no trace packets associated with them. This technique enables the use of a trace port that has a lower data bandwidth than the maximum peak bandwidth.

The width of the trace packet port is determined by the bandwidth of data trace that you require:

- You can use a 4-pin port when the number of data accesses to be traced is relatively low.
- The 8-pin and 16-pin variations of the port are more suitable when medium or high numbers of data accesses must be traced to provide the required debugging capabilities.

Using the on-chip FIFO means that a particular pipeline status event and its associated trace packet (or packets) might not appear in the same cycle. A mechanism is provided to ensure synchronization of the two streams of information. For more information:

For ETMv1.x See *Trace synchronization in ETMv1* on page 5-12.

For ETMv2.x See *Trace synchronization in ETMv2* on page 6-14.

ETMv3.x signals

The following signals are output from the ETM:

TRACECLK

The trace port must be sampled on both edges of this clock. There is no requirement for this to be linked to the core clock.

TRACEDATA[n-1:0]

This signal can be any size. If this is not a multiple of 8 bits then some realignment might be required in the decompressor. See *A-sync, alignment synchronization* on page 7-65 for more information.

TRACECTL

This signal indicates whether trace can be stored this cycle, in conjunction with **TRACEDATA[0]**. This signal does not have to be stored.

4.3.2 Multiplexed trace port (ETMv1.x and ETMv2.x only)

You can implement a narrow (multiplexed) trace port where it is important to reduce the number of output pins to a minimum. You can achieve this by clocking the trace port at twice the processor operating frequency and routing pairs of trace port outputs to a single output pin. Use the **PORTMODE** signals to select multiplexed trace port operation, see *ETM Control Register* on page 3-20. For implementation details, see the appropriate ETM *Technical Reference Manual*.

4.3.3 Demultiplexed trace port (ETMv1.x and ETMv2.x only)

You can implement a wide (demultiplexed) trace port where it is important to reduce the switching rate to a minimum. This is achieved by clocking the trace port at half the processor operating frequency and routing trace port outputs to pairs of output pins. Use the **PORTMODE** signals to select demultiplexed trace port operation (see *ETM Control Register* on page 3-20). For implementation details, see the appropriate ETM *Technical Reference Manual*.

4.3.4 ETM structures

The ETM structures for the different architectures are shown in:

- *ETMv1.x* on page 4-6
- *ETMv2.x* on page 4-6
- *ETMv3.x* on page 4-6.

ETMv1.x

Figure 4-1 shows the structure of devices implementing ETMv1.x.

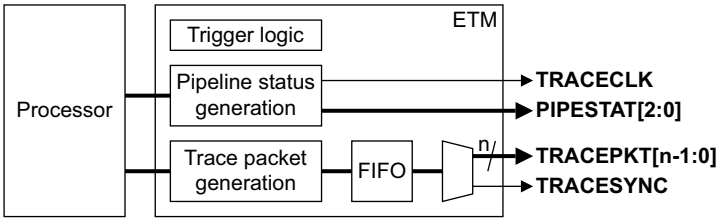


Figure 4-1 ETMv1.x structure

ETMv2.x

Figure 4-2 shows the structure of devices implementing ETMv2.x.

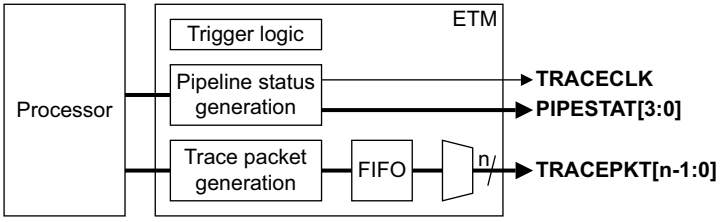


Figure 4-2 ETMv2.x structure

ETMv3.x

Figure 4-3 shows the structure of devices implementing ETMv3.x.

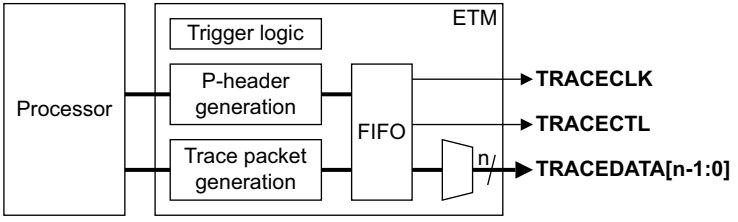


Figure 4-3 ETMv3.x structure

4.4 Decoding required by trace capture devices

The two conditions that must be decoded by TCDs, for example, a TPA, logic analyzer, or on-chip trace buffer, are described in:

- *Trigger conditions*
- *Trace disabled conditions*.

4.4.1 Trigger conditions

When the trigger occurs, trace capture must end before the current trace is overwritten by newer trace. See *Triggering a trace run* on page 2-16 for more information.

For ETMv1.x the condition for detecting a trigger is:

- **PIPESTAT[2:0]** has the value `0x6` (TR).

For ETMv2.x the condition for detecting a trigger is:

- **PIPESTAT[3:0]** has the value `0x6` (TR).

For ETMv3.x the conditions for detecting a trigger are:

- **TRACECTL** is HIGH
- **TRACEDATA[0]** is LOW.

4.4.2 Trace disabled conditions

This indicates that the current cycle must not be captured because it contains no useful data. For ETMv1.x the trace disabled conditions are:

- **PIPESTAT[2:0]** has the value `0x7` (TD)
- **TRACEPKT[0]** is LOW.

For ETMv2.x the trace disabled conditions are:

- **PIPESTAT[3:0]** has the value `0x7` (TD)
- **TRACEPKT[0]** is LOW.

For ETMv3.x the trace disabled conditions are:

- **TRACECTL** is HIGH
- **TRACEDATA[0]** is HIGH.

Storage of TRACECTL

TCDs designed for ETMv3.x can discard **TRACECTL** after it has been used to detect trigger and trace disabled conditions, storing only **TRACEDATA**. This means that more efficient packing of the trace data in the TCD is possible.

Because future devices might not be able to output the trigger condition on the trace port without corrupting the trace stream, TCDs must be able to capture the trigger condition without capturing **TRACEDATA**. The CoreSight formatting protocol is an example where **TRACEDATA** must not be captured under a trigger condition. For more information, see the *CoreSight Architecture Specification*.

Table 4-1 shows the situations that must be recognized.

Table 4-1 Trace disabled conditions

TRACECTL	TRACEDATA[0]	TRACEDATA[1]	Action
0	x	x	Capture **TRACEDATA[n:0]**
1	1	x	Trace disabled, discard **TRACEDATA[n:0]**
1	0	0	Trigger[a], capture **TRACEDATA[n:0]**
1	0	1	Trigger[b], discard **TRACEDATA[n:0]**

a. This is how all ETMv3.x devices output a trigger.
b. This is for future devices where the trigger is indicated on the trace port, but **TRACEDATA** must not be captured because this might corrupt the trace stream.

TCDs that are only designed to capture ETMv3.x trace only have to inspect **TRACECTL** and **TRACEDATA[0]** to detect the trigger condition. However, if a TCD is to be compatible with future devices, it must inspect **TRACEDATA[1]**.

4.5 Instruction trace

Instruction trace works by outputting the destination address of branches. You can use this information, along with the pipeline status signals, to determine how many instructions are executed after each branch.

4.5.1 Instruction trace filtering

The main technique that you can use to reduce the bandwidth required by instruction trace is to enable or disable tracing dynamically, using the **TraceEnable** function (see *TraceEnable and filtering the instruction trace* on page 2-20 for more information).

4.5.2 Direct and indirect branches

For some branches it is not necessary to output the destination address. For direct branches (`B`, `BL`, or `BLX <immediate>` instructions) the assembler code provides an offset to be added to the current PC. All direct branches are branches whose target can be determined solely from the executed instruction. Therefore, to calculate the destination of the branch, it is necessary only to know the address of the instruction, along with the fact that it executed. See *Direct branch instructions* on page 4-23 for a list of direct branch instructions.

The branch address must be output only when the program flow changes for a reason other than a direct branch. These are collectively known as indirect branches. Examples of indirect branches are:

- a load instruction (`LDR` or `LDM`) with the PC as one of the destination registers
- a data operation (`MOV` or `ADD`, for example) with the PC as the destination register
- a `BX` instruction, that moves a register into the PC
- a `SVC` instruction or an Undefined Instruction exception
- all other exceptions, such as interrupt, abort, and processor reset.

Exceptions and state changes (between ARM, Thumb, ThumbEE and Jazelle states) are indicated by outputting the destination address. Depending on the ETM architecture version, there might be no specific indication that an exception occurred.

Note

ThumbEE state is only supported from ETMv3.3. This state is typically used when executing dynamically compiled code, for example by Jazelle RCT technology.

4.5.3 Exceptions

For processors that support vector table relocation, the vector table resides at:

- `0x00-0x1C` if **HIVECS** is LOW
- `0xFFFF0000-0xFFFF001C` if **HIVECS** is HIGH.

Note

HIVECS is an input signal to the processor core, that is tied HIGH to indicate the High-vectors configurations. On some processors this signal is called **VINITHI**.

In ARMv6 interrupts might be relocatable to any address, and in processors implementing the Security Extensions all exceptions are relocatable. These processors are only supported by ETMv3.

In ETMv1 and ETMv2, detection of exceptions is only possible if the value of **HIVECS** remains constant during the trace run, and this value is known to the decompressor. In ETMv3 and later, knowledge of **HIVECS** is not required because exceptions are explicitly flagged in the trace.

The possible exception types and the corresponding trace behavior are:

Processor reset This causes the current instruction to be abandoned. When the processor reset is deasserted a branch occurs to the reset vector.

Interrupts (IRQ and FIQ)

These cause the interrupted instruction to become a branch to the appropriate exception vector.

Prefetch abort When the aborted instruction is executed, it becomes a branch to the Prefetch abort vector.

Data abort The instruction on which the data abort is signaled becomes a branch to the Data abort vector. This instruction might have data traced, that must be ignored.

In ETMv1 and ETMv2, all branches to these exception vectors must be treated as an exception.

In ETMv3, these exceptions are explicitly flagged in the protocol at the time of the branch. A branch to these exception vectors that is not flagged must be treated as a normal branch.

Undefined Instruction

When the undefined instruction is encountered, it becomes a branch to the Undefined Instruction vector.

SVC When the `SVC` instruction is executed, it becomes a branch to the SVC vector.

SMC When the `SMC` instruction is executed, it becomes a branch to the SMC vector.

Note

In each of these cases, the next instruction traced is the instruction at the exception vector.

The trace decompressor can treat exceptions as belonging to one of two categories:

Processor reset, IRQ, FIQ, prefetch abort

In ETMv1 and ETMv2, all branches to these exception vectors must be treated as an exception. The instruction on which the branch occurred was canceled and must not be marked as executed.

In ETMv3, these exceptions are explicitly flagged in the protocol at the time of the branch. A branch to these exception vectors that is not flagged must be treated as a normal branch.

Data abort, Undefined Instruction, SVC, SMC

In ETMv1 and ETMv2, all branches to these exception vectors must be treated as an exception. The instruction on which the branch occurred was not canceled and must be marked as executed.

In ETMv3, these exceptions are explicitly flagged in the protocol at the time of the branch. A branch to these exception vectors that is not flagged must be treated as a normal branch.

For some ARM7 and ARM9 processors, tracking the pipeline is not always possible under some exception sequences. This might result in tracing an exception by changing the pipeline status for the last instruction executed to a *Branch Executed* (BE) or *Branch Executed with Data* (BD) to the exception vector. See the relevant ETM *Technical Reference Manual* for more information.

Instruction execute means that the instruction at that address has reached the Execute stage of the pipeline and includes instructions that fail their condition codes. This is slightly different from the pipeline status codes that indicate instruction executed and condition code test passed (these codes have the letter E, standing for “Executed”, in their mnemonics), and instruction executed and condition code failed (these codes have the letter N, standing for *Not Executed*, in their mnemonics).

If the instruction reaches execution but fails its condition code test, a pipeline status code or P-header is generated that includes the letter N in its mnemonic, to indicate an instruction not executed. ETMv1.2 introduced the facility to control trace using the result of the condition code test whenever an instruction is executed.

Table 4-2 shows how ETMv3 traces exceptions. In many cases, exceptions can be either cancelling or non-cancelling depending on when the last instruction traced completed execution. Where an exception can be cancelling or non-cancelling, the value of r14 in the exception handler and the address of the last traced instruction determine whether the last traced instruction was cancelled. For example, when an FIQ occurs, if the last traced instruction was at r14-4, this instruction was cancelled because the exception handler returns to re-execute this instruction. If the last traced instruction was at r14-8, the last traced instruction was not cancelled.

Table 4-2 ETMv3 exception tracing

Exception	Cancelling	Non-cancelling
Halt, entry to Debug state	Last instruction traced did not complete	Last instruction traced completed
Secure Monitor Call[a] (SMC)	-	SMC always completes
Asynchronous data abort	Last instruction traced (r14-8) did not complete	Last instruction traced (r14-12) completed
Jazelle/ThumbEE[b]	Last instruction traced caused the exception	Last instruction traced did not cause the exception
Processor reset	This is always cancelling	-
Undefined Instruction	-	Always non-cancelling

Table 4-2 ETMv3 exception tracing (continued)

Exception	Cancelling	Non-cancelling
Supervisor Call[a] (SVC)	-	Always non-cancelling
Prefetch abort[c] External prefetch abort[b] Breakpoint debug exception[b] Watchpoint debug exception[b] BKPT instruction[b]	Last instruction traced (r14-4) did not complete	Last instruction traced (r14-8) completed
DABORT[b] Vector Catch Debug Exception	Last instruction traced (r14-8) did not complete	Last instruction traced (r14-12) completed
Generic exception for IMPLEMENTATION DEFINED exceptions	Last instruction traced did not complete	Last instruction traced completed
IRQ	Last instruction traced (r14-4) did not complete	Last instruction traced (r14-8) completed
NMI FIQ	Last instruction traced (r14-4) did not complete	Last instruction traced (r14-8) completed

a. Before ARMv7, the Secure Monitor Call (SMC) was called the Secure Monitor Interrupt (SMI), and the Supervisor Call (SVC) was called the Software Interrupt (SWI).
b. Jazelle and ThumbEE exceptions can be cancelling or non-cancelling. See *Jazelle and ThumbEE exceptions* on page 4-13.
c. ARM recommends that you trace and cancel these exceptions because it is useful to output the instruction that caused the exception in the trace stream. In these cases, the instruction that causes the exception is traced and then indicated as cancelled.

Processor Reset exceptions are always traced as cancelling. However, zero or more instructions might not complete execution when a processor Reset exception occurs.

Where an instruction is considered for tracing, subject to **TraceEnable**, it must be considered by the address comparators. For example, a prefetch abort can be traced in one of two ways:

- the instruction that prefetch aborts is traced and then indicated as cancelled by a prefetch abort exception
- the instruction that prefetch aborts is not traced and the prefetch abort exception is non-cancelling.

When traced as cancelling, the prefetch aborted instruction address is compared by the comparators and is subject to the rules defined in *Address comparators* on page 2-36. When traced as non-cancelling, the ETM does not know the address of the prefetch aborted instruction because the instruction is not traced so the comparators do not compare based on this address.

Jazelle and ThumbEE exceptions

For Jazelle and ThumbEE exceptions, the cancelling concept is:

- if the instruction that caused the exception, for example an index check instruction, is traced, it must be cancelled
- if the instruction was not traced, the exception must be non-cancelling.

For null pointer checks on loads and stores, if the instruction is traced the exception must be cancelling. Otherwise it is non-cancelling. If the data transfer is traced, decompression tools must discard the data transfer, and the ETM comparators treat the data transfer as if it were an aborted data transfer. For more information, see *Address comparators* on page 2-36.

4.5.4 32-bit Thumb instructions

The behavior of 32-bit Thumb instructions depends on the setting of bit [18], Support for 32-bit Thumb instructions, of the ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71:

- If bit [18] is set to 1, each 32-bit Thumb instruction is traced as a single instruction. Address comparators only match on the address of the lower halfword of the instruction.
- if bit [18] is set to 0, each 32-bit Thumb instruction is traced as two instructions. Exceptions can occur between the two instructions. Address comparators match on the address of either halfword of the instruction.

4.5.5 Thumb CBZ and CBNZ instructions

If a CBZ or CBNZ instruction does not branch then it is traced as an instruction that failed its condition code. CBZ and CBNZ are direct branches and do not require a branch packet output when the condition matches, unless the branch output bit, bit [8], of the ETM control register is set to 1. For more information, see *Direct branch instructions* on page 4-23.

Note

Although the CBZ or CBNZ instruction makes the required comparison, NE or E, with zero, the instruction does not update the CPSR flags.

4.6 Data trace

Data trace works by outputting the data accesses (that is address, data value, or both) performed by the processor.

Tracing every data access might require a large number of pins to achieve the required bandwidth, and data trace cannot use all of the compression techniques that are available with instruction trace. However, data trace can be optimized as follows:

User-controlled optimization

The following sections describe how you can avoid generating unnecessary data trace:

- *Data access filtering*
- *Address and data selection*.

Data trace compression performed by the ETM

The ETM can compress the data trace by reducing the number of bits output for the data address, as described in:

- *Address compression performed by the ETM* on page 5-16 for ETMv1
- *Address compression performed by the ETM* on page 6-28 for ETMv2
- *Data tracing* on page 7-44. ETMv3 supports data address compression and leading zero compression.

4.6.1 Data access filtering

The main technique that you can use to reduce data trace is to be selective about the data accesses that are observed. The trace filtering facilities generate an internal function called **ViewData**, and this is used to indicate whether data from individual locations or address regions is to be traced. See *ViewData and filtering the data trace* on page 2-26 for details of the generation of the **ViewData** signal.

4.6.2 Address and data selection

You can use choose what information is output for each data access, to reduce the bandwidth of the data trace. The following three modes of operation are available:

Address only

Broadcasts only the address of the transfer, or the first address in the case of a load/store multiple. This approach is useful when checking the code to ensure that the correct address is generated for all transfers.

Data value only

Broadcasts only the data value of the transfer.

This approach is useful when there is a high degree of confidence that the address of the transfers is generated correctly, and the address of a transfer can easily be inferred by looking at the instruction that caused the access. For example, if the instruction is located in a section of code for the UART mode control you can assume that the address is that of the UART control register.

Address and data value

The address and data value option outputs both the address and data value of the transfer. This ensures that all information about the transfer is known. However, it requires a larger overall bandwidth through the trace port than the address-only or data-only options.

Because of the techniques used to decompress the trace information after it is captured, you must select a single mode of operation to apply to all data transfers. This means, for example, that in a single trace it is not possible to provide the address only for some transfers and data value only for others.

4.6.3 Preloads

Data that is transferred by a preload `PLD` instruction is ignored by the ETM. Preloads are not recognized as data transfer instructions because they do not cause any data trace.

4.6.4 Asynchronous data aborts

An asynchronous data abort occurs when the cache or memory system generates an error after it has signalled to the processor that the data transfer has completed. The abort is unrecoverable and usually results in the termination of the process that caused it. Examples of generating an asynchronous data abort include a program executing in Non-secure state writing to Secure memory, or a parity error in the memory system.Writes that are subject to an asynchronous data abort might be traced as having completed, because this is the view from the processor. Out-of-order data corresponding to reads subject to an asynchronous data abort might not be traced. When an asynchronous data abort is traced, you must consider this when interpreting the data trace leading up to the abort.

Note

In previous versions of this document:

- synchronous aborts were described as precise aborts
- asynchronous aborts were described as imprecise aborts.

4.7 Context ID tracing

Context ID tracing is possible only with ETMv1.2 or later.

Note

Context ID was previously known as Process ID. This has been changed to avoid confusion with the *Fast Context Switch Extensions* (FCSE) field, sometimes referred to as the FCSE Process ID.

Context ID is a 32-bit value accessed through CP15 register c13 that is used to identify and differentiate between different code streams. You can use the Context ID in:

- systems that dynamically load or overlay code into shared RAM
- complex operating systems to enable the trace to be filtered based on which process is executing.

Without the Context ID, software might not be able to determine the instruction address space from which the traced instructions are executing. This can result in incorrect decompression in systems with dynamic memory maps.

Most ARM processors have defined a Context ID register in the system control coprocessor (CP15). See the appropriate *Technical Reference Manual* for more information.

Where supported, the instruction to write to the Context ID register is:

```
MCR p15, 0, <Rd>, c13, c0, 1
```

The instruction to read the Context ID register is:

```
MRC p15, 0, <Rd>, c13, c0, 1
```

The ProcIDSize bits (bits [15:14]) of the ETM Control Register set to 1 the number of bytes of the Context ID bus that are traced. These bits are encoded as listed in Table 4-3.

Table 4-3 ETM Control Register ProcIDSize bits

Bits [15:14]	Meaning
b00	No Context ID tracing
b01	Context ID bits [7:0] traced
b10	Context ID bits [15:0] traced
b11	Context ID bits [31:0] traced

When Context ID tracing is enabled and the current value of the Context ID Register changes, this is output in the trace. The following situations might cause the Context ID to be traced:

- `MCR` instruction changes in the current Context ID.
- `MCR` instruction from Non-secure state changes the Non-secure Context ID in systems that implement the Security Extensions.

- Changing from the Non-secure state to the Secure state in systems that implement the Security Extensions.
- Changing from the Secure state to the Non-secure state in systems that implement the Security Extensions.
- A processor reset caused the Context ID to be reset to a known value.

In all these cases, the Context ID might not be traced if the watched part of the Context ID value does not change. For example, if you are only tracing bits [7:0] of the Context ID:

- an implementation might not trace the Context ID on a Context ID change that does not change bits [7:0] of the Context ID
- if any of Context ID bits [7:0] change, they are always traced.

In a system that implements the Security Extensions, if the Non-secure Context ID is changed from the Secure state this is not a change to the current Context ID and is not traced as a Context ID change. In this case, the data for the `MCR` instruction that changes the Non-secure Context ID is traced as a normal data packet and is subject to the normal rules for tracing coprocessor register transfers.

After a processor reset, the Context ID is UNKNOWN. When tracing through a processor reset, the current Context ID is UNKNOWN after the Reset exception until it is explicitly changed by writing to the CP15 Context ID Register. If the processor resets the Context ID to a known value, this value is output when Context ID changes, and this new Context ID is used for Context ID comparisons. Otherwise, the ETM uses the last known Context ID for comparisons. If the Context ID changes, the new Context ID is used for comparisons from when the first instruction is executed after the reset.

4.8 Debug state

When the ARM processor enters debug state, instruction execution stops. This means that tracing also stops and the FIFO continues to drain until empty.

Instructions executed in debug state are ignored by the ETM.

When the ARM processor exits debug state, tracing restarts if tracing is enabled. The reason code that is generated indicates that the ARM processor has exited from debug state.

If an overflow has occurred on entry into debug state, the debug tools can detect this by reading the ETM Status Register. For more information see *ETM Status Register, ETMv1.1 and later* on page 3-33 and *Processor stalling, FIFOFULL* on page 2-31.

For more information about Debug state see:

- the Debug part of the *ARM Architecture Reference Manual*
- the data sheet or *Technical Reference Manual* for the appropriate processor.

4.9 Endian effects and unaligned access

ARM processors support big-endian modes of operation, and some forms of unaligned access. When these occur, the data address requested by the instruction can be different from the address accessed in memory, and the order of the bytes in a word or halfword might be changed. Depending on the situation, you might be interested in the memory view or the processor view of the address and data value accessed.

4.9.1 Summary of ARM behavior

For a load or store, the mapping between the data address and data value in memory, and the value transferred to or from the register depends on:

- the alignment, bits [1:0], of the address
- the size of the transfer
- the value of the U and B bits in the CP15 System Control Register
- the value of the Ebit in the *Current Program Status Register* (CPSR).

The U bit controls whether certain unaligned loads and stores, such as `LDR` and `STR`, rotate the data value in the accessed word, or perform a true unaligned access (ARMv6 and later). This bit is set to 1 shortly after a processor reset and is normally left unchanged after this point. Therefore, its value is not included in the trace.

The B bit controls whether little-endian (LE) or word-invariant big-endian (BE-32) mappings apply to data transfers. In BE-32, byte and halfword transfers have their addresses modified so that the lowest addressed byte corresponds to the most significant byte of a word. For example, a load of a byte at address `0x2000` in fact loads the value at memory address `0x2003`. This bit is set to 1 shortly after a processor reset and is normally left unchanged after this point. Therefore, its value is not included in the trace.

The E bit controls whether little-endian (LE) or byte-invariant big-endian (BE-8, ARMv6 only) mappings apply to data transfers. It cannot be set to 1 at the same time as the B bit. In BE-8, halfword and word transfers have the bytes in the data value swapped so that the most significant byte of a word is stored in the lowest addressed byte. Because this bit is in the CPSR, the setting of this bit can change between transfers. If it is set to 1, this is indicated in the trace with the data transfer if data address tracing is enabled.

If either the B or E bit is set to 1 during a VFP double-precision transfer, the word at the lower location corresponds to the high VFP register number, and the word at the higher location corresponds to the lower register number. For example, if a double-precision value is loaded from address `0x2000` into registers r0 and r1 in LE, the word at `0x2000` is loaded into register r0 and the word at `0x2004` is loaded into register r1. If however the transfer occurs in BE-8 or BE-32, the word at `0x2000` is loaded into register r1 and the word at `0x2004` is loaded into register r0.

For more information, see the *ARM Architecture Reference Manual*, where this topic is covered extensively.

4.9.2 Representation of data in the trace

If the address requested by the instruction does not match the address accessed in memory, the address requested by the instruction is traced. This means the *processor view* is traced.

When considering data tracing:

The memory view refers to the order in which bytes are transferred to or from memory

The processor view refers to the order of the bytes in the source or destination register.

When these two views do not match, the version traced depends on the ETM architecture version:

- in ETMv1, the memory view of the data is traced
- in ETMv2 and later, the processor view of the data is traced.

This only applies to the order of data in a word or halfword. When VFP double-precision transfers are traced in BE-8 or BE-32, the order of the words is given according to the memory view. The trace decompressor must therefore be aware of the current endianness configuration to reconstruct the data transferred to or from each register for these instructions.

The values traced are the values before sign extension. Therefore an `LDRSB` of the value `0xAB` causes the value `0xAB` to be traced, not `0xFFFFFFAB`.

4.10 Definitions

This section defines some terms used in the signal protocol definitions.

4.10.1 Load/Store Multiple (LSM) instructions

Some sections of this specification refer to *Load/Store Multiple* (LSM) instructions. These are instructions that passed their condition codes and cause more than one data transfer.

The LSM instructions are:

LDC{2} Load coprocessor.

LDM{<amode>} Load multiple.

LDRD Load register dual.

LDREXD Load register exclusive doubleword.

MCRR{2} Move to coprocessor from two ARM core registers.

MRRC{2} Move to two ARM core registers from coprocessor.

POP Pop multiple registers.

PUSH Push multiple registers.

RFE Return from exception.

SRS Save return state.

STC{2} Store coprocessor.

STM{<amode>} Store multiple.

STRD Store register dual.

STREXD Store register exclusive doubleword.

SWP Swap a word.

SWPB Swap a byte.

VLDM Vector load multiple. Loads multiple extension registers from consecutive memory locations.

VLDn Vector load, where n is the number of elements to load, from 1 to 4.

VLDR.64 Vector load register, 64-bit option.

VMOV, between two ARM core registers and two single-precision registers

Vector move that transfers the contents between two single-precision VFP registers and two ARM core registers.

VMOV, between two ARM core registers and a doubleword extension register

Vector move that transfers the contents between two ARM core registers and a doubleword extension register.

VPOP Vector pop. Loads multiple consecutive extension registers from the stack.

VPUSH Vector push. Stores multiple consecutive extension registers to the stack.

VSTM Vector store multiple. Stores multiple extension registers to consecutive memory locations.

VSTn Vector store, where n is the number of elements to store, from 1 to 4.

VSTR.64 Vector store register, 64-bit option.

4.10.2 Data Instructions

An instruction is a data instruction if it passed its condition code test and caused a data transfer.

Data instructions comprise all LSM instructions and the following:

BXJ Branch and exchange Jazelle.

CLREX Clear exclusive.

LDR{T} Load register.

LDRB{T} Load register byte.

LDREX Load register exclusive.

LDREXB Load register exclusive byte.

LDREXH Load register exclusive halfword.

LDRH{T} Load register halfword.

LDRSB{T} Load register signed byte.

LDRSH{T} Load register signed halfword.

MCR{2} Move to coprocessor from ARM core register.

MRC{2} Move to ARM core register from coprocessor.

STR{T} Store register.

STRB{T} Store register byte.

STREX Store register exclusive.

STREXB Store register exclusive byte.

STREXH Store register exclusive halfword.

STRH{T} Store register halfword.

TB{B|H} Table branch, Thumb and ThumbEE instruction sets only.

VDUP**, ARM core register**

Vector duplicate. Duplicates an element from an ARM core register into every element of the destination vector.

VLDR.32 Vector load register, 32-bit option.

VMOV**, ARM core register to scalar**

Vector move that copies a byte, halfword, or word from an ARM core register into an Advanced SIMD scalar.

VMOV**, between ARM core register and single-precision register**

Vector move that transfers the contents between a single-precision VFP register and an ARM core register.

VMOV**, scalar to ARM core register**

Vector move that copies a byte, halfword, or word from an Advanced SIMD scalar to an ARM core register.

VMRS Vector move, extension system register (FPSCR) to general-purpose register.

VMSR Vector move, general-purpose register to extension system register (FPSCR).

VSTR.32 Vector store register, 32-bit option.

BXJ might not trace data in all implementations, but can always be treated as a data instruction. It is IMPLEMENTATION DEFINED whether data is traced. If data is traced, it is a load of Jazelle local variable 0.

Preload (PLD) instructions are not data instructions because they do not cause any data trace. See *Preloads* on page 4-15.

Data instructions in Jazelle state are IMPLEMENTATION DEFINED.

Note

- The following instructions are not LSM instructions and are not data instructions even though their mnemonics are the same as other LSM and data instructions:

  VMOV**, register**

  VMOV**, immediate.**

- The following instruction is not a data instruction even though it has the same mnemonic as a data instruction:

  VDUP**, scalar.**

4.10.3 Direct branch instructions

Direct branches in ARM, Thumb, and ThumbEE instruction sets are defined to be the following:

B	Branch.
BL	Branch with link.
BLX immed	Branch with link and exchange instruction sets, immediate.
CBZ	Compare and branch on zero, Thumb and ThumbEE instruction sets only.
CBNZ	Compare and branch on nonzero, Thumb and ThumbEE instruction sets only.
ENTERX	Enter ThumbEE state, from Thumb state only.
LEAVEX	Leave ThumbEE state, from ThumbEE state only.

In ETMv1.x there are no direct branches in Jazelle state. In ETMv3.0 and later direct branches in Jazelle state are defined to be the following:

if<cond>	Conditional branch.
if_icmp<cond>	Conditional branch.
if_acmp<cond>	Conditional branch.
goto	Unconditional branch.
jsr	Jump to subroutine.
ifnull	Conditional branch.
ifnonnull	Conditional branch.
goto_w	Long unconditional branch.
jsr_w	Long jump to subroutine.

Any branch not caused by a direct branch is traced as an indirect branch, even if the destination can be inferred by the decompressor. For example, the ARM instruction ADD pc, pc, #4 is an indirect branch.

4.11 Coprocessor operations

This section describes the three different types of coprocessor instruction:

- *Coprocessor data operation*
- *Coprocessor data transfer*
- *Coprocessor register transfer*.

4.11.1 Coprocessor data operation

A *Coprocessor Data Operation* (CPDO) occurs when the processor executes a `CDP` instruction. A CPDO instructs a coprocessor to perform an internal data operation. This cannot produce any data trace and is treated exactly the same as any other instruction.

4.11.2 Coprocessor data transfer

A *Coprocessor Data Transfer* (CPDT) occurs when the processor executes an `LDC` or `STC` instruction. CPDT instructions are treated in the same way as standard processor loads and stores. Only when you analyze the disassembled trace information can you determine that the data access involved a coprocessor.

ETMv1

In ETMv1, it is important that the number of words transferred for an `LDC` or `STC` instruction can be statically determined from the instruction. To do this, the decompressor requires specific knowledge of the coprocessor involved because the number of data packets is not directly encoded in the instruction but is determined by the coprocessor during execution of the instruction. The debugger must be aware of this mapping.

4.11.3 Coprocessor register transfer

A *Coprocessor Register Transfer* (CPRT) occurs when the processor executes any of the following instructions:

- `MCR`
- `MCRR`
- `MRC`
- `MRRC`.

These instructions move data between the processor registers and the coprocessor. There is no address related to a CPRT. The data size is always 32 bits for `MCR` and `MRC`, and 64 bits for `MCRR` and `MRRC` instructions.

Up to ETMv2.x

ViewData is ignored when the processor determines whether to trace a CPRT. This is because a CPRT does not have an associated data address associated. Instead, a configuration bit selects whether the data must be captured. This is the as MonitorCPRT bit, bit [1] of register `0x00`, the ETM Control Register.

ETMv3.0 upwards

Although CPRTs cannot be filtered based on data address, they can be filtered based on the address of the instruction associated with them. A bit in the ETM Control Register, FilterCPRT, is used with MonitorCPRT to enable you to use **ViewData**. See *ETM Control Register* on page 3-20 for information about how to use **ViewData** with CPRTs.

4.12 Wait For Interrupt and Wait For Event

Processors might implement either or both of:

- a *Wait For Interrupt* (WFI) mechanism
- a *Wait For Event* (WFE) mechanism.

If implemented, these mechanisms enable the processor to execute a `WFI` or `WFE` instruction and then stop execution until an interrupt or event occurs. Some systems might stop the clocks, or save energy by removing power to the processor while waiting for the interrupt or event.

When tracing a processor that halts execution on a `WFI` or `WFE` instruction, the ETM behaves as follows:

1. The `WFI` or `WFE` instruction is presented in the trace if **TraceEnable** is HIGH and instruction tracing is enabled.

2. The ETM drains its FIFO. When this is complete, it does not generate any more trace. The system must not stop the clocks until the FIFO has drained.

3. If power is removed or the ETM clock is stopped, it is IMPLEMENTATION SPECIFIC whether cycle accuracy is maintained while waiting for the interrupt or event.

An ETM might stop tracing while the processor is waiting for an interrupt or waiting for an event. In this case, when the interrupt or event occurs tracing restarts, using the normal trace starting sequence. If the ETM FIFO overflowed before the `WFI` or `WFE` instruction executed, this must be indicated when tracing restarts. If the time spent waiting for the interrupt or event is short, for example execution restarts before the ETM FIFO has fully drained, the ETM might not restart tracing immediately and trace might be lost.

If an implementation removes power while waiting for the interrupt or event, it must use the OS Save and Restore registers to save the ETM state before power is removed, and to restore the ETM state when power is restored. For more information, see *Power-down support, ETMv3.3 and later* on page 3-119.

Chapter 5
ETMv1 Signal Protocol

This chapter describes the signal protocol for ETMv1. It contains the following sections:

5.1 ETMv1 pipeline status signals

You can consider the pipeline status as a view of the Execute stage of the pipeline. Any side effects of the instruction, for example an aborted load or a write to the PC, are observed immediately, before the pipeline status for the next instruction is generated. This means that the protocol is independent of the exact pipeline implementation of the ARM core.

Each executed instruction generates a single pipeline status message on the **PIPESTAT** pins of the port. These are shown in Table 5-1.

Table 5-1 PIPESTAT messages

Status	Mnemonic	Meaning	Description
b000	IE	Instruction Executed	Indicates an instruction has been executed that has not generated any associated trace packets. This includes load or store instructions that did not have their data traced. Usually used for an operation that did not cause a branch, but it is also used for direct branches, that is where the destination can be worked out by referring back to the code image.
b001	ID	Instruction Executed with Data	A load or store instruction has been executed, and the memory access is output on the trace port.
b010	IN	Instruction Not Executed	An instruction has reached the Execute stage of the pipeline, but failed its condition code test.
b011	WT	Wait	No instruction was executed in the cycle and the pipeline has not advanced. This might be for several reasons. For example, the memory system might have asserted the wait signal to the processor, or the processor might be performing an internal cycle. Wait cycles are also generated when tracing is disabled. Wait cycles are used to output trace packet data. If no packet is output in this cycle, this status is replaced by Trace Disabled.
b100	BE	Branch Executed	This is generated when an indirect branch is executed (that is a branch whose target address cannot be directly inferred from the source code). A destination address is required for the branch. Direct branches can also generate this status. The trace port can optionally be switched into a mode where it outputs this status for all branches.

Table 5-1 PIPESTAT messages (continued)

Status	Mnemonic	Meaning	Description
b101	BD	Branch Executed with Data	Used whenever a data access causes a branch, (occurs when a load instruction has the PC as the destination register).
b110	TR	Trigger	Indicates that a trigger condition has occurred. See *Trigger PIPESTAT signals*.
b111	TD	Trace Disabled	The trace might be disabled to prevent the TPA from filling up with unwanted information. This encoding is used when the trace is disabled or when no packet is output on a wait cycle.

For multi-cycle instructions, and for cycles where the memory system has asserted a wait signal, wait cycles are indicated.

Instructions that are fetched but not executed, because of an executed branch, are not indicated as Instruction Not Executed (IN), but as Wait (WT). Only instructions that reach the Execute stage of the pipeline are traced, with the exception of instructions that are canceled because of an interrupt, prefetch abort, or processor Reset exception (see *Exceptions* on page 4-9).

Every time the processor performs a branch operation (BE or BD), at least two instructions that have been fetched into the pipeline are discarded. However, for the two cycles after a branch, the pipeline status pins are re-used to output an address packet offset that indicates how many branch addresses are currently in the on-chip FIFO. For more information, see *Address Packet Offset* on page 5-12.

The association between trace packets and pipeline status signals is summarized in *Pipeline status and trace packet association in ETMv1* on page 5-8.

5.1.1 Trigger PIPESTAT signals

Rather than having a dedicated pin to indicate a trigger event, a special pipeline status encoding is used.

When a trigger event occurs, the TR (Trigger) pipeline status replaces the current pipeline status and the pipeline status that is replaced is output on the **TRACEPKT[2:0]** pins. The FIFO draining is stopped for that cycle to enable this to happen, and the decompressor must take account of this.

To ensure that trace trigger events can be used to trigger external logic, such as a logic analyzer, it is important that generation of the TR pipeline status is not delayed. The TR pipeline status must be generated as soon as possible after the trigger event goes active. This means that a TR can occur before the instruction that caused it is traced.

Note

Trace discontinuities result in an imprecise the trigger condition. They can be caused by overflow of the FIFO, or **TraceEnable** going inactive. Therefore, decompression of the trace does not associate the trigger with a particular processor cycle. In addition, if an instruction causes the trigger to occur, the trigger status might not be generated on the pipeline status for that instruction.

5.2 ETMv1 trace packets

The **TRACEPKT** pins output packaged address and data information related to the pipeline status. All packets are eight bits in length, irrespective of the number of **TRACEPKT** pins implemented. The trace packets are output on the **TRACEPKT** pins as follows:

Four pins A packet is output over two cycles on **TRACEPKT[3:0]**. In the first cycle packet [3:0] is output and in the second cycle packet [7:4] is output.

Eight pins A packet is output in a single cycle on **TRACEPKT[7:0]**.

Sixteen pins Up to two packets can be output per cycle. If there is only one valid packet, it is output on **TRACEPKT[7:0]**, and **TRACEPKT[15:8]** is UNKNOWN. If there are two packets to output, the first is output on **TRACEPKT[7:0]** and the second on **TRACEPKT[15:8]**.

An additional pin indicates the type of packet being output. **TRACESYNC** is HIGH on the first cycle of a PC address packet sequence. See *Trace synchronization in ETMv1* on page 5-12 for more information.

You must be aware of the rules used when outputting trace packets. These are described in *Rules for generating and analyzing the trace in ETMv1* on page 5-5.

5.3 Rules for generating and analyzing the trace in ETMv1

This section describes the restrictions and requirements that are observed during the generation of trace packets in ETMv1. This information is very important because the software that decompresses the trace must always be able to infer:

- when there is valid data on the **TRACEPKT** pins
- which data packets are associated with particular instructions
- whether there is valid **TRACEPKT** data on cycles with a pipeline status other than WT (Wait).

The rules for trace packet generation in ETMv1 are:

- Packets generated by a particular instruction must form a continuous block in the packet stream.
- Gaps in a trace packet block are permitted only when the normal **PIPESTAT** for a particular cycle is WT. In this case the **PIPESTAT** is changed to be TD (Trace Disabled), indicating that there is no data in the trace packet. This means that you can discard all TD packets unless you are performing cycle-accurate tracing as described in *Cycle-accurate tracing* on page 5-19. When TDs are filtered out, the block of packets appears continuous.
- The group of packets for a particular instruction cannot start on or before any previous functional **PIPESTAT** (IE, IN, ID, BE, or BD). This includes any *Address Packet Offset* (APO) cycles for the instruction (described in *Address Packet Offset* on page 5-12). For example, a BE (Branch Executed) followed by an ID (Instruction Executed with Data) causes the packets to be delayed until after the BE and APOs. If an instruction generates Wait **PIPESTAT**s before generating its functional **PIPESTAT**, the packets for this instruction can begin on any of these Wait cycles (provided that any gaps in the packet stream are indicated by a TD).
- When the FIFO is not empty, packets generated by a particular instruction must be generated immediately after any data already in the FIFO is output. This means that there must be no gaps between the packets.
- When the FIFO is empty, the start of the group of packets for a particular instruction must occur no later than the associated **PIPESTAT**.

Note

If a trigger occurs, all subsequent trace packets are delayed by one cycle. This cycle outputs the replacement pipeline status.

5.3.1 Additional considerations for 16-bit ports

You must treat a 16-bit port slightly differently to deal with the possibility of port transactions that do not use the full port width.

The following rules apply for 16-bit ports:

- a single packet is output only if the **PIPESTAT** is not WT (Wait)
- the first packet of a branch address must always be output on **TRACEPKT[7:0]**.

There are three exceptions to these rules, when it can be guaranteed that the next trace packet has an associated **TRACESYNC**. The exceptions are:

- When the FIFO is draining after an overflow has occurred. The decompressor must be aware that FIFO draining can generate an empty byte. The address packet offset for this BE (Branch Executed) might not be zero if the FIFO has not drained completely when tracing is enabled.
- When the ARM processor is in debug state. This is indicated by **DBGACK** asserted HIGH.
- When the FIFO is draining because tracing has been disabled.

5.3.2 Example ETMv1 trace

Consider the following simple code fragment, where memory location `0x0020000` contains the value `0x44332211`:

```
1000    MOV R2, #20000
1004    LDRB R0, [R2]    ; 0x11
1008    LDRH R0, [R2]    ; 0x2211
1012    LDR  R0, [R2]    ; 0x44332211
1016    NOP
1020    B    1000
```

The execution of this code on an ARM7TDMI processor produces the trace output shown in Example 5-1. Data address tracing is disabled. Significant cycles are labeled in bold, for example, **Branch**.

Example 5-1 Sample ETMv1 trace output

TRACESYNC	PIPESTAT[2:0]	TRACEPKT[7]	TRACEPKT[6:0]	
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	IE	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	ID **LDRB executed**	0	0010001	**Byte data**
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	ID **LDRH executed**	0	0010001	**Halfword data**
0	WT	0	0100010	**Halfword data**
0	TD	0	0000001	
0	TD	0	0000001	
0	WT	0	0010001	**Word data**
0	WT	0	0100010	**Word data**

0	WT	0	0110011	**Word data**
0	WT	0	1000100	**Word data**
0	ID **LDR executed**	0	0000001	**Unused**
0	TD	0	0000001	
0	TD	0	0000001	
0	TD	0	0000001	
0	IE	0	0000001	
1	BE	0	0000000	**Branch**

The trace output shown in Example 5-1 on page 5-6 is analyzed as follows:

- All packets output with TD **PIPESTAT**s can be discarded.
- The byte data must be output on the same cycle as its associated ID (Instruction Executed with Data) **PIPESTAT**, because there are no WT (Wait) cycles between it and the preceding IE (Instruction Executed).
- The packets for the LDRH cannot occur on or before the ID **PIPESTAT** that is associated with the LDRB instruction. Therefore the first byte for the LDRH instruction is output in the same cycle as its associated ID **PIPESTAT**, with the second byte following immediately on a WT cycle.
- The LDRH instruction requires two packets, so the data packets following the second LDRH packet must be associated with the next instruction to be executed. That instruction is an LDR, and requires four packets. These are output on Wait cycles, and then the LDR generates its functional **PIPESTAT** (ID).
- When the functional **PIPESTAT** for the LDR is generated, no associated packet data remains to be output, so **TRACEPKT** is unused in this cycle.

5.4 Pipeline status and trace packet association in ETMv1

Table 5-2 shows the various trace packets that can be associated with different types of pipeline status.

Table 5-2 PIPESTAT and TRACEPKT association

Pipeline status	Associated trace packets
IE (Instruction Executed)	No packets.
ID (Instruction Executed with Data)	Data value only. 1 packet for bytes, 2 packets for halfwords, 4 packets for word transfers. n * 4 packets for load/store multiple operations, where n is the number of registers transferred.
ID (Instruction Executed with Data)	Address only. 1-5 packets containing the address of the transfer. For load/store multiples this is the address of the first transfer in the sequence.
ID (Instruction Executed with Data)	Data value and address. A number of packets for address followed by a number of packets for data. The address packets are the same as for ID, address only. The Data packets are the same as for ID, data value only.
IN (Instruction Not Executed)	No packets.
WT (Wait)	No packets.
BE (Branch Executed)	1-9[a] packets containing the branch destination. The most significant bit of each packet, bit [7], indicates if an additional packet is to follow. When bit [7] is HIGH there is an additional packet, when bit [7] is LOW it is the last packet. For a 5-packet branch destination, bits [6:4] of the final packet are used to indicate the reason for the branch address, for example a trace discontinuity.
BD (Branch Executed with Data)	A number of packets providing the information on the data access, followed by 1-9[a] packets containing the branch destination. The packets holding information on the data access are the same as those described in the ID entries in this table.
TR (Trigger)	The **TRACEPKT[2:0]** pins are used to output the pipeline status that is generated.
TD (Trace Disabled)	No packets. **TRACEPKT[0]** is used to indicate the status of **TraceEnable** when cycle-accurate tracing is enabled.

a. 1-5 packets of address plus (for ETMv1.2 or later only) 1- 4 packets of Context ID.

5.5 Instruction tracing in ETMv1

Instruction trace works by outputting the destination address of branches. This section contains information about instruction tracing that relates specifically to ETMv1. For a more general description of instruction tracing with the ETM, see Chapter 4 *Signal Protocol Overview*.

5.5.1 Direct branches to the exception vector table

In ETMv1, a direct branch to an address in the vector table is traced as an indirect branch.

5.5.2 ARM and Thumb code

Bit [0] of the address indicates whether the destination of the branch is ARM code (bit [0] LOW) or Thumb code (bit [0] HIGH).

Bit [1] of the address is traced as zero for ARM instruction accesses, regardless of the alignment of the instruction address. The decompressor must ignore bit [1] of the address for ARM instruction accesses. This bit of the address is reserved for future expansion.

5.5.3 Java code

When tracing Java code (in ETMv1.3 only), bit [0] indicates bit [0] of the bytecode address. Bit [7] of the fifth address packet is used to indicate a branch to Java code (see *Moving to and from Jazelle state (ETMv1.3 only)* on page 5-10).

5.5.4 Compressed branch address packet structure

When a processor performs a branch operation the destination of the branch is often reasonably close to the current address. The spatial locality of branch destinations provides additional compression of the branch addresses. It is necessary to output only the low order bits that have changed since the last branch. The full address can be reconstructed when decompression of the trace information takes place.

All trace packets are eight bits in length and a branch address can be made up of between one and five packets. The **TRACESYNC** signal indicates the first packet and is asserted HIGH only for the first packet of any branch address.

Each packet of a branch address is structured so that the most significant bit (bit [7]) indicates if there are more address packets. This makes it possible for the decompressor to detect the last packet. If bit [7] is HIGH, another address packet follows. Bit [7] LOW means it is the last address packet.

To decide how many packets are required, the on-chip logic registers the last branch address that it has output, and when another branch occurs, the new address is compared with the one that was previously output. Only sufficient low order bits must be output to cover all the bits that have changed in the address. For example, if the upper 12 bits of the address are unchanged and A[19] is the most significant bit to have changed, then it is only necessary to output A[19:0]. You can do this in three address packets instead of five.

A full 32-bit address is made up of five packets. In ARM and Thumb state, the first four have bit [7] HIGH and the last packet has bit [7] LOW. The address is made up as follows:

Address[6:0] Bits [6:0] of first packet, bit [7] HIGH.

Address[13:7] Bits [6:0] of second packet, bit [7] HIGH.

Address[20:14] Bits [6:0] of third packet, bit [7] HIGH.

Address[27:21] Bits [6:0] of fourth packet, bit [7] HIGH.

Address[31:28] Bits [3:0] of last packet, bit [7] LOW.

This is shown in Figure 5-1.

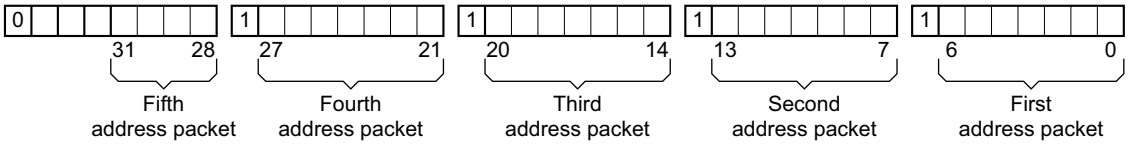


Figure 5-1 Full address output in ARM and Thumb state

Note

When an address is output that is less than 32 bits the new address value replaces the appropriate bits in the previously output branch address. The value does not have to be added to or subtracted from the previous value, nor is it based on the immediately preceding PC value.

Moving to and from Jazelle state (ETMv1.3 only)

When in Jazelle state or moving to and from Jazelle state, bit [7] of the fifth address packet is used as follows:

Bit [7] asserted When branching into Jazelle state.

Bit [7] cleared When branching into ARM/Thumb state.

5.5.5 Branch reason codes

Bits [6:4] of the fifth packet of a full branch address contain a reason code. The reason code indicates why the full branch is been generated. The list of possible codes is shown in Table 5-3.

Table 5-3 Branch reason codes

Bits [6:4]	Description
b000	A normal PC change. Periodic synchronization point, in ETMv1.1 or earlier.
b001	Tracing enabled.
b010	Trace restarted after a FIFO overflow.
b011	The processor has exited from debug state.
b100	Periodic synchronization point, in ETMv1.2 or later.
b101-b111	Reserved for future expansion.

Any reason code other than b000 or b100 indicates a discontinuity in the trace.

5.6 Trace synchronization in ETMv1

For large trace captures the first trace sample is likely to be lost from the TPA (overwritten by a newer trace sample) by the time that the trigger event occurs. For this reason it is necessary to periodically output synchronization information so that the decompression of the trace can be accomplished successfully. The ETMv1 trace synchronization mechanisms are described in the following sections:

- *Address Packet Offset*
- *Full address output*
- *Context ID tracing* on page 5-13.

5.6.1 Address Packet Offset

The *Address Packet Offset* (APO) is used by the decompressor to synchronize between the pipeline status signals (**PIPESTAT**) and the trace packet signals (**TRACEPKT**).

Every time the processor performs a branch operation, at least two instructions that have been fetched into the pipeline are discarded, and the PIPESTAT pins are re-used to output an APO over those two cycles.

The APO indicates the number of addresses that the decompressor must skip, including the cycle on which the branch **PIPESTAT** signal is generated. APOs can take a value of 0-3 on each cycle, so offsets ranging from 0-14 can be output. (The value of 15 is reserved to indicate that the offset is too large to be encoded.) Only two of the three PIPESTAT bits are used to avoid conflicting with BE (Branch Executed) and TR (Trigger), that can occur at any time.

The least significant two bits of the APO are output during the first cycle after a branch. The most significant two bits are output during the second cycle. An offset of 0 specifies that the next **TRACESYNC** in the trace determines the first packet of the address. An offset of 1, for example, indicates that the second **TRACESYNC** in the trace identifies the start of the associated branch packets.

Note

Wait states might result in more than two WT cycles following a branch. No attempt is made to use these additional cycles to output a larger offset.

If a BE is observed in an APO cycle, this indicates that the previous BE has been abandoned. The address for this abandoned branch is still output, and can be ignored.

A TD (Trace Disabled) **PIPESTAT** cannot be observed in an APO cycle.

If a trigger occurs, the TR is output on PIPESTAT[2:0] and the APO is output on the TRACEPKT pins (see *Trigger PIPESTAT signals* on page 5-3).

5.6.2 Full address output

For large trace captures it is necessary to periodically output a full 32-bit address so that the trace can be correctly decompressed.

A cycle counter is implemented. When this counter reaches its maximum count of 1024 a full address is output, for an indirect branch, at the next opportunity, provided that the FIFO can accept the five trace packets required without overflowing. If this is not possible then the branch is treated normally. This process continues for up to 512 cycles until there is enough space in the FIFO to accept a full 32-bit address. The ETM resets the counter whenever a full address is generated.

If there are no indirect branches (such as in a large program loop), or there is insufficient space in the FIFO, a full address is not generated. In this case a full 5-packet address is forced by either:

- Turning the next direct branch into a 5-packet indirect branch. The direct branch is made indirect by generating a BE (Branch Executed) pipeline status.
- Forcing the next indirect branch to be a 5-packet branch, even if this was not scheduled.

Both of these measures can cause an overflow to occur. However, this drawback is a reasonable penalty for ensuring synchronization.

ETMv1.2 or later assigns a reason code of b100 to these full addresses. ETMv1.0 and ETMv1.1 use reason code b000.

5.6.3 Context ID tracing

Context ID tracing is possible only in ETMv1.2 and later.

When a 5-packet address is output that does not have a b000 reason code, a Context ID is traced following the address (provided that ProcIDSize is not b00).

When the Context ID changes, the next branch that occurs (whether direct or indirect) forces the new Context ID to be output.

The packets containing the address and the Context ID must be contiguous, but the continuity bit on the fifth address packet must remain LOW.

5.7 Data tracing in ETMv1

Data trace works by outputting the data accesses (that is address, data value, or both) performed by the processor. This section contains information about data tracing that relates specifically to ETMv1. For a more general description of data tracing with the ETM, see Chapter 4 *Signal Protocol Overview*.

5.7.1 PIPESTAT signals indicating data accesses in the pipeline

The ETM generates specific pipeline status encodings to indicate which load and store instructions caused packets to be inserted into the trace stream. A load/store operation can be signaled in four possible ways, depending on the state of **ViewData** and whether the instruction causes a branch. The possible signals are:

IE (Instruction Executed)

An instruction has executed and one of the following applied:

- the instruction was not a load/store operation
- the instruction was a load/store operation but the data access was not placed into the trace stream because **ViewData** was inactive.

Instruction execute means that the instruction at that address has reached the Execute stage of the pipeline and includes instructions that fail their condition codes. This is slightly different from the pipeline status codes that indicate instruction executed and condition code test passed (these codes have the letter E, standing for *Executed*, in their mnemonics), and instruction executed and condition code failed (these codes have the letter N, standing for *Not Executed*, in their mnemonics).

If the instruction reaches execution but fails its condition code test, a pipeline status code or P-header is generated that includes the letter N in its mnemonic, to indicate an instruction not executed. ETMv1.2 introduced the facility to control trace using the result of the condition code test whenever an instruction is executed.

ID (Instruction Executed with Data)

A load/store instruction has executed causing a data access, and **ViewData** was active when the access occurred.

BD (Branch Executed with Data)

This special case is similar to Instruction Executed with Data, except that the instruction also caused a write to the PC. There are two possible reasons for this:

- there was an explicit load operation to the PC
- the load/store was aborted.

Both of these occurrences cause the processor to branch, and therefore the trace packets must include a branch destination address in addition to any packets associated with the data transfer.

BE (Branch Executed)

An instruction has executed and one of the following applied:

- the instruction was not a load/store operation but caused a write to the PC

- the instruction was a load to the PC and **ViewData** was not active when the access occurred
- an exception occurred.

This **PIPESTAT** signal is used for all indirect branches, see *Instruction trace* on page 4-9 for more details.

5.7.2 Load/Store Multiple instructions

For LSM instructions, **ViewData** is sampled for the first access of the sequence only. This ensures that either none or all of the words transferred are traced. This is necessary for successful decompression, and because the transferred data must be associated with the correct ARM core registers. LSM instructions are listed in *Definitions* on page 4-21.

Note

Non-Wait **PIPESTAT** values (that is, those that indicate an instruction was executed) are always given on the last cycle the instruction is executing. This is important for LSM instructions that execute and return data for several cycles.

5.7.3 Trace packet sequence for data accesses

A data access can cause several different types of trace packet to be inserted into the trace packet stream. The sequence of packet types is always the same, although some packets might not be generated depending on the type of access.

The sequence of trace packet types for data accesses is as follows:

1. Address of data access.

   If the trace port is configured to capture the address of data accesses, this address is the first information to be placed into the trace stream. Between one and five trace packets are required to encode the address, in a similar manner to branch addresses (see *Compressed branch address packet structure* on page 5-9). The address is compressed relative to the last data address output. Bit [7] of each packet indicates whether another address packet is to follow. The reason code is always b000.

   Note

   TRACESYNC is not asserted when the load/store address is output.

2. Data value used in the transfer.

   If the trace port is configured to provide address and data value, or data value only, that data value is the next piece of information to be placed in the trace stream. The number of bytes of data value trace is the same as the number of bytes transferred by the instruction.

3. Branch destination address.

   If the instruction is a load operation with the PC as a destination register, a branch destination address is output last. This is encoded in the same way as all other branch addresses.

TRACESYNC is asserted when the first of the instruction address packets is output, in the same way as for other branches.

5.7.4 Data aborts

If one or more of the data accesses was aborted by the memory system, a PC address is also output as part of the same instruction. See *Full address output* on page 5-12 for details.

A data abort can occur on any or all of the data transferred. Data tracing ignores the abort status of data transferred. All transferred data for an instruction, whether aborted or not, is traced.

The pipeline status for the aborted instruction is *branch executed* or *branch with data* depending on whether there is any traced data associated with the instruction.

5.7.5 Address compression performed by the ETM

The ETM compresses the data trace by reducing the number of bits that are output for the address of the data transfer. The same technique is used as for branch addresses, where a copy of the last data access address is kept and only the low order bits that have changed are output for the next address.

This is particularly effective, for example, if you are viewing data in one small address range, because all the traced data accesses have the same high-order address bits.

When the address of a data access output it is compressed only if both of the following conditions are satisfied:

- A full 32-bit data address has been output in the synchronization period that ends with the current cycle. The synchronization period is set in the Synchronization Frequency Register, `0x78`, if present, and otherwise is 1024 cycles.
- There has been no interruption in tracing.

Otherwise no compression takes place, and the full 32-bit address is used to ensure that the captured trace information contains a reference point for the other compressed address packets.

5.8 Filtering the ETMv1 trace

The ETM has a **TraceEnable** function that you can use to enable or disable tracing during a trace run. This signal is typically used to select the areas of code that are traced and to disable the trace when code is executed that is of limited use to the debugging process. The advantage of disabling the trace information is that it effectively increases the amount of useful information that can be captured by a given size of buffer in the TPA, enabling selective tracing over a longer time period.

5.8.1 Enabling trace

When **TraceEnable** becomes active during a trace run, tracing is enabled. The trace port outputs a BE status, and associated with it is a 5-packet address. This provides a start address so that the trace information can be successfully decompressed from the first instruction after the trace is enabled. This indicates to the trace decompressor software that there is a discontinuity in the trace. A correct address packet offset must be generated, because the FIFO might not be empty at the point when tracing has been enabled.

In ETMv1.3, bit [7] of the fifth address packet indicates the state of the J-bit for the instruction where tracing is enabled.

5.8.2 Disabling trace

When **TraceEnable** becomes inactive, tracing stops and the pipeline status changes to WT (Wait) while the FIFO drains. When there is no more data in the FIFO the pipeline status changes to TD (Trace Disabled). This enables the TPA to suppress tracing, and so improve the trace buffer utilization.

Note

In cycle-accurate tracing, TD cycles can correspond to WT cycles. In this case the TPA might still have to capture these cycles. For more information see *Cycle-accurate tracing* on page 5-19.

5.8.3 Data accesses during disabled trace

When the trace port is disabled you cannot view data accesses and the **ViewData** output is ignored.

5.8.4 Precise events

Data filtering is not always precise and it is sometimes not possible to trace the event that is used to control data filters. For more information, see *Imprecise TraceEnable events* on page 2-22 and *Imprecise ViewData events* on page 2-28.

5.9 FIFO overflow

Under certain circumstances it is possible that so much trace information is generated on-chip that the FIFO can overflow. When this occurs a two-stage process to empty the FIFO and restart the trace takes place:

1. First the pipeline status is changed to WT (Wait) and emptying of the FIFO is enabled. This ensures that all trace information up to the overflow condition is collected. This information might be useful in determining the cause of the FIFO overflow. If overflow occurs part way through a load or store instruction the pipeline status for the instruction must be generated. This is to enable the decompression software to associate any trace packets with an instruction.
2. When the FIFO has been drained, tracing must be re-enabled as soon as possible, if **TraceEnable** is still active.

Note

A trace decompressor must be able to deal with the loss of some or all of the packets for an instruction. This lost information might include data, address, and Context ID packets.

The trace decompressor can successfully decode a FIFO overflow sequence, if it is aware that a BE (Branch Executed) marked as *FIFO overflow* is speculative and might be followed by another BE also marked as *FIFO overflow*. If this is the case, then the second BE marked as *FIFO overflow* can overwrite the Address Packet Offset of the speculative BE. This is not a problem, because the speculative BE has been found to be incorrect, and therefore can be discarded.

In ETMv1.1 and later, an ETM Status Register is provided, see *ETM Status Register, ETMv1.1 and later* on page 3-33. Bit [0] of this register is a pending overflow flag, indicating that an overflow has occurred but no *overflow occurred* reason code been generated. This indication is required where tracing stops because of an ARM breakpoint. The pending overflow flag remains set until tracing is restarted and the overflow can be traced.

5.9.1 System stalling

To prevent loss of trace data, it is recommended that your system recognizes when the FIFO is about to overflow. An on-chip **FIFOFULL** output is provided that indicates when the FIFO has less than a configured number of bytes of space available. You can use this signal to stall the ARM processor to prevent the FIFO from overflowing. The assertion of this signal is controlled by configurable instruction address regions to prevent system stalling in critical code. See *Processor stalling, FIFOFULL* on page 2-31 for details.

5.10 Cycle-accurate tracing

When profiling the execution of critical code sequences, it is often useful if you can observe the exact number of cycles that a particular code sequence takes to execute. To perform this *cycle-accurate tracing*, you must set bit [12] of the ETM Control Register to 1 (see *ETM Control Register* on page 3-20).

When cycle-accurate tracing is enabled, **TRACEPKT[0]** is HIGH when **PIPESTAT** has the value `0x7` (TD). This causes the TCD to capture trace on all cycles, even if there is no trace to output on that cycle. The number of cycles taken by a region of code can therefore be determined by counting the number of cycles of trace captured.

Cycle-accurate tracing is disabled when:

- tracing is disabled, that is, when **TraceEnable** is inactive or prior to restarting following FIFO overflow.
- the ARM processor enters debug state.

5.11 Tracing Java code, ETMv1.3 only

Support for Jazelle state tracing is included in ETMv1.3 or later. A branch into or out of Jazelle state forces a full 5-packet instruction address to be generated. Bit [7] of the fifth address packet indicates the new state of the J-bit. When tracing Java bytes, one or more pipeline status messages might be generated for a particular bytecode, based on the number of interesting loads and stores that occur. Exactly the same number of pipeline status messages is generated if the bytecode is retraced.

TraceEnable is sampled on the first interesting load or store if appropriate, otherwise it is sampled at the end of the bytecode. If **TraceEnable** is not asserted at this point, tracing is disabled.

ViewData is sampled on the first interesting load or store, and all remaining loads or stores for that bytecode are traced. Each bytecode might therefore result in one or two traced instructions.

Decompression of Java trace in ETMv1 requires a knowledge of the Jazelle architecture, that is confidential. If you want to decompress Java trace, contact ARM Limited for more information.

Chapter 6
ETMv2 Signal Protocol

This chapter describes the signals that are output from the trace port. It contains the following sections:

6.1 ETMv2 pipeline status signals

You can consider the pipeline status as a view of the Execute stage of the pipeline. Any side-effects of the instruction, for example an aborted load or a write to the PC, are observed immediately, before the pipeline status for the next instruction is generated. This means that the protocol is independent of the exact pipeline implementation of the ARM core.

Each executed instruction generates a single pipeline status message on the **PIPESTAT** signals of the port. These are shown in Table 6-1.

Table 6-1 PIPESTAT messages

Status	Mnemonic	Meaning	Description
b0000	IE	Instruction Executed	An instruction passed its condition code test and was executed. No trace packets were generated. Usually this status is generated by an operation that did not cause a branch. It is also generated by direct branches, that is, branches where the destination can be calculated by referring back to the code image.
b0001	DE	Instruction Executed with Data	An instruction passed its condition code test and was executed. The instruction placed one or more packets on the FIFO. The system outputs these packets from the FIFO as a **TRACEPKT** signal.
b0010	IN	Instruction Not Executed	An instruction reached the Execute stage of the pipeline, but failed its condition code test. No trace packets were generated.
b0011	DN	Instruction Not Executed with Data	An instruction reached the Execute stage of the pipeline, but failed its condition code test. The instruction placed one or more packets on the FIFO. The system outputs these packets from the FIFO as a **TRACEPKT** signal.
b0100	WT	Wait	No instruction this cycle, but there is valid data on the trace port. If there is no packet output in a cycle, the status returned is TD, not WT.
b0101	DW	Wait with Data	No instruction this cycle, however packets have been placed on the FIFO. The system outputs these packets from the FIFO as a **TRACEPKT** signal.
b0110	TR	Trigger	Indicates that the trigger condition occurred, see *Trigger PIPESTAT signals* on page 6-5. The real pipeline status value is on **TRACEPKT[3:0]**.

Table 6-1 PIPESTAT messages (continued)

Status	Mnemonic	Meaning	Description
b0111	TD	Trace Disabled	There is no FIFO data on the TRACEPKT pins this cycle. There are two possible reasons for this: • The FIFO is empty. This is likely to occur after tracing is disabled until it is next enabled. • A TFO is being output for ETM synchronization. See *Trace synchronization in ETMv2* on page 6-14 for details.
b1000 b1001 b1010 b1011	PTIE PTDE PTIN PTDN	Branch phantom taken plus IE Branch phantom taken plus DE Branch phantom taken plus IN Branch phantom taken plus DN	A branch was correctly predicted, taken, and executed in parallel with the instruction following, that caused an IE, DE, IN, or DN. See *Branch phantom PIPESTAT signals* on page 6-4.
b1100 b1101 b1110 b1111	PNIE PNDE PNIN PNDN	Branch phantom not taken plus IE Branch phantom not taken plus DE Branch phantom not taken plus IN Branch phantom not taken plus DN	A branch was correctly predicted, not taken (because it failed its condition codes), and executed in parallel with the instruction following, that caused an IE, DE, IN, or DN. See *Branch phantom PIPESTAT signals* on page 6-4.

Only instructions that reach the Execute stage of the pipeline are traced, except for instructions cancelled by certain exceptions, see *Exceptions* on page 4-9 and *Instruction Not Executed PIPESTAT signals* on page 6-4.

Note

ETMv2 protocol differs from the ETMv1 protocol because ETMv2 describes what happens each cycle. With ETMv1, ID **PIPESTAT** only occurs on a Data instruction. With ETMv2, DE **PIPESTAT** can occur on any instruction (for example `MOV r1,r2`) if no data is loaded or stored in parallel with the instruction being executed.

6.1.1 Wait PIPESTAT signals

WT (Wait) **PIPESTAT** signals are generated for several reasons, including:

- an instruction was fetched but is not executed because of a branch

- a wait signal from the memory system is asserted
- a multi-cycle instruction is executing.

6.1.2 Branch phantom PIPESTAT signals

On some processors, for example the ARM10 processor, a branch can be predicted, pulled out of the normal instruction stream, and effectively executed in parallel with the next instruction in the program. This is known as *branch folding* and the folded branches are referred to as *branch phantoms*.

The branch phantom **PIPESTAT** encodings are used to identify these instances of parallel instruction execution. The branch instruction is always first in the execution stream. Only direct branches are predicted, so branch phantoms never place data packets on the FIFO. They can be interpreted as an IE (Instruction Executed) or IN (Instruction Not Executed) **PIPESTAT** (depending on whether or not the branch was taken) followed by an IE, DE, IN or DN **PIPESTAT** as appropriate.

Folded branches that are mispredicted result in IE or IN **PIPESTAT** signals. This is because any instruction that might be executed in parallel with a mispredicted branch is from the wrong instruction stream and is canceled.

6.1.3 Data PIPESTAT signals

Where a **PIPESTAT** mnemonic contains the letter D, it means that a data packet of some sort was placed on the FIFO that cycle. Subsequently, the system outputs these packets from the FIFO on the TRACEPKT pins.

With the introduction of DW (Wait with Data) and DN (Instruction Not Executed with Data) **PIPESTAT**s, data packets can be associated with any cycle. (DW and DN **PIPESTAT** signals are not present in ETMv1.)

6.1.4 Instruction Executed PIPESTAT signals

PIPESTAT values that indicate an instruction was executed are always given on the first cycle the instruction is executing. This is important for LSM instructions that execute and return data for several cycles. (In ETMv1, **PIPESTAT** values are output on the last cycle the instruction is executing.)

6.1.5 Instruction Not Executed PIPESTAT signals

PIPESTAT values indicating that an instruction failed its condition code test (containing mnemonics IN (Instruction Not Executed) or DN (Instruction Not Executed with Data) can occur for two reasons:

- the instruction failed its condition codes
- the instruction was not executed because an exception occurred. If this is the case, the ETM traces the instruction as having either passed or failed its condition codes (mnemonics IE, DE, IN or DN), and a branch address packet follows indicating the exception. The decompressor must ignore the condition code information.

  Possible exceptions that might be given an IN or DN status are:
  — interrupts

— prefetch aborts
— processor reset assertion.

Load/store instructions that result in data aborts are not given an IN or DN status because they are considered to have executed.

6.1.6 TD PIPESTAT signals

A pipeline status of TD (Trace Disabled) means that trace FIFO data is not present on the **TRACEPKT** pins this cycle. There are two possible reasons for this:

There is no data to be traced in the FIFO

If the FIFO is not empty, the status is WT (Wait).

The trace output is a *Trace FIFO Offset* (TFO), for ETM synchronization

The decompression software must inspect the **TRACEPKT** value to determine whether the output is a *Trace FIFO Offset* (TFO). If **TRACEPKT[0]** is asserted HIGH, **TRACEPKT[3:1]**is used for TFO outputs, as described in *Trace FIFO offsets* on page 6-14.

TCDs can discard TD cycles where **TRACEPKT[0]** = 0. **TRACEPKT[0]** is used to differentiate between cycle-accurate and non-cycle-accurate tracing. For more information, see *Cycle-accurate tracing* on page 5-19.

6.1.7 Trigger PIPESTAT signals

Rather than having a dedicated pin to indicate a trigger event, a special pipeline status encoding is used.

When a trigger event occurs, the TR pipeline status replaces the current pipeline status and the pipeline status that is replaced is output on the **TRACEPKT[3:0]** pins. The FIFO draining is stopped for that cycle to enable this to happen, and the decompressor must take account of this.

To ensure that trace trigger events can be used to trigger external logic, such as a logic analyzer, it is important that generation of the TR pipeline status is not delayed. The TR pipeline status must be generated as soon as possible after the trigger event goes active. This means it is possible for a TR to occur before the instruction that caused it is traced.

If a trigger occurs when the FIFO is empty, the replaced **PIPESTAT** value that appears on **TRACEPKT[3:0]** is WT (Wait).

If a trigger and a TFO are pending at the same time, the replaced **PIPESTAT** value that appears on **TRACEPKT[3:0]** is TD. This is uniquely identifiable as a true TFO because a WT is never converted to a TD (Trace Disabled) when a trigger occurs.

Triggers are never delayed and are guaranteed to be output immediately when generated. If a trigger is pending in the second cycle of a TFO output (or the gap cycle) from a 4-bit port, the trigger occurs and the FIFO output is delayed by an extra cycle to output the remaining TFO nibble(s). See *Trigger considerations* on page 6-16 for more details.

6.2 ETMv2 trace packets

The **TRACEPKT** pins output all trace information that is not encoded by the **PIPESTAT** pins. This information is organized into packets, each of one or more bytes in length.

There are three possible scenarios for information output on the **TRACEPKT** pins, depending on the number of pins in use:

Four pins A byte is output over two cycles on **TRACEPKT[3:0]**. In the first cycle bits [3:0] are output and in the second cycle bits [7:4] are output.

Eight pins A byte is output in a single cycle on **TRACEPKT[7:0]**.

Sixteen pins

Up to two bytes can be output per cycle. If there is only one valid byte, it is output on **TRACEPKT[7:0]**, and the signal on **TRACEPKT[15:8]** is `0x66`. If there are two bytes to output, the first is output on **TRACEPKT[7:0]** and the second on **TRACEPKT[15:8]**.

The **TRACEPKT** pins are used to output the following types of information:

Trace packets

Trace packets are output when the ETM is collecting trace data.

Note

- In this document, a *packet* is a discrete quantity of trace information comprising one or more bytes. In previous versions of this document, the word packet and byte were used interchangeably.
- Multiple packets can be placed in the FIFO in one cycle. Each packet begins with a header that indicates whether or not more packets follow. The only exception to this is branch addresses. A branch address is always the last packet to be placed in the FIFO in any cycle.
- Types of data packet include the following:
  - a branch address
  - a normal data packet
  - a Context ID packet.
- Each packet is identified by a packet header, see *Trace packet headers* on page 6-8.

Trace FIFO Offset (TFO) and TFO packets

TFO packets are occasionally output as part of the trace synchronization mechanism. Trace synchronization is described in *Trace synchronization in ETMv2* on page 6-14.

Trigger information

When a TR (Trigger) pipeline status occurs, the pipeline status that it replaces is output on the **TRACEPKT[3:0]** pins. For more information, see *Trigger PIPESTAT signals* on page 6-5.

6.3 Rules for generating and analyzing the trace in ETMv2

This section describes the restrictions and requirements that are observed during the generation of trace packets in ETMv2. This information is very important because the software that decompresses the trace must always be able to infer:

- when there is valid data on the **TRACEPKT** pins
- which data packets are associated with particular instructions
- whether there is valid **TRACEPKT** data on cycles with a pipeline status other than WT (Wait).

The rules for trace packet generation in ETMv2 are:

- If a **PIPESTAT** is encountered whose mnemonic contains the letter D, one or more data packets have been placed in the FIFO in that cycle.
- The first byte of each packet indicates the packet type. All packets other than branch address packets indicate whether another packet follows from the same cycle. This enables a group of packets corresponding to a **PIPESTAT** to be identified, possibly ending with a branch address packet.
- No gaps in the trace stream can occur in or between a group of packets from the same cycle.
- When the FIFO is empty, a group of packets corresponding to a **PIPESTAT** must be output starting at the same time as the **PIPESTAT**.
- When the FIFO is not empty, packets must be output immediately after any data already in the FIFO is output. This means there is no gap between the two groups of packets.

No special considerations are required for 16-bit ports. Packets are always output as soon as possible.

6.4 Trace packet types

Trace packets are placed in the FIFO because of any **PIPESTAT** value that includes a D in its encoding. Trace packets can be any of the following types:

Branch Address packet

Used for destination addresses that cannot be directly inferred from the source code. If a branch address is output, it is always placed on the FIFO last in the cycle. For more details, see *Branch Address trace packets* on page 6-22.

Normal Data packet

Used for the following:

- store data packets
- all loads that do not miss in the cache
- CPRT data packets.

For more details, see *Normal Data packets* on page 6-10.

Load Miss packet

Used for load requests that miss in the data cache. For more details, see *Load Miss packets* on page 6-11.

Value Not Traced packet

Used when performing partial tracing of an LSM. For more details, see *Value Not Traced packets* on page 6-12.

Context ID packet

Used when a new Context ID value is output. For more details, see *Context ID tracing* on page 6-20.

6.4.1 Trace packet headers

The trace packet header indicates the type of trace packet being output on the **TRACEPKT** pins, and specifies how to interpret the subsequent bytes of the packet. Trace packet header encodings are shown in Table 6-2.

Table 6-2 Trace packet header encodings

Value	Description
bC0A0SS10	Normal data, (address expected). See *Normal Data packets* on page 6-10.
bX0X1XX00	Reserved.
bX0X1XX10	Reserved.
bX10XXX10	Reserved.

Table 6-2 Trace packet header encodings (continued)

Value	Description
bX1100X10	Reserved.
bC1101010	Value Not Traced. See *Value Not Traced packets* on page 6-12.
bC1101110	Context ID. See *Context ID tracing* on page 6-20.
bX111XX10	Reserved.
bC1A1TT00	Load Miss occurred. See *Load Miss packets* on page 6-11.
bCTT0SS00	Load Miss data. See *Load Miss packets* on page 6-11.

Note

Branch addresses are encoded cXXXXXX1, where c is the address continue bit. Branch addresses are always output last in a cycle and are not preceded by a header. See *Branch Address trace packets* on page 6-22 for more information.

Certain bits in the trace packet header encodings shown in Table 6-2 on page 6-8 have specific functions, as follows:

C
: Informs the decompression tool how many packets are placed on the FIFO in a single cycle. The C bit is set to 1 in every packet except the last packet placed in the FIFO in a cycle. This enables the decompressor to correctly associate packets with cycles (and therefore with instructions).

  All headers other than Branch Address have a C bit. A Branch Address packet is always the last packet to be placed in the FIFO in a cycle, and does not require a C bit.

X
: Indicates that the value is UNDEFINED.

A
: Specifies that this is the first data packet for a particular instruction, and that a data address is expected (if address tracing is enabled). This information enables the decompressor to maintain synchronization when tracing through sections of code that cannot be decompressed (any region for which a binary is not available). The A bit is not asserted on *Coprocessor Register Transfer* (CPRT) packets.

TT
: Used as a tag to identify each load miss. For more details, see *Load Miss packets* on page 6-11.

SS Used for data value compression. The SS bits specify the size of the data value transferred. Leading zeros are removed from the value as a simple form of data compression. Typically this compression technique is enough to offset the additional bandwidth cost of the header byte. The encodings for the SS bits are given in Table 6-3.

Table 6-3 SS bit encodings

Encoding	Description
b00	Value = 0, no data bytes follow
b01	Value < 256, one data byte follows
b10	Value < 65536, two data bytes follow
b11	No compression done, four data bytes follow

6.4.2 Normal Data packets

The Normal Data packet header is used for:

- all loads that do not miss in the cache
- store data packets
- CPRT data packets if CPRT data tracing is enabled.

A Normal Data packet comprises the following contiguous components:

Normal Data packet header

Output first. Always present.

Data address Present if both of the following conditions are satisfied:

- data address tracing is enabled in the ETM Control Register
- the A bit in the header is set to 1.

Data addresses consist of one to five bytes. To enable the decompressor to detect the last byte, bit [7] of each byte is set to 1 if there are more address bytes to follow. Bit [7] is LOW in the last address byte. Whether or not data addresses are traced must be statically determined before tracing begins.

Data value Present only if data value tracing is enabled in the ETM Control Register.

Normal Data packets correspond to the most recently-traced *data instruction*. This is to support cores where instructions that do not perform a data transfer might execute before a previous transfer completes.

64-bit data transfers

When data for LSM instructions is output, the data address is output with the first data packet only.

6.4.3 Load Miss packets

ETMv2 supports cores with nonblocking data caches. A nonblocking data cache enables instructions, including memory instructions, to execute underneath a single outstanding miss. This means that the data cache can return data to the core out of order.

Load requests that miss in the data cache are handled by the out-of-order placeholder and out-of-order data header types.

Load Miss Occurred

When a load miss occurs, an out-of-order placeholder packet is placed in the FIFO instead of a normal data packet. The packet includes the data address if both of the following conditions are satisfied:

- Data address tracing is enabled
- The miss does not occur in the middle of an LSM that has already output data packets. This can be determined from the A bit.

Otherwise, the packet comprises only the out-of-order placeholder header byte.

When an out-of-order placeholder packet is read, the corresponding data packet is output later in the trace. Decompression software must be able to identify and correctly process this situation.

Load Miss Data

When load miss data is returned, the Load Miss Data packet, comprising the Load Miss Data header byte and the data value, is placed in the FIFO.

A Load Miss Data packet never includes a data address.

Out-of-order miss data

Some processors might return miss data out of order. A Load Miss Data packet always corresponds to the most recent Load Miss Occurred packet with the same TT tag value.

If the decompressor receives an unexpected Load Miss Data packet (that is, a Load Miss Data packet is given without a pending Load Miss Occurred packet with the same TT tag), it must be ignored. If trace is disabled before the outstanding miss data is returned, this data item is placed in the FIFO with a DW (Wait with Data) **PIPESTAT** as soon as it is available.

Rules for generation of Load Miss trace packets

These rules do not affect decompression, but describe how load misses are handled by the ETM.

Load Miss Occurred packets are placed in the FIFO if **TraceEnable** and **ViewData** are active at the time of the load miss, in the same way as Normal Data packets.

Load Miss Data packets are placed in the FIFO if and only if the corresponding out-of-order placeholder packet was traced, with the following exceptions:

- Load Miss data might be missing following overflow, if the Load Miss Occurred packet was placed in the FIFO before the overflow occurred.
- Load Miss data might be missing following restart from debug, if the Load Miss Occurred packet was placed in the FIFO before the entry to debug state.
- Load Miss data might be missing following a processor reset, if the Load Miss Occurred packet was placed in the FIFO before the reset occurred.
- Load Miss data might be missing if it is returned in the same cycle as a Non-periodic TFO. This is because of FIFO bandwidth limitations. A periodic TFO must be delayed if it would cause the loss of an out-of-order data packet.

A Load Miss Data packet is never output without a corresponding Load Miss Occurred packet. However, unpaired Load Miss Data packets might be observed at the beginning of the captured trace, if the original Load Miss Occurred packets have been lost.

64-bit loads

When a miss occurs on a 64-bit load value, two Load Miss packets are placed in the FIFO in the same cycle. The decompressor must recognize that these two misses are for a single 64-bit value because both packets have the same tag value and they are consecutive. As with Normal Data packets, the data address is present only with the first Load Miss Occurred packet, and is not present at all if the miss occurs in the middle of an `LSM` that has already output data packets.

When 64-bit Load Miss data is returned, it is always returned as two separate Load Miss Data packets given in the same cycle. Both packets have the same miss tag, and are consecutive.

Note

It is possible to detect 64-bit Load Miss packets by checking for two packets with the same tag on the same cycle. However, to ensure compatibility with the mechanism used in ETMv3, it is recommended that the decompressor must check for two consecutive packets with the same tag value.

6.4.4 Value Not Traced packets

It is possible for a compiler to combine adjacent `LDR` or `STR` operations into an `LSM` without your knowledge. In ETMv1, data tracing can be enabled only at the beginning of a *Load/Store Multiple* (`LSM`) instruction. ETMv2 can partially trace an `LSM` and output only the data values that match the trigger criteria.

When the first data value associated with an LSM is traced, a Normal Data packet is placed in the FIFO containing the data address (if address tracing is enabled) and the data value (if data value tracing is enabled). All subsequent data transfers for that LSM place a packet in the FIFO according to the following rules:

- If a subsequent value is to be traced, a Normal Data packet containing the header and the data value is traced.
- If a subsequent data transfer is not to be traced, a Value Not Traced packet is placed in the FIFO for that transfer.

Value Not Traced packets comprise only a Value Not Traced header byte. The decompression software must work backwards from the final data transfer, using the Value Not Traced packets in combination with the normal data packets, to determine which of the LSM values were traced.

Note

If tracing begins on a LSM instruction, it continues until the LSM completes, even if **TraceEnable** is deasserted before the instruction completes. This means that instructions executed under an LSM are also traced, regardless of whether **TraceEnable** remains asserted (see *Independent load/store unit* on page 2-71).

6.4.5 Context ID packets

When the Context ID changes, a Context ID is output to give the new value. It comprises the following components:

- Context ID packet header (1 byte)
- Context ID (1-4 bytes).

The number of bytes output depends on the ProcIDSize bits (bits [15:14]) of the ETM Control Register, see *ETM Control Register* on page 3-20. If Context ID tracing is disabled by setting these bits to b00, Context ID packets are never generated.

If the Context ID is changed by a data transfer that would normally have been traced, and a Context ID packet is output, it is IMPLEMENTATION SPECIFIC whether the Context ID packet is generated instead of or in addition to the normal data trace.

This means that, when Context ID tracing is enabled, data trace might be missing for an instruction that changes the Context ID.

6.5 Trace synchronization in ETMv2

For large trace captures it is likely that the first trace sample is lost from the TPA (overwritten by a newer trace sample) by the time that the trigger event occurs. For this reason it is necessary to periodically output synchronization information so that the decompression of the trace can be accomplished successfully. The ETMv2 trace synchronization mechanisms are described in the following sections:

- *Trace FIFO offsets*
- *Data address synchronization* on page 6-20
- *Context ID tracing* on page 6-20.

6.5.1 Trace FIFO offsets

ETMv2 generates *Trace FIFO Offsets* (TFO) to enable the decompressor to synchronize the pipeline status (**PIPESTAT**) and FIFO output (**TRACEPKT**) signals.

There are two reasons for generating a TFO:

- trace is first enabled
- periodic synchronization.

Periodic synchronization occurs as soon as possible after the synchronization counter reaches zero, when the current **PIPESTAT** is IE (Instruction Executed).

When the synchronization counter reaches zero it sets an internal flag to indicate that a periodic TFO is required, and immediately resets. If a periodic TFO does occur before the counter next reaches zero, the ETM must output a TFO with reason code 2, overflow. Some trace is lost as a result of this. This condition is very unusual and usually indicates that the core is in an infinite loop.

When a TFO is generated, the following occur in that cycle:

- A **PIPESTAT** of TD is output on **PIPESTAT[3:0]**:
  - If the TFO occurs when trace is turned on, no functional **PIPESTAT** is implied and the **PIPESTAT** for the first traced instruction is given in the following cycle. The header of a TFO caused by turning trace on includes a reason code of b01, b10, or b11.
  - If the TFO occurs for normal synchronization while trace is already enabled, an existing **PIPESTAT** of IE is implied. In this case the TFO header includes the reason code b00.
- The value of the TFO is output on the **TRACEPKT** pins. The TFO value represents a count of the number of bytes in the FIFO, and indicates where the TFO packet can be found on **TRACEPKT**. For more information see *TFO values* on page 6-15.
- Several bytes of data, known as a *TFO packet*, are placed in the FIFO. The TFO packet eventually appears on the **TRACEPKT** pins. For more details of TFO packets, see *General TFO packet structure* on page 6-16.

TFO values

TCDs can discard TD (Trace Disabled) cycles where **TRACEPKT[0]** = 0. If **TRACEPKT[0]** is asserted, the TFO value is output on **TRACEPKT[7:1]** (lower bits on **TRACEPKT[7:4]**). The range of TFO encodings is shown in Table 6-4.

Table 6-4 TFO encodings

TRACEPKT[3:0]	Description
bXXXXXXX0	Trace disabled, not cycle-accurate
bXXXX0111	Trace disabled, cycle-accurate
bXXXX1001	TFO value 0-15 (**TRACEPKT[7:4]** + 0)
bXXXX1011	TFO value 16-31 (**TRACEPKT[7:4]** + 16)
bXXXX1101	TFO value 32-47 (**TRACEPKT[7:4]** + 32)
bXXXX1111	TFO value 48-63 (**TRACEPKT[7:4]** + 48)
bXXXX0001	TFO value 64-79 (**TRACEPKT[7:4]** + 64)
bXXXX0011	TFO value 80-95 (**TRACEPKT[7:4]** + 80)
bXXXX0101	Reserved

TFO formula

The following formula generates the TFO values in Table 6-4:

- **TRACEPKT[7:4]** = TFO[3:0]
- **TRACEPKT[3]** = !TFO[6]
- **TRACEPKT[2:1]** = TFO[5:4].

Example 6-1 shows how to calculate the value of a TFO.

Example 6-1 Calculating a TFO value

Suppose that there is one byte left in the FIFO before the TFO packet header (that is, the TFO value is b0001). Using the TFO formula, the mapping of **TRACEPKT[7:1]** to TFO is:

```
TRACEPKT  7 6 5 4  3 2 1
TFO       3 2 1 0 !6 5 4
```

So a TFO value of b0000001 is output on the **TRACEPKT[7:1]** pins as 0001100. **TRACEPKT[0]** must be asserted, so the full **TRACEPKT[7:0]** output is 00011001.

This example TFO value is used in *Example signal sequence for a mid-byte TFO* on page 6-16.

General TFO packet structure

A TFO packet typically consists of:

- a TFO header byte, see *TFO packet headers* on page 6-17
- a full instruction address, see *Instruction tracing with ETMv2* on page 6-22
- the current Context ID, see *Context ID tracing* on page 6-20.

Trigger considerations

When using a 4-bit port, the lower 4 bits are output on **TRACEPKT** on the cycle of the TD (Trace Disabled), and the upper 4 bits on the following cycle. The pipeline status is TD for the first cycle only.

If a trigger occurs on the same cycle as a TFO TD cycle, the **PIPESTAT** is TR (Trigger), and **TRACEPKT[3:0]** = 0111 (TD). This is the only way a trigger can have a replacement **PIPESTAT** of TD. If a trigger occurs when the **PIPESTAT** would have been a non-TFO TD, the replacement **PIPESTAT** is WT (Wait). The offset is output on the following cycle, or the following two cycles in the case of a 4-bit port.

If a trigger occurs on the cycle in which the upper 4 bits of the offset are being output on a 4-bit port, the replacement **PIPESTAT** is output, and the upper 4 bits of the offset are output on the following cycle.

If a trigger occurs on the same cycle as a *gap nibble* following a TFO on a 4-bit port, the replacement **PIPESTAT** is output and the gap nibble is output on the following cycle. *Mid-byte TFO outputs* describes the gap nibble.

Note

The trigger can occur only once per trace run.

Mid-byte TFO outputs

If a TFO occurs mid-byte in the 4-bit trace packet port configuration, a gap nibble is inserted in the **TRACEPKT[3:0]** output stream. TFO values specify synchronization in terms of bytes rather than nibbles. The gap nibble ensures that the current top of the FIFO, pointed to by the TFO value, is always byte-aligned.

The value of the gap nibble is always `0x6`.

The example sequence in Table 6-5 shows how the **PIPESTAT[3:0]** and **TRACEPKT[3:0]** signals change when a TFO occurs between data nibbles.

Table 6-5 Example signal sequence for a mid-byte TFO

Trace operation	PIPESTAT[3:0]	TRACEPKT[3:0]
Data `0xABCD` is output to **TRACEPKT[3:0]**	ID	b1101 (data nibble `0x---D`)
	IE	`b1100` (data nibble `0x--C-`)
	WT	`b1011` (data nibble `0x-B--`)

Table 6-5 Example signal sequence for a mid-byte TFO (continued)

Trace operation	PIPESTAT[3:0]	TRACEPKT[3:0]
TFO occurs, TFO value output begins	TD (originally IE)	b1001
	IE	b0001
Gap nibble is inserted immediately following TFO value	WT	b0110
Remaining data nibble is output	IN	b1010 (data nibble 0xA---)
TFO packet output begins	DN	TFO header [3:0]

The TFO value output indicates one byte remaining. That byte is made up of the gap nibble followed by nibble 0xA---.

FIFO output is delayed until the complete TFO value (and extra nibble, if required) have been output on **TRACEPKT[3:0]**.

Note

In cases where synchronization is not required, the decompressor must be aware that the gap nibble appears on **TRACEPKT[3:0]**. The decompressor must always expect this extra nibble when a TFO is generated on an odd nibble regardless of whether the TFO is because of synchronization or because trace has been enabled.

6.5.2 TFO packet types

The following types of TFO packet can be output during trace synchronization:

- Normal TFO packet (see *Normal TFO packets* on page 6-18)
- LSM In Progress TFO packet (see *LSM In Progress TFO packets* on page 6-18).

6.5.3 TFO packet headers

TFO packets are placed in the FIFO by a TFO cycle. The decompressor knows when a packet is placed in the FIFO by a TFO, so TFO packets have their own header byte encodings, as shown in Table 6-6.

Table 6-6 TFO packet header encodings

Value	Description
b0RR00000[a]	Normal TFO packet
b1RR00000a	LSM In Progress TFO packet

a. RR represents the TFO reason code.

All other encodings are reserved. The TFO packet header encodings are completely independent of the encoding space used by trace data packets.

6.5.4 Normal TFO packets

A normal TFO packet comprises the following contiguous components:

A header byte Broadcast first. The TFO header byte includes the 2-bit reason code that is labeled as RR in Table 6-6 on page 6-17). For more information about TFO reason codes, see *TFO reason codes*.

Context ID The number of Context ID bytes traced (0 to 4) is statically determined by ETM Control Register bits [15:14]. For more information on Context ID, see *Context ID tracing* on page 6-20.

Instruction address

The instruction address is always four bytes and is not compressed. Bit [0] specifies the Thumb bit.

TFO reason codes

The TFO reason codes are consistent with the branch reason codes used in ETMv1.0 and ETMv1.1. Table 6-7 shows the TFO reason codes.

Table 6-7 TFO reason codes

Value	Description
b00	Normal synchronization
b01	Tracing has been enabled
b10	Trace restarted after overflow
b11	ARM processor has exited from debug state

6.5.5 LSM In Progress TFO packets

LSM In Progress packets occur only when both of the following conditions occur simultaneously:

- Trace is enabled in the middle of a LSM multiple memory access instruction. See *Definitions* on page 4-21 for a list of these instructions.
- Another instruction is currently executing.

An LSM In Progress TFO packet comprises the following contiguous components:

A header byte

Broadcast first. The header byte contains the 2-bit reason code that is labeled as RR in Table 6-6 on page 6-17). For more information about TFO reason codes, see *TFO reason codes* on page 6-18.

Context ID The number of Context ID bytes traced (0 to 4) is statically determined by ETM Control Register bits [15:14]. For more information on Context ID, see *Context ID tracing* on page 6-20.

The instruction address for the LSM

The LSM instruction address is a fixed 4-byte address with bit [0] specifying the Thumb bit.

The compressed current instruction address

The address for the instruction currently executing (1 to 5 bytes) is compressed using the same technique as is used for branch addresses (described in *Branch Address trace packets* on page 6-22).

This instruction address is compressed relative to the full address from the LSM instruction. The next instruction **PIPESTAT** is for the instruction pointed to by the compressed current instruction address and tracing begins in the normal way from this point forwards.

The LSM In Progress TFO packet type enables correct tracing of all instructions that touch a particular data address or data value. Without it, the LSM instruction cannot be properly traced based on the data address.

An LSM In Progress TFO packet *is not* output when the following condition occurs:

- trace is enabled in the middle of an LSM
- another instruction has been executed and has left the pipeline
- no instruction is currently executing.

In this case, a normal TFO packet is output, giving the address of the LSM. A branch address packet is output, giving the address of the next instruction to execute before it is traced.

Note

Instructions occurring underneath the LSM are traced even if tracing was programmed to turn on only during the LSM itself. Similarly, if tracing is turned on because of the instruction address of an instruction that executes underneath an LSM, an LSM In Progress TFO packet is still output.

To illustrate the differences between the Normal TFO packet and the LSM In Progress TFO packet, Table 6-8 shows the bytes that can be expected for each.

Table 6-8 Comparison of Normal and LSM in progress TFO packets

Normal TFO packet	LSM in progress TFO packet
Normal Header (1 byte)	LSM in Progress header (1 byte)
Context ID (0-4 bytes)	Context ID (0-4 bytes)
Instruction Address (4 bytes)	LSM Address (4 bytes)
(Not applicable)	Instruction Address (0-5 bytes)

6.5.6 Data address synchronization

The full data address output is made every *n* cycles, where *n* is the counter value programmed in the Synchronization Frequency Register (see *Synchronization Frequency Register, ETMv2.0 and later* on page 3-69). The default counter value is 1024. Every time the counter reaches zero, the next data address output is always a full 5-byte address.

It is expected that a single counter is used for both data address synchronization and TFOs, and that the counter values are staggered to reduce the likelihood of overflow.

The full address encoding is shown in Figure 6-3 on page 6-25.

6.5.7 Context ID tracing

The unique header value for Context ID updates enables the decompressor to recognize Context ID changes even when tracing through code regions that are not decompressable (any region for which a binary is not available). For more information on trace packet headers, see *Trace packet headers* on page 6-8.

The Context ID value is output when either of the following occurs:

- the Context ID value is updated
- a TFO packet is output.

6.6 Tracing through regions with no code image

Decompressing the trace requires the code image to be available. However, the code image is often not available for some areas of memory, such as system libraries, and it is not practical to filter all these regions out. These are referred to as unknown regions. The decompressor can resume tracing when an indirect branch occurs to a known region, without having to wait for the next TFO. The protocol is designed to enable the length of each packet to be determined without reference to the code image, so that alignment synchronization is not lost. The following information must continue to be monitored:

Branch addresses These must be monitored to keep track of the last output address, used to compress branch addresses.

Data addresses These must be monitored so that the first data address can be decompressed.

Context IDs These can still be traced.

When tracing from a known region to an unknown region, data corresponding to the last data instruction in the known region must be discarded if both of the following occur:

- the last data instruction did not have all of its data traced
- the first Normal Data or Load Miss Occurred packet in the unknown region does not have its A bit set to 1.

This is because the first data traced in the unknown region might correspond to the last data instruction in the known region, or to a CPRT instruction at the beginning of the unknown region. Alternatively, the decompressor can discard all data corresponding to the last data instruction in the known region whenever an unknown region is encountered.

6.7 Instruction tracing with ETMv2

Instruction trace works by outputting the destination address of branches. This section contains information about instruction tracing that relates specifically to ETMv2. For a more general description of instruction tracing with the ETM, see Chapter 4 *Signal Protocol Overview*.

6.7.1 Branch Address trace packets

When a processor performs a branch operation, the destination of the branch is often reasonably close to the current address. The spatial locality of branch destinations provides additional compression of the branch addresses. It is necessary to output only the low order bits that have changed since the last branch or TFO. The full address can be reconstructed when decompression of the trace information takes place.

To decide how many bytes are required, the on-chip logic registers the last branch address that it has output, and when another branch occurs, the new address is compared with the one that was previously output. Only sufficient low order bits must be output to cover all the bits that have changed in the address. For example, if the upper 12 bits of the address are unchanged and **A[19]** is the most significant bit to have changed, then it is only necessary to output **A[19:0]**. This can be done in three address packets instead of five.

A full 32-bit address is output over five bytes. When an address is output that is less than 32 bits, the new address value replaces the appropriate bits in the previously output branch address. The value does not have to be added to or subtracted from the previous value, nor is it based on the immediately preceding PC value.

If present, a branch target address is always the last item to be placed into the FIFO on a given cycle. Reason codes are output as part of the TFO packet header (see *TFO packet headers* on page 6-17).

A branch address can be made up of a maximum of five bytes. Bit [7] is asserted in every byte except the last. This enables the decompressor to detect the last byte of an address.

If an instruction that causes an indirect branch is traced, a branch address packet must be output even if the target of the branch is not traced. This enables the address of the first instruction in any trace gap to be determined. If a branch address packet is output in a cycle in which no instruction is executed (the **PIPESTAT** is DW (Wait with Data), the branch corresponds to the most recent instruction executed.

Branch address generation

This section describes how a branch address is produced. The sequence is shown in Figure 6-1 on page 6-23 for ARM addresses and Figure 6-2 on page 6-24 for Thumb addresses.

The address is produced as follows:

1. The address is prefixed with a 1 in the position of bit [33]. The decompressor uses the position of this bit in the final address to identify whether the code is currently in ARM or Thumb state.
2. If the packet is an ARM address, it is shifted right by two bits (ARM addresses are word-aligned so the first two bits are always zero).

   If the packet is a Thumb address, it is shifted right by one bit (Thumb addresses are halfword-aligned so the first bit is always zero).
3. A 1 is added to the end of the packet. This identifies the packet as a branch address.

4. The value produced (at most 33 bits wide) is divided into 7-bit quantities.

5. An address continue bit is added to each 7-bit fragment. This bit is set to 1 in all but the last byte of the address packet.

This encoding mechanism means that ARM and Thumb addresses can always be uniquely identified by the high order bits of the fifth address byte.

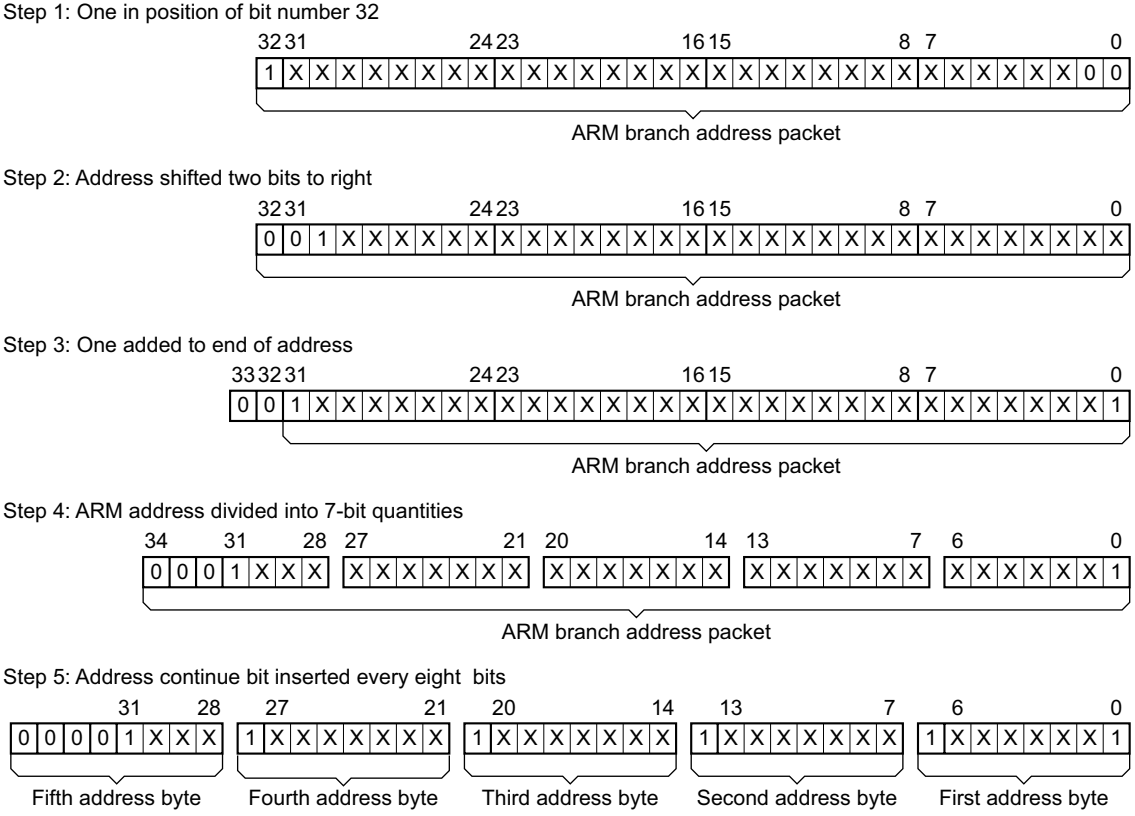


Figure 6-1 Generating an ARM branch address

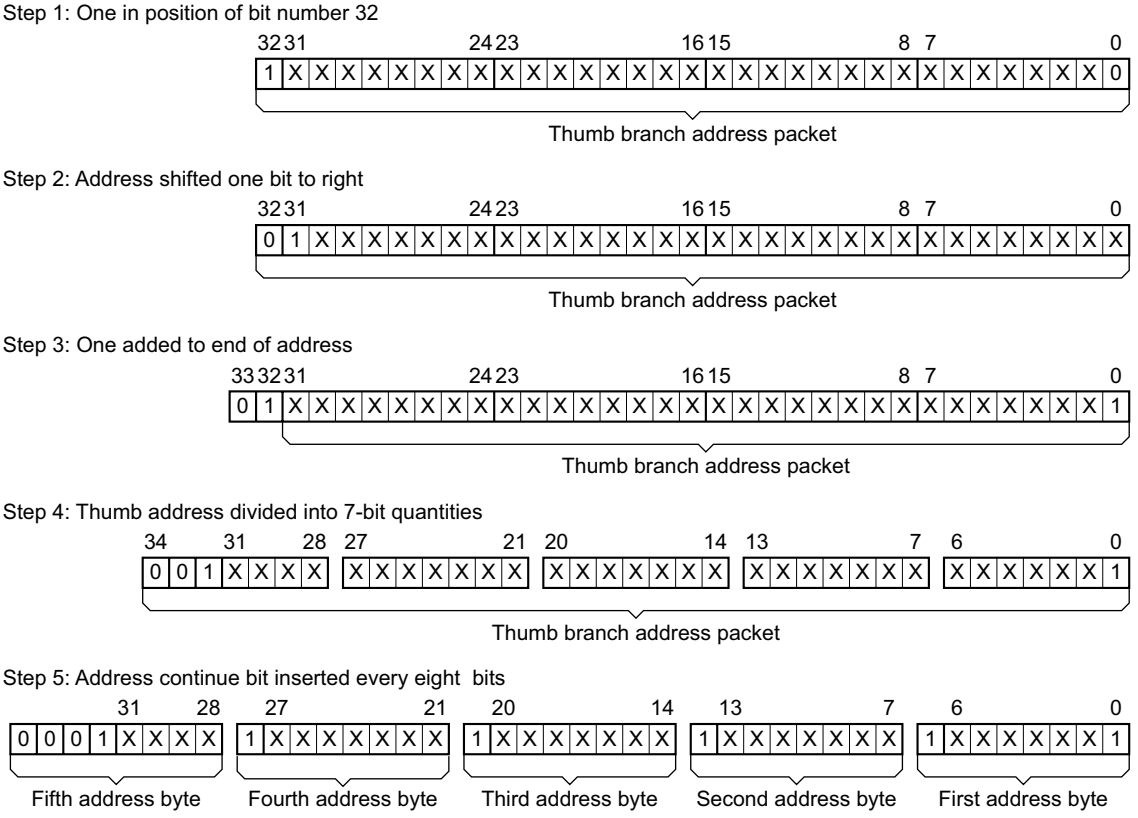


Figure 6-2 Generating a Thumb branch address

Exception branch addresses

Bit [6] of the fifth byte of an ARM address packet (the E bit) is used to indicate an exception branch address (see Table 6-9 on page 6-25). This bit is asserted on any branch that is because of a canceling exception. This enables the decompressor to recognize and inform you that these interrupted instructions were canceled (see *Exceptions* on page 4-9).

There is no E bit for Thumb addresses because all exception vectors are executed in ARM state.

Table 6-9 ARM and Thumb 5-byte addresses

5-byte address	
ARM	**Thumb**
b1XXXXXX1	b1XXXXXX1
b1XXXXXXX	b1XXXXXXX
b1XXXXXXX	b1XXXXXXX
b1XXXXXXX	b1XXXXXXX
b0E001XXX	b0001XXXX

All encodings of the fifth address byte not specified in Table 6-9 are reserved.

The complete encoding of a full branch address is shown in Figure 6-3.

Note

For future compatibility, when decompressing the trace you must enable the E bit to be set to 1 when branching to any exception vector. The most recent instruction traced is canceled and must be ignored.

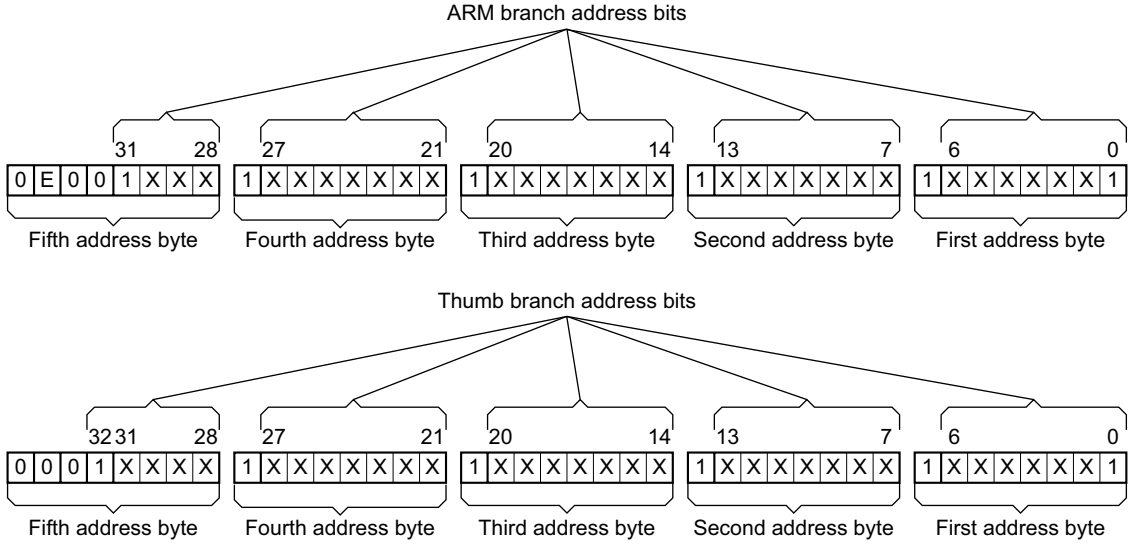


Figure 6-3 Full branch address encodings for ARM and Thumb states

6.7.2 Full branch address reason codes

In ETMv2, reason codes correspond to TFO packets instead of branch address packets, and are encoded as part of the *Trace FIFO Offset* (TFO) packet header. For more information about reason codes, see *TFO reason codes* on page 6-18.

6.8 Data tracing in ETMv2

Data trace works by outputting the data accesses (that is address, data, or both) performed by the processor. Data trace packets are described in *ETMv2 trace packets* on page 6-6. This section contains additional information about data tracing in ETMv2. For a more general description of data tracing with the Embedded Trace Macrocell, see Chapter 4 *Signal Protocol Overview*.

6.8.1 Data aborts

If one or more of the data accesses was aborted by the memory system, a branch address to the data abort exception vector is also output as part of the same instruction. See *Data address synchronization* on page 6-20 for details.

A data abort can occur on any or all of the data transferred. Data tracing ignores the abort status of data transferred. All transferred data for an instruction, whether aborted or not, is traced. The decompressor must ignore all data traced by an instruction that causes an abort.

For information about specifying comparator behavior when data aborts occur, see *Exact matching for data address comparisons* on page 2-46.

Imprecise data aborts, ETMv2.1 and later

ETMv2.1 and later support imprecise data aborts. The implications of this are:

Trace implications

Regular (precise) data aborts are traced as non-canceling exceptions because the instruction that causes the data abort is regarded as having executed, and the exception bit in the branch to the data abort exception vector is not set to 1.

Imprecise data aborts are traced as canceling exceptions. The exception bit in the branch to the exception vector is set to 1, and the last instruction traced before this branch is deemed not to have executed and must be ignored.

Resource implications

All resources treat an imprecise data abort in the same way as any other canceling exception, not as a data abort. For example, an imprecise data abort does not prevent a data address comparator with its Exact match bit set to 1 from matching, but prevents an instruction address comparator with its Exact match bit set to 1 from matching on the last instruction traced.

6.8.2 Decoding the data trace packets

Data trace is performed by trace packets, described in *ETMv2 trace packets* on page 6-6. To interpret data trace, you require the code image. However, tracing through a region for which the code image is not available does not cause synchronization to be lost (see *Tracing through regions with no code image* on page 6-21).

6.8.3 Address compression performed by the ETM

The ETM compresses the data trace by reducing the number of bits that are output for the address of the data transfer. The same technique is used as for branch addresses, where a copy of the last data access address is kept and only the low order bits that have changed are output for the next address.

This is particularly effective, for example, if you are viewing data in one small address range, because all the traced data accesses have the same high order address bits.

When the address of a data access is output it is compressed only if both of the following conditions are satisfied:

- A full 32-bit data address has been output in the synchronization period that ends with the current cycle. The synchronization period is set in the Synchronization Frequency Register, `0x78`, if present, and otherwise is 1024 cycles.
- There has been no interruption in tracing.

Otherwise no compression takes place, and the full 32-bit address is used to ensure that the captured trace information contains a reference point for the other compressed address packets.

6.9 Filtering the ETMv2 trace

The ETM has a **TraceEnable** function that you can use to enable or disable tracing. This signal is typically used to select the areas of code that are traced and to disable the trace when code is executed that is of limited use to the debugging process. The advantage of disabling the trace information is that it effectively increases the amount of useful information that can be captured by a given size of buffer in the TPA, enabling selective tracing over a longer time period.

6.9.1 Enabling trace

When **TraceEnable** becomes active then tracing is enabled. The trace port outputs a TFO packet that includes a full 32-bit address. The full address output indicates to the trace decompressor software that there is a discontinuity in the trace. It enables the decompressor to synchronize the trace information from the first instruction after the trace is enabled.

When tracing is enabled the TFO header byte contains one of the following reason codes (see Table 6-7 on page 6-18):

- tracing has been enabled by the **TraceEnable** signal
- tracing has been restarted after a FIFO overflow
- tracing has been restarted following exit from debug state.

6.9.2 Disabling trace

When **TraceEnable** becomes inactive, tracing stops and the pipeline status changes to Wait (WT) while the FIFO drains. When there is no data left in the FIFO the pipeline status changes to Trace Disabled (TD). This enables the TPA to suppress tracing, and so improve the trace buffer utilization.

Note

Trace disabled cycles can correspond to wait cycles. In cycle-accurate tracing it might be necessary for the TPA to capture these cycles. For more information see *Cycle-accurate tracing* on page 6-31.

6.9.3 Data accesses during disabled trace

When the trace port is disabled you cannot view data accesses and the **ViewData** output is ignored.

6.10 FIFO overflow

Sometimes, so much trace information is generated on-chip that the FIFO can overflow. When this occurs the ETM uses a two-stage process to empty the FIFO and restart the trace:

1. The pipeline status is changed to Wait and the FIFO empties. This ensures that all trace information up to the overflow condition is collected. This trace information might be required to determine the cause of the FIFO overflow.

2. When the FIFO has drained, if **TraceEnable** is still active tracing is re-enabled as soon as possible.

Note

Either all or none of the data to be placed in the FIFO in a given cycle must be traced. The ETM must not place any data in the FIFO unless there is room for all the data generated in that cycle.

Bit [0] of the ETM Status Register provides a pending overflow flag, indicating that an overflow has occurred but that the FIFO overflow reason code has not been generated (see *ETM Status Register, ETMv1.1 and later* on page 3-33). This is required where tracing stops because of an ARM breakpoint, but before tracing can be restarted.

6.11 Cycle-accurate tracing

When profiling the execution of critical code sequences, it is often useful if you can observe the exact number of cycles that a particular code sequence takes to execute. To perform this *cycle-accurate tracing*, you must set bit [12] of the ETM Control Register to 1, see *ETM Control Register* on page 3-20.

When cycle-accurate tracing is enabled, **TRACEPKT[0]** is HIGH when **PIPESTAT** has the value `0x7` (TD). This causes the TCD to capture trace on all cycles, even if there is no trace to output on that cycle. The number of cycles taken by a region of code can therefore be determined by counting the number of cycles of trace captured.

Cycle-accurate tracing is disabled when:

- tracing is disabled (that is, when **TraceEnable** is inactive or prior to restarting following FIFO overflow.)
- the ARM processor enters debug state.

Chapter 7
ETMv3 Signal Protocol

This chapter describes the signals output from the ETMv3.x trace port that are not backwards-compatible with previous ETM architecture versions. It contains the following sections:

7.1 Introduction

The major areas of improvement from ETMv2 are:

- Introduction of P-headers. The **PIPESTAT** signal is removed, the information is now embedded into a single packet stream. Advantages are:
  - improvements in bandwidth efficiency especially when data trace is disabled
  - improved efficiency with the *Embedded Trace Buffer* (ETB)
  - trace port speed can be decoupled from core clock speed
  - trace can be stored as a raw byte stream.
- Jazelle™ support.
- The **FIFOFULL** signal is replaced with a data trace suppression mechanism. If overflow is imminent then the ETM stops tracing data instead of stopping the core.

7.2 Packet types

All trace information is output in packets over a single set of trace pins, **TRACEDATA**. See *Trace port interface* on page 7-76 for information on **TRACEDATA**. Each packet comprises:

- a one-byte header
- zero or more bytes of payload.

The headers are defined in the following paragraphs and are described in more detail later in this chapter.

The header encodings are listed in Table 7-1.

Table 7-1 Header encodings

Header description	Value	Payload (max bytes)	Category	Remarks
Branch address	bCxxxxxx1	Additional address bytes (5)	Instruction	No header. C = another byte follows. See *Branch packets* on page 7-11.
A-sync	b00000000	None but repeated	Sync.	Alignment synchronization. See *A-sync, alignment synchronization* on page 7-65.
Cycle count	b00000100	Cycle count (5)	Instruction	1 to (2^{32}–1) x **W** (see *P-headers* on page 7-5). See *Cycle count packet* on page 7-10.
I-sync	b00001000	[a](14)	Sync.	Instruction flow synchronization. See *I-sync instruction synchronization* on page 7-66.
Trigger	b00001100	None	Trace port	See *Trigger* on page 7-76.
Out-of-order data	b0TT0SS00	Data value (4)	Data	TT = tag (1-3), SS = data value size. See *Out-of-order data* on page 7-48.
Store failed	b01010000	None	Data	For use with the ARMv6 instruction `STREX`
I-sync with cycle count	b01110000	[a](19)	Sync.	See *I-sync instruction synchronization* on page 7-66.
Out-of-order placeholder	b01A1TT00	Address (5)	Data	TT = tag (1-3), A = Address follows where address tracing is enabled. See *Out-of-order placeholder* on page 7-47.
Reserved	b00x1xx10	-	-	-
Reserved	b0001xx00	-	-	-
Reserved	b0011xx00	-	-	-

Table 7-1 Header encodings (continued)

Header description	Value	Payload (max bytes)	Category	Remarks
Normal data	b00A0SS10	Address (5) Data value (4)	Data	A = address expected, SS = data value size. See *Normal data packet* on page 7-45.
Reserved	b010xxx10	-	-	-
Data suppressed	b01100010	None	Data	See *Data suppressed packet* on page 7-51.
Ignore	b01100110	None	Trace port	See *Ignore* on page 7-77.
Value not traced	b011A1010	Address (5)	Data	A = address follows. See *Value not traced packet* on page 7-50.
Context ID	b01101110	Context ID (4)	Instruction	See *Context ID packets* on page 7-43.
Exception exit	b01110110	None	Instruction	See *Tracing return from an exception* on page 7-59.
Exception entry	b01111110	None	Instruction	*Automatic stack push on exception entry and pop on exception exit* on page 7-57.
Reserved	b01110010	-	-	-
P-header	b1xxxxxx0	None	Instruction	See *P-headers* on page 7-5.

a. For more information, see the section referred to in the *Remarks* column.

Headers are described in the following sections:

- *Instruction tracing* on page 7-5
- *Data tracing* on page 7-44
- *Synchronization* on page 7-65
- *Trace port interface* on page 7-76.

Some bits in the payload of certain packets are defined as Reserved. These bits are always zero in current versions of the architecture, but might indicate additional information in future versions. These bits can be ignored.

7.3 Instruction tracing

This section describes how the execution of instructions is represented in the trace. Instruction trace is represented by:

- P-headers, that indicate the execution of instructions
- Branch packets, that indicate the address of instructions, where this cannot be inferred from the address of the previous instruction. These are referred to as indirect branches.
- Context ID packets, that indicate a change in the executing process or memory map.

Synchronization of the instruction trace is also required. This is described in *Synchronization* on page 7-65.

7.3.1 P-headers

P-headers represent a sequence of *Atoms* that indicate the execution of instructions or Java bytecodes. There are three atom types, as follows:

- **E** is an instruction that passed its condition codes test
- **N** is an instruction that failed its condition codes test
- **W** is a cycle boundary, and occurs in cycle-accurate mode only.

These atoms are mapped onto several P-header encodings for efficient output in the trace. Different encodings are, depending on whether cycle-accurate mode is enabled. Where cycle-accurate tracing is not required, a more compressible stream can be generated by removing the **W** atoms.

Generation

The rules for P-header generation are IMPLEMENTATION SPECIFIC. Any P-headers that unpack to the correct set of P-header atoms is permitted, however inefficient. In the extreme, a device might generate one P-header per cycle, although this is not a preferred implementation.

P-header encodings in non cycle-accurate mode

The P-header encodings in non cycle-accurate mode are listed in Table 7-2.

Table 7-2 P-header encodings in non cycle-accurate mode

Description	Value	Payload (max bytes)	Remarks
Format 1 P-header	b1NEEEE00	None	0-15 x **E**, 0-1 x **N** Bits [5:2], shown as EEEE, are the count of **E** atoms.
Format 2 P-header	b1000FF10	None	1 x (**N**/n**E**), 1 x (**N**/n**E**) Bit [3] represents the first instruction and bit [2] represents the second instruction.
Reserved	b1001xx10	-	-
Reserved	b101xxx10	-	-
Reserved	b11xxxx10	-	-

Example 7-1 shows two non cycle-accurate mode encodings and their meanings.

Example 7-1 P-header encodings in non cycle-accurate mode

A header of value b11001000 is encountered in the trace when cycle-accurate mode is disabled. This is a format 1 P-header representing the atoms **EEN**. It indicates that two instructions were executed and passed their condition codes, followed by one instruction that failed its condition codes.

A header of value b10001010 is encountered in the trace when cycle-accurate mode is disabled. This is a format 2 P-header representing the atoms **NE**. It indicates that one instruction was executed that failed its condition codes, followed by one instruction that passed its condition codes.

P-header encodings in cycle-accurate mode

For details of the possible use of P-headers for tracing gaps in trace during cycle-accurate tracing, see *Tracing long gaps in cycle-accurate trace* on page 7-79.

The P-header encodings in cycle-accurate mode are listed in Table 7-3.

Table 7-3 Cycle count and P-header encodings in cycle-accurate mode

Description	Value	Payload (max bytes)	Remarks
Cycle count	b00000100	Cycle count (5)	1 to (2^{32}–1) x **W**
Format 0 P-header	b10000000	None	**W** Permitted in ETMv3.0 only. In all other versions of the ETM this encoding is reserved.
Format 1 P-header	b1N0EEE00 (except b10000000)	None	0-7 x (**WE**), 0-1 x **WN** Bits [4:2], shown as `EEE`, are the count of **WE** atoms.
Format 2 P-header	b1000FF10	None	1 x **W**, 1 x (**N**/n**E**), 1 x (**N**/n**E**) Bit [3] represents the first instruction and bit [2] represents the second instruction.
Format 3 P-header	b1E1WWW00	None	1-8 x **W**, 0-1 x **E** Bits [4:2], shown as `WWW`, are the count of W atoms.
Format 4 P-header	b10010F10	None	1 x (**N**/n**E**) Only supported in ETMv3.3 and later. In ETMv3.2 and earlier the 10010x10 encodings are reserved.
Reserved	b10011x10	-	-
Reserved	b101xxx10	-	-
Reserved	b11xxxx10	-	-

Example 7-2 shows three non cycle-accurate mode encodings and their meanings.

Example 7-2 P-header encodings in cycle-accurate mode

A header of value b1100 1000 is encountered in the trace when cycle-accurate mode is enabled. This is a format 1 P-header representing the atoms **WEWEWN**, that indicates that three instructions were executed as follows:

Cycle 1 An instruction was executed and passed its condition codes.

Cycle 2 An instruction was executed and passed its condition codes.

Cycle 3 An instruction was executed and failed its condition codes.

A header of value b1000 1010 is encountered in the trace when cycle-accurate mode is enabled. This is a format 2 P-header representing the atoms **WNE**, that indicates that two instructions were executed as follows:

Cycle 1 An instruction was executed that failed its condition codes, followed by an instruction that passed its condition codes.

A header of value b1110 1000 is encountered in the trace when cycle-accurate mode is enabled. This is a format 3 P-header representing the atoms **WWWE**, that indicates that one instruction was executed as follows:

Cycle 1 No instructions were executed.

Cycle 2 No instructions were executed.

Cycle 3 An instruction was executed and passed its condition codes.

The cycle-accurate mode format 4 P-header, ETMv3.3 and later

ETMv3.3 introduces a format 4 P-header in cycle-accurate mode. This packet is used to indicate that an instruction has executed without any cycle boundary. This is required when two instructions are executed in the same cycle and the first instruction is either an indirect branch or has some data associated with it.

The encoding of the format 4 P-header is included in Table 7-3 on page 7-7. Table 7-4 illustrates a case where this packet is required.

Table 7-4 Use of format 4 P-header in cycle-accurate mode

Cycle count	Event	Atoms	Packet, with P-header encoding
1000	Instruction at address `0x1000` in ALU pipe 0	W E	Format 1 P-header: 8'b10000100
1000	Data for instruction in ALU pipe 0	D	Data header
1000	Instruction at address `0x1004` in ALU pipe 1	E	Format 4 P-header: 8'b10010010

7.3.2 Condition codes on canceled and undefined instructions

Some instructions that pass their condition codes but that are subject to a later canceling exception or Undefined Instruction exception might be traced as having failed their condition codes to prevent them being considered as a data instruction. See *Exceptions on Data Instructions* on page 7-55.

7.3.3 Cycle information, for cycle-accurate tracing

For more information about cycle-accurate tracing in ETMv3 see *Cycle-accurate tracing* on page 7-79.

Cycle-accurate mode enables some trace packets to be traced along with the cycle in which they are generated. Cycle information is output using the following packets:

- P-header, see *P-headers* on page 7-5
- Cycle count, see *Cycle count packet* on page 7-10
- I-sync with cycle count, see *Normal I-sync with cycle count packet* on page 7-68.

In addition to giving the cycle count for instruction trace, cycle count information is meaningful for the following packets:

- Trigger, in ETMv3.1 and later
- Out-of-order placeholder
- Normal data
- Data suppressed
- Value not traced
- Context ID.

You cannot rely on the cycle count information for other packets. The packets for which cycle count information is not relevant are:

- Branch address
- A-sync
- I-sync
- Out-of-order data
- Store failed
- Trigger, in ETMv3.0 only
- Ignore
- Exception entry
- Exception exit.

Cycle information can only give cycle information for a particular point in the core pipeline. In complex cores some stages in the pipeline might be capable of advancing independently of others, with the effect that the number of cycles between instructions is not the same at all points in the pipeline. You must be aware of these limitations when interpreting the cycle information.

I-sync packets might contain a cycle count, but you cannot rely on this count to determine the precise cycle of the I-sync packet. However, if you use the cycle count information with the I-sync, plus the other cycle information produced by P-headers, the cycle accuracy is maintained for instructions and for the other packets for which the cycle count information is meaningful.

7.3.4 Cycle count packet

A Cycle count packet consists of a Cycle count header followed by 1-5 bytes of data, as shown in Figure 7-1. A 1 in bit [7] of each byte indicates that another byte follows, in the same way as branch addresses. Up to 32 bits are output in this way. Any missing high-order bits are 0. This value is a number of **W**s, inserted before the most recent Non-periodic I-sync packet. This enables the number of cycles between trace regions to be output efficiently. Future versions of the architecture might support larger cycle counts. For more information see:

- *Synchronization* on page 7-65, for information on synchronization
- *Branch packets* on page 7-11, for information on branch addresses.

A cycle count of zero indicates a counter overflow. When this is encountered, the length of the gap is unknown.

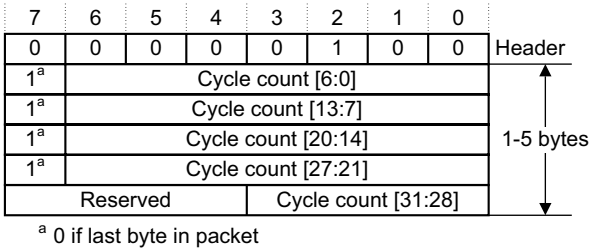


Figure 7-1 Cycle count packet

The cycle counter is reset whenever the Programming bit or the power-down bit is set to 1 in the ETM Control Register, register `0x00`. The reset value of the counter is zero, indicating counter overflow.

In ETMv3.0, the Cycle count output following overflow or entry to debug state must be ignored. In ETMv3.1 and later, the trace is cycle-accurate through overflow and debug state.

When tracing in cycle-accurate mode, a cycle count is required for every Non-periodic I-sync packet to indicate the number of cycles (W atoms) since the last P-header packet prior to the I-sync packet. This is output as follows:

- The I-sync packet, followed by a Cycle count packet before the next Non-periodic I-sync packet. The Cycle count packet might not be present if there is a subsequent ETM FIFO overflow, and in this case the cycle count is unknown.
- The I-sync packet is a normal I-sync with cycle count packet.
- The I-sync packet is a *Load/Store in Progress* (LSiP) I-sync with cycle count packet.

For more information about synchronization, see *I-sync instruction synchronization* on page 7-66.

7.3.5 Branch packets

Branch packets are used to indicate the destination address of indirect branches. See *Direct and indirect branches* on page 4-9 for more information on indirect branches. Branch packets are also used to give information on exception types, and to indicate changes of the instruction set state or security state of the processor.

This section provides a full description of all possible branch packets in ETMv3.0 and later. A quick reference summary of all situations where branch packets are generated is given in *Summary of branch packets, ETMv3.0 and later* on page A-13.

If an instruction that causes an indirect branch is traced, a Branch address packet must be output even if the target of the branch is not traced, unless prevented by a FIFO overflow. This enables the address of the first instruction in any trace gap to be determined.

Multiple branch packets can be output for a single instruction. If this happens, each must be interpreted in turn in relation to the previous branch packet, after which all but the final branch packet must be ignored.

In cycle-accurate mode, the branch might not be traced on the same cycle as the instruction.

Branch packets summary

In all ETM versions from ETMv3.0

A Branch address packet is made up of a maximum of five address bytes, optionally followed by an Exception information byte.

Note

In earlier issues of the ETM Architecture Specification, the Exception information byte was called the *branch continuation byte.*

Bit [7] is always asserted in every address byte other than the last. Bit [7] of the last address byte:

- Is set to 0 if there is no exception, or if the five address bytes are followed by an Exception information byte. This enables the decompressor to detect the last byte of the address.
- Can be set to 1 to indicate an exception. In this case, a full 5-byte address is required, with bit [7] set to 1 for the last byte.

When no exception occurs, the address information is compressed, if possible, as described in *Branch address compression, original scheme* on page 7-16.

An exception can be traced in one of two ways:

- With a full 5-byte packet, with bit [7] set for the last byte. The last byte includes information about the exception, as shown in Table 7-8 on page 7-26. However, this format is deprecated, in favour of using the Exception information byte to indicate the exception.

- With a 6-byte packet, comprising five address bytes followed by a single Exception information byte. The inclusion of an Exception information byte is indicated by the contents of the fifth byte of the packet, as shown in Table 7-8 on page 7-26. The encoding of the Exception information byte is shown in Table 7-10 on page 7-28.

These exception branch packets are described in *Exception branch addresses packets* on page 7-24.

From ETMv3.4

The branch packet formats used in ETMv3.3 and earlier are also available in ETMv3.4. However, in general a Branch address packet is made up of between one and five address bytes followed by between zero and three *Exception information bytes*.

It is IMPLEMENTATION DEFINED which of two address compression schemes is used:

- The format that is always used in ETMv3.0 to ETMv3.3, where:
  - address compression is only possible when there is no exception
  - tracing an exception requires five address bytes, possibly followed by an Exception information byte, as described earlier.
- An alternative format, that supports address compression when tracing an exception. Tracing an exception requires between two and five address bytes, followed by an Exception information byte. For more information see *Branch packet formats with the alternative address compression scheme* on page 7-18.

ETMv3.4 defines additional branch packets for tracing the extended exception handling features provided in the ARMv7M architecture. These branch packets are made up of between one and five address bytes followed by between one and three Exception information bytes. For more information see *Extended exception handling, from ETMv3.4* on page 7-33.

From ETMv3.4, a bit in the ETM ID Register indicates which address compression scheme is implemented. This is bit [20] of the register, see *ETM ID Register, ETMv2.0 and later* on page 3-71. If this bit is:

- set to 0 the implementation uses the original address compression scheme
- set to 1 the implementation uses the alternative address compression scheme.

Table 7-5 summarizes the branch packet formats for the two address compression schemes.

Table 7-5 Summary of branch packet lengths, with original and alternative address compression

Packet type	Compression scheme[a]		Notes
	Original	Alternative[b]	
Simple branch[c]	1-5 addr.	1-5 addr.	The 1 byte and 5 byte packets are identical in the two schemes.
Branch with state change, no exception	(5 addr.) + (0-1 excp.)		The schemes are identical. The Exception information byte is only used if state change is to or from ThumbEE state.
Branch with security state change, no exception	(5 addr.) + (1 excp.)	(2-5 addr.) + (1 excp.)	
Branch with exception[d]	(5 addr.) + (0-1 excp.)	(2-5 addr.) + (1 excp.)	In the original scheme, use of the encoding without an Exception information byte is deprecated.
Branch with extended ARMv7-M exception	(5 addr.) + (1-3 excp.)	(1-5 addr.) + (1-3 excp.)	

a. The address compression scheme. These columns give the packet lengths under the two addressing schemes. In theses columns, *addr* refers to the address bytes, and *excp* refers to the Exception information bytes.
b. The alternative address compression scheme is only available from ETMv3.4.
c. Branch with no exception, or change of instruction set state or security state.
d. Packet lengths are the same if there is also a change of processor instruction set state, security state, or both. This row of the table excludes packets for tracing the extended exceptions introduced in the ARMv7-M architecture.

The branch packets generated under the two schemes are described in:

- *Branch packet formats with the original address compression scheme*
- *Branch packet formats with the alternative address compression scheme* on page 7-18.

These descriptions do not include the additional branch packets use for tracing the extended exceptions implemented on processor cores that implement the ARMv7-M architecture. These packets are available from ETMv3.4, and are described in *Extended exception handling, from ETMv3.4* on page 7-33.

More information about the generation of the branch packet address bytes is given in *Branch address generation* on page 7-39.

Branch packet formats with the original address compression scheme

In this section, where field names in branch packets are abbreviated, the abbreviations are explained in Table 7-10 on page 7-28.

For packets where the number of address bytes can vary, the address compression scheme is described in *Branch address compression, original scheme* on page 7-16.

Figure 7-2 shows a branch packet to an instruction in ARM state.

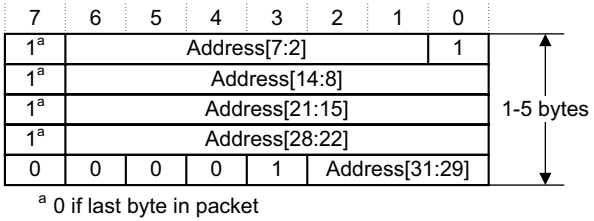


Figure 7-2 Branch packet to an instruction in ARM state

Figure 7-3 shows a branch packet to an instruction in Thumb state.

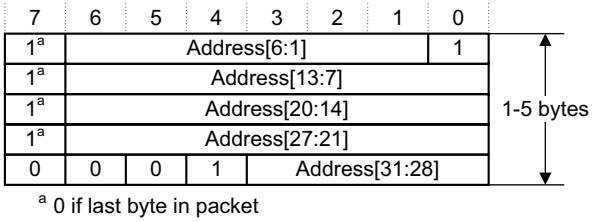


Figure 7-3 Branch packet to an instruction in Thumb state

Figure 7-4 shows a branch packet to an instruction in Jazelle state.

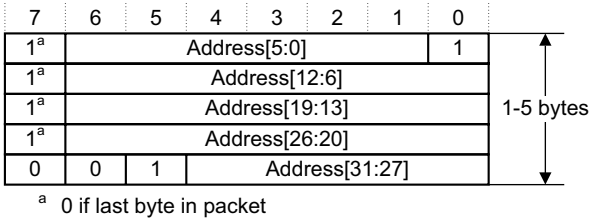


Figure 7-4 Branch packet to instruction in Jazelle state

Tracing branches with exceptions, original scheme

With the original address compression scheme, a branch with an exception always requires the trace to output five bytes of address information. In this scheme, a branch to an exception vector can be output as:

- A 5-byte packet, with information about the exception included in the fifth byte. Figure 7-5 on page 7-15 shows a 5-byte branch packet to an exception vector in ARM state.

  This format can only be used in ARM state, and is deprecated in favor of the 6-byte format.

- A 6-byte packet, with information about the exception included in the *Exception information byte*, the sixth byte of the packet. Figure 7-6 shows a branch to an exception vector in ARM state with an Exception information byte, giving a 6-byte packet. The presence of an Exception information byte is indicated by the value held in byte 5, see Table 7-8 on page 7-26.

Note

This description does not include the tracing of the extended exceptions implemented on ARMv7-M processors. Tracing these exceptions is described in *Extended exception handling, from ETMv3.4* on page 7-33.

Exception branch addresses packets on page 7-24 describes the exception information given in the five byte and six byte Exception branch packets.

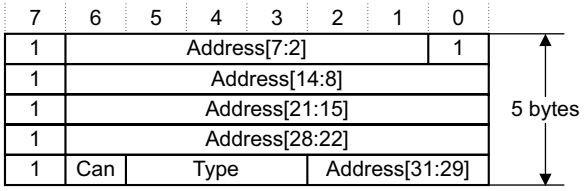
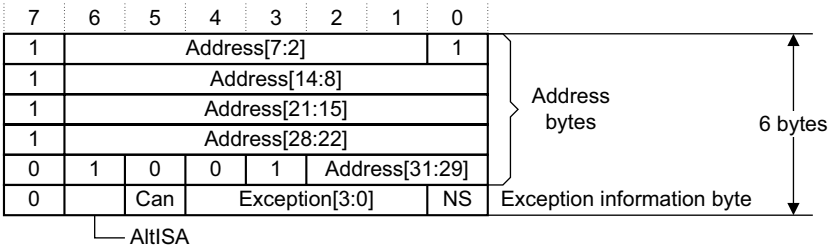


Figure 7-5 Branch packet to an exception vector in ARM state (5-byte packet)



The AltISA (Alternative ISA) bit is only defined from ETMv3.3. In earlier ETM versions this bit is always 0.

Figure 7-6 Branch packet to an exception vector in ARM state, with Exception information byte

Note

See *Branch address packets for change of processor state* on page 7-31 for details of the use of the AltISA (Alternative instruction set) bit of the branch continuation byte, in ETMv3.3 and later.

Branch address compression, original scheme

This branch address compression scheme:

- Is always implemented in ETMv3.0 to ETMv3.3
- Can be implemented in ETMv3.4 and later. In these ETM versions, when this scheme is implemented, bit [20] of ETM ID Register is set to 0, see *ETM ID Register, ETMv2.0 and later* on page 3-71.

When a processor performs a branch operation, the destination of the branch is often reasonably close to the current address. This permits compression of the branch addresses. The ETM is required to output only the low-order bits that have changed since the last branch. The full address can be reconstructed when decompression of the trace information takes place. This is why, in the diagrams Figure 7-2 on page 7-14, Figure 7-3 on page 7-14 and Figure 7-4 on page 7-14, the address length is shown as *1 to 5 bytes*.

To decide how many bytes are required, the on-chip logic registers the last branch address that it has output, and when another branch occurs, the new address is compared with the one that was previously output. Only sufficient low-order bits are output to cover all the bits that have changed in the address. For example, if the upper 12 bits of the address are unchanged and **A[19]** is the most significant bit to have changed, then it is only necessary to output **A[19:0]**. This can be done in three address bytes instead of five.

Taking a normal Thumb branch as an example:

- Figure 7-7 shows the single-byte packet, when there is no change in **A[31:7]**
- Figure 7-8 shows the 2-byte packet, when there is no change in **A[31:14]**
- Figure 7-9 on page 7-17 shows the 3-byte packet, when there is no change in **A[31:21]**
- Figure 7-10 on page 7-17 shows the 4-byte packet, when there is no change in **A[31:28]**
- Figure 7-11 on page 7-17 shows the full 5-byte packet, when there is a change in **A[31:28]**.

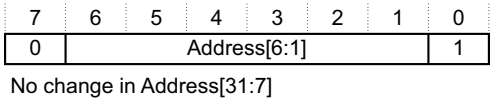


Figure 7-7 Normal Thumb branch with no change in address bits [31:7]

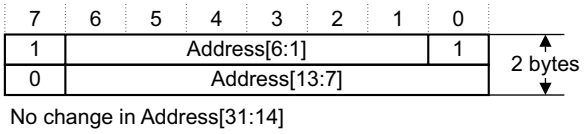


Figure 7-8 Normal Thumb branch with no change in address bits [31:14]

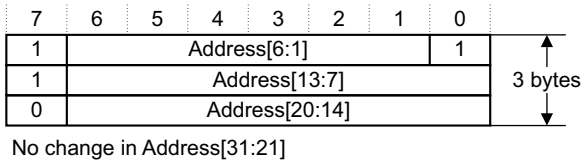


Figure 7-9 Normal Thumb branch with no change in address bits [31:21]

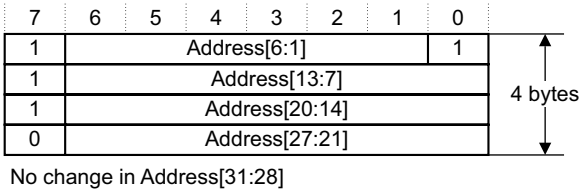


Figure 7-10 Normal Thumb branch with no change in address bits [31:28]

When there is a change in **A[31:28]**, no address compression is possible and the full 5-byte packet is required. Figure 7-11 shows this packet.

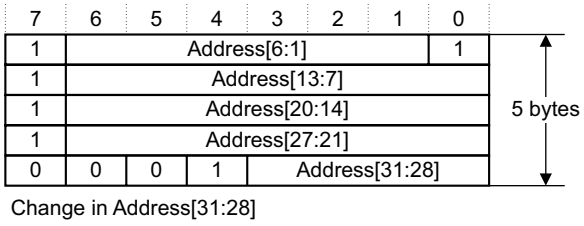


Figure 7-11 Normal Thumb branch with a change in address bits [31:28]

Figure 7-11 shows how a full 32-bit address is output over five bytes. When an address is output that is less than 32 bits, the new address value replaces the appropriate bits in the previously output Branch address or I-sync packet. The value does not have to be added to or subtracted from the previous value, nor is it based on the immediately preceding PC value. The instruction set state is unchanged from the previous address.

In ETMv3.3 and earlier, whenever an exception occurs the full five bytes of address information must be output, even when only low-order address bits have changed.

Branch packet formats with the alternative address compression scheme

In the normal scheme for branch address compression, described in *Branch address compression, original scheme* on page 7-16:

- Bit [7] is used to indicate the last address byte of the packet. It is set to 0 for the last byte, and set to 1 for all other bytes.
- Branch address compression cannot be used for exception packets.

From ETM v3.4, an ETM implementation can choose to implement an alternative scheme for branch address compression, that permits address compression on exception packets.

Bit [20] of the ETM ID register indicates whether an ETM implements the original or the alternative scheme for branch address compression. This bit is set to 1 if the alternative compression scheme is implemented.

If implemented, the alternative encoding applies to all branches in ARM, Thumb and Jazelle states, including branches on an exception.

Note

The alternative encoding does not apply to instruction set changes, including changes to or from ThumbEE state, see *Branch address packets for change of processor state* on page 7-31. How these events are traced does not depend on whether the ETM implements the original or the alternative scheme for branch address compression. These events are always traced by a packet with five address bytes followed by a single Exception information byte.

Because address compression is always applied when it is possible to do so, the branch packet formats for the alternative scheme are described in:

- *Normal (no exception) branch packets in the alternative encoding* on page 7-19
- *Exception branch packets in the alternative encoding* on page 7-22.

A debugger must be able to detect:

- the end of the address bytes in the packet
- whether the packet includes an Exception information byte.

The overall format of branch packets in this scheme is described in *Overall branch packet format in the alternative address compression scheme*.

Overall branch packet format in the alternative address compression scheme

As with the original encoding, a branch packet is one or more bytes long. In the alternative branch address compression scheme:

- when bit [7] = 1, the byte is interpreted in exactly the same way as in the original scheme
- when bit [7] = 0:
  - for the first byte of a branch packet, this is the only byte in the packet
  - otherwise, bit [6] holds information about the completion of the packet.

Table 7-6 describes the use of bits [7:6] in full. The use of these bits is clarified by the packet examples in:

- *Normal (no exception) branch packets in the alternative encoding*
- *Exception branch packets in the alternative encoding* on page 7-22.

This format also supports multiple Exception information bytes. These are required for tracing the extended exceptions implemented on ARMv7-M processors. The packets generated when tracing these exceptions is described in *Extended exception handling, from ETMv3.4* on page 7-33.

Table 7-6 Interpretation of bits [7:6], alternative branch compression encoding

Bit [7]	Bit [6]	Interpretation
1	-	The address continues in the next byte of the packet. Bit [6] of this byte of the packet is part of the Address field. The interpretation of Bit [7] = 1 is the same as it is in the original encoding.
0	1	This byte contains the final address bits for the packet. There is an Exception information byte after this byte.
0	0	This byte contains the last address bits for the packet, and is the last byte of the packet.

Normal (no exception) branch packets in the alternative encoding

In the alternative encoding, for a branch packet where no exception has occurred:

- the single-byte packet is exactly the same as in the original encoding
- longer packets comprise:
  - one or more bytes with bit [7] = 1, where bits [6:0] hold address information
  - a final byte with bits [7:6] = b00, where bits [5:0] can hold address information
- the 5-byte packet is identical to a 5-byte packet in the original encoding.

The possible encoding formats for branches without an exception are shown in:

- Figure 7-12 on page 7-20, for the single-byte packet where there is no change in **A[31:7]**. This is identical to the encoding in the original scheme.
- Figure 7-13 on page 7-20, for the 2-byte packet where there is no change in **A[31:13]**
- Figure 7-14 on page 7-20, for the 3-byte packet where there is no change in **A[31:20]**
- Figure 7-15 on page 7-20, for the 4-byte packet where there is no change in **A[31:27]**
- Figure 7-16 on page 7-20, for the full 5-byte packet, where there is a change in address bits **A[31:27]**, meaning no address compression is possible.

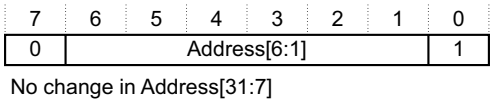


Figure 7-12 Alternative encoding of normal Thumb branch with no change in address bits [31:7]

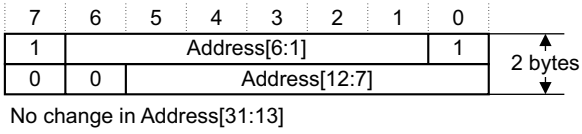


Figure 7-13 Alternative encoding of normal Thumb branch with no change in address bits [31:13]

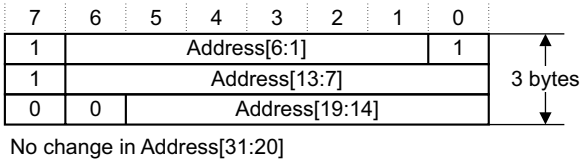


Figure 7-14 Alternative encoding of normal Thumb branch with no change in address bits [31:20]

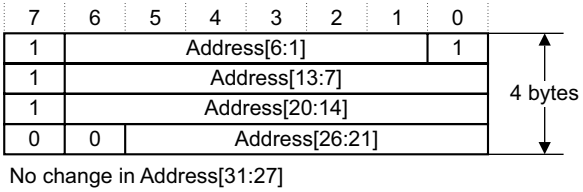


Figure 7-15 Alternative encoding of normal Thumb branch with no change in address bits [31:27]

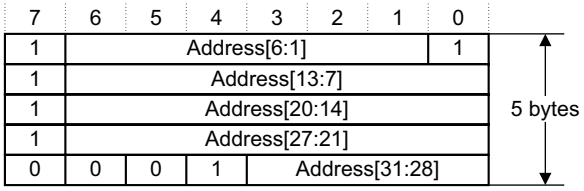


Figure 7-16 Alternative encoding of normal Thumb branch when address bits [31:27] change

These examples show all possible packet lengths for branches in Thumb state. Address compression is similar for branches in other processor states. The only difference is in the address bit range held in each byte of the packet:

- Figure 7-17 shows the trace packet for a branch in ARM state when there is no change in bit [31:14] of the address
- Figure 7-18 shows the trace packet for a branch in Jazelle state when there is no change in bit [31:19] of the address.

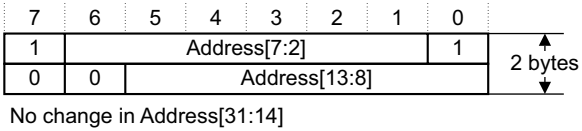


Figure 7-17 Alternative encoding of normal ARM branch with no change in address bits [31:14]

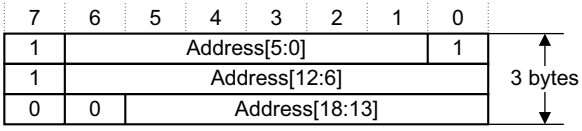


Figure 7-18 Alternative encoding of normal Jazelle branch with no change in address bits [31:19]

In all processor states, the 1-byte and 5-byte packets are identical to the 1-byte and 5-byte packets in the original encoding.

Using the alternative encoding increases the packet size output in a few cases, for example for a branch in Thumb state when the most significant change in the address is bit Address[13]. However, it reduces the packet size required for any exception case where the most significant change in the address is Address[27] or lower.

Note

In Thumb state, there is no change in trace packet length for branches where address changes are limited to bits [12:1]. This means there is no loss of efficiency in tracing tight loops. Normally, code contains many more tight loops than loops that branch to a more remote address. Similarly:

- in ARM state there is no change in trace packet length when address changes are limited to bits [13:2]
- in Jazelle state there is no change in trace packet length when address changes are limited to bits [11:0].

Exception branch packets in the alternative encoding

In the original encoding, that is always used in ETMv3.3 and earlier, address compression is not possible when an exception occurs, and an exception always requires a 5-byte or 6-byte exception packet, as shown in Figure 7-5 on page 7-15 and Figure 7-6 on page 7-15.

From ETMv3.4, the alternative encoding scheme provides address compression on exception branch packets. These exception branch packets always use a branch continuation byte for the exception information. All possible data-compression encoding formats for exception branches are shown in these diagrams:

- Figure 7-19 shows the 3-byte exception packet, issued where there is no change in **A[31:13]**
- Figure 7-20 on page 7-23 shows the 4-byte exception packet, issued where there is no change in **A[31:20]**
- Figure 7-21 on page 7-23 shows the 5-byte exception packet, issued where there is no change in **A[31:27]**
- When there is a change in A[31:27], a 6-byte exception packet is issued. This is identical to the branch extension packet with continuation byte. This packet is shown for a Thumb instruction in Figure 7-22 on page 7-23, and is shown for an ARM instruction in Figure 7-6 on page 7-15.

More information about exception packets is given in *Exception branch addresses packets* on page 7-24.

Note

These examples do not include the tracing of the extended exceptions implemented on ARMv7-M processors. Tracing these exceptions is described in *Extended exception handling, from ETMv3.4* on page 7-33.

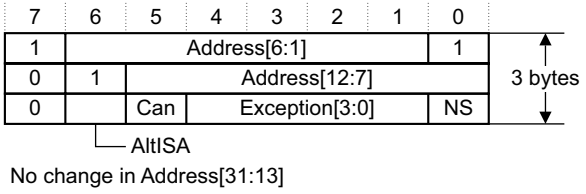


Figure 7-19 Exception Thumb branch with no change in address bits [31:13], alternative encoding

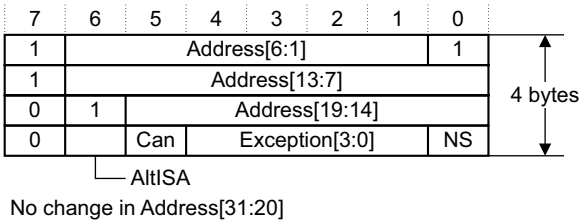


Figure 7-20 Exception Thumb branch with no change in address bits [31:20], alternative encoding

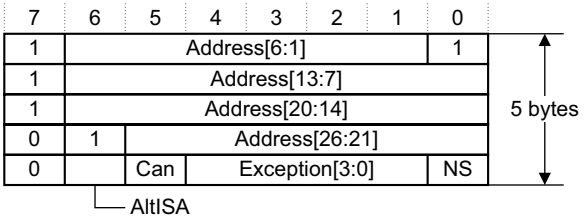


Figure 7-21 Exception Thumb branch with no change in address bits [31:27], alternative encoding

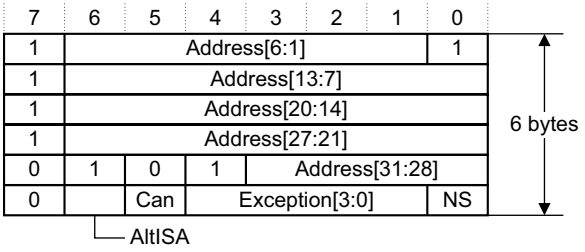


Figure 7-22 Exception Thumb branch when address bits [31:27] change, alternative encoding

Note

See *Branch address packets for change of processor state* on page 7-31 for details of the use of the AltISA (Alternative instruction set) bit of the branch continuation byte, in ETMv3.3 and later.

These examples show all possible packet lengths for exception branches in Thumb state. Address compression is similar for branches in other processor states. The only difference is in the address bit range held in each byte of the packet:

- Figure 7-23 on page 7-24 shows the trace packet for an exception branch in ARM state when there is no change in bit [31:21] of the address

- Figure 7-24 shows the trace packet for a branch in Jazelle state when there is no change in bit [31:26] of the address.

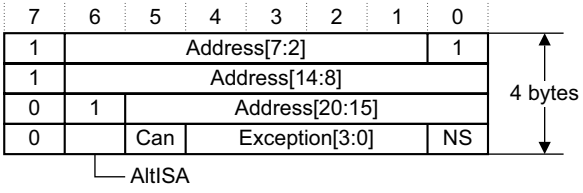


Figure 7-23 Exception ARM branch with no change in address bits [31:21], alternative encoding

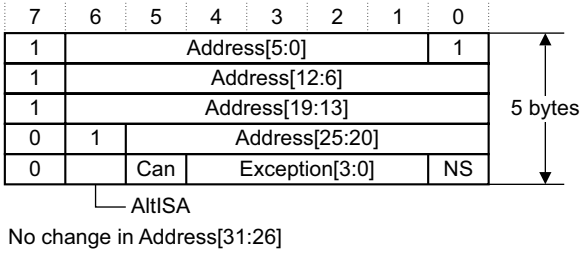


Figure 7-24 Exception Jazelle branch with no change in address bits [31:26], alternative encoding

Exception branch addresses packets

Extended information is given about exceptions, enabling:

- All exceptions to be detected without knowledge of **HIVECS**. See *Exceptions* on page 4-9 for information about the significance of the **HIVECS** signal.
- IRQ and FIQ vectors to be implemented in a floating manner by a vectored interrupt controller.
- Jazelle exceptions to be flagged explicitly.

Note

For ETMv3.4 and later, the exception handling information given in *Extended exception handling, from ETMv3.4* on page 7-33 applies in addition to the information given in this section.

The encoding of the exception information depends on the address compression scheme that is implemented:

Original address compression scheme, always used on ETMv3.3 and earlier

The exception branch packet always includes five address bytes, and can include an additional Exception information byte. Exception branches in Thumb and Jazelle states always consist of five address bytes plus an Exception information byte.

The last address byte, which is always the fifth byte of the packet:

- Indicates the processor state.
- Indicates if the packet includes an Exception information byte.
- If the exception branch occurred in ARM state, can contain details of the exception. In this case the packet does not include an Exception information byte, and last address byte is the last byte of the packet.

Note

Use of this format option is deprecated.

The format of the last address byte is shown in Table 7-8 on page 7-26.

If the packet includes an Exception information byte the encoding of that byte is shown in Table 7-10 on page 7-28.

From ETMv3.4, this address compression scheme is implemented if bit [20] of the ETM ID Register is 0, see *ETM ID Register, ETMv2.0 and later* on page 3-71.

Alternative address compression scheme, available from ETMv3.4

The exception branch packet consists of between two and five address bytes, followed by an Exception information byte.

The encoding of the Exception information byte is shown in Table 7-10 on page 7-28.

This address compression scheme is implemented if bit [20] of the ETM ID Register is 1, see *ETM ID Register, ETMv2.0 and later* on page 3-71.

When the original address compression scheme is used, in a branch packet of five bytes or more bits [7:6] of the fifth byte of the packet gives information about the packet, as shown in Table 7-7 on page 7-26. This byte is always present when an exception is traced, and the table also applies to the last byte of a 5-byte branch address packet where no exception occurred.

Table 7-7 Meaning of bits [7:6] of byte five of a branch packet on the original address compression scheme

Byte 5, bits [7:6]	Branch packet type
00	Normal branch. Byte 5 is the last byte of the packet.
01	Exception branch. At least one[a] more byte follows.
10	Deprecated non-cancelling ARM state exception. Byte 5 is the last byte of the packet.
11	Deprecated cancelling ARM state exception. Byte 5 is the last byte of the packet.

a. In ETMv3.3 and earlier, exactly one more byte follows.

In all of these cases, Table 7-8 gives a full description of all possible encodings the fifth address byte of the branch packet.

Table 7-8 Encoding of byte five of Branch address packets on the original address compression scheme

Value	Description
b00001xxx[a]	Normal ARM state branch address. Bits marked xxx are Address[31:29].
b0001xxxx[a]	Normal Thumb state branch address. Bits marked xxxx are Address[31:28].
b001xxxxx[a]	Normal Jazelle state branch address. Bits marked xxxxx are Address[31:27].
b00000xxx[a]	Reserved.
b01001xxx	ARM state branch address with continuation byte. Bits marked xxx are Address[31:29]. The next byte is an Exception information byte.
b0101xxxx	Thumb state branch address with continuation byte. Bits marked xxxx are Address[31:28]. The next byte is an Exception information byte.
b011xxxxx	Jazelle state branch address with continuation byte. Bits marked xxxxx are Address[31:27]. The next byte is an Exception information byte.
b01000xxx[a]	Reserved.

Table 7-8 Encoding of byte five of Branch address packets on the original address compression scheme (continued)

Value	Description
b1CEEExxx	Exception executed in ARM state. The C bit is set to 1 if the exception cancels the last traced instruction. The EEE bits, bits [5:3], indicate the type of exception as shown in Table 7-9. Use of this format is deprecated in favor of using an Exception information byte.

a. These values cannot occur on an exception branch.

Table 7-9 Exception encodings for bits [5:3] of the fifth address byte

EEE field[a]	Exception
b000	Processor Reset, Undefined Instruction, SVC, prefetch abort, or data abort. The exact type of exception is determined from the branch address. Software breakpoint and software watchpoint exceptions are also traced using this encoding, and are indistinguishable from prefetch aborts and data aborts respectively. The C bit of the fifth address byte is set to 1 for data aborts, and is set to 0 for Undefined Instruction exceptions and SVCs. The C bit might be 0 or 1 for processor Reset exceptions and prefetch aborts. **Note** Unusually, a non-cancelling prefetch abort can occur. This does not set the C bit to 1.
b001	IRQ.
b010	Reserved.
b011	Reserved.
b100	Jazelle. Jazelle exception, other than *unimplemented bytecode*. The exception type is determined from the branch address, in conjunction with the Jazelle exception vector table. An ordinary exception, such as a FIQ or data abort, while in Jazelle state is not a Jazelle exception and is traced using the normal encodings.
b101	FIQ.
b110	Asynchronous data abort.
b111	Debug exception.

a. Bits [5:3] of the fifth address byte.

Table 7-10 shows the encoding of Exception information byte 0. It applies to exception branch packets encoded using either of the address compression schemes.

Table 7-10 Exception information byte 0 encoding

Bits	Name[a]	Function	Description
[7]	Con	From ETMv3.4: Continue	Indicates whether another Exception information byte follows. The possible values are: **0** This is the only Exception information byte. **1** Another Exception information byte follows. For more information see *Extended exception handling, from ETMv3.4* on page 7-33.
	-	ETMv3.3[b]: Reserved	Must be ignored.
[6]	AltISA	From ETMv3.3: Alternative instruction set	If set to 1, the processor is in ThumbEE state after the branch. For more information see *The AltISA bit, ETMv3.3 and later* on page 7-29.
	-	ETMv3.2[c]: Reserved	Must be ignored.
[5]	Can	Canceled	If set to 1, the most recently-traced instruction has been canceled and must be discarded.
[4:1]	Exception	Exception[3:0][d]	Bits [3:0] of the Exception description field. With trace output from non-ARMv7-M cores[e], the encoding of this field is given in Table 7-12 on page 7-30.
[0]	NS	Security level	If set to 1, the core is in Non-secure state following the branch.

a. As shown in the Branch Packet illustrations, Figure 7-5 on page 7-15 and Figure 7-6 on page 7-15.
b. ETMv3.3 and earlier.
c. ETMv3.2 and earlier.
d. This table only describes the standard exception encodings. This information is encoded in the Exception[3:0] field of the Exception information byte. From ETMv3.4, extended exception handling is provided for processors that comply with the ARMv7-M architecture. For details see *Extended exception handling, from ETMv3.4* on page 7-33.
e. Trace output from processors that comply with the ARMv7-M architecture is described in *Extended exception handling, from ETMv3.4* on page 7-33.

The AltISA bit, ETMv3.3 and later

The fifth address byte of the packet for an exception branch address indicates the Instruction Set, see Table 7-8 on page 7-26 for details. From ETMv3.3, this information is qualified by the value of the AltISA bit, bit [6], of the continuation byte. The meaning of this bit is shown in Table 7-11.

Table 7-11 Meaning of the AltISA bit in the Continuation byte

State and alignment[a]	AltISA bit	Meaning
ARM, word	0	Processor is in ARM state after the branch
	1	AltISA=1 is Reserved when processor is in ARM state
Thumb, halfword	0	Processor is in Thumb state after the branch
	1	Processor is in ThumbEE state after the branch
Jazelle, byte	0	Processor is in Jazelle state after the branch
	1	AltISA=1 is Reserved when processor is in Jazelle state

a. The instruction set state and alignment are determined by the most significant bits of the fifth byte of the Branch address packet, see Table 7-8 on page 7-26.

Note

From ETMv3.4, if the alternative address compression scheme is implemented, the fifth address byte of a branch packet might not be output. In this case the branch packet does not give an explicit indication of the current state of the processor. However, all five address bytes are always included in the branch packet for a change of processor state, see *Branch address packets for change of processor state* on page 7-31.

Encoding of Exception[3:0], for core architectures other than ARMv7-M

When tracing processor cores that comply with architectures other than the ARMv7-M, branch packets only include a maximum of one Exception information byte, and the exception is described by the Exception[3:0] field. The encoding of this field is given in Table 7-12 on page 7-30.

Note

See *Extended exception handling, from ETMv3.4* on page 7-33 for details of exception tracing for ARMv7-M cores.

Table 7-12 Encoding of Exception[3:0] for non-ARMv7-M cores

Exception[3:0]	Exception
b0000	No exception occurred
b0001	Halting-debug exception
b0010	Secure Monitor Call (SMC)
b0011	Reserved 1
b0100	Asynchronous data abort
b0101	Jazelle exception, see the address for the type
b0110	Reserved 2
b0111	Reserved 3
b1000	Processor reset exception
b1001	Undefined Instruction exception
b1010	Supervisor Call (SVC)
b1011	Prefetch abort or software breakpoint exception
b1100	Synchronous data abort or software watchpoint exception
b1101	Generic exception[a]
b1110	IRQ
b1111	FIQ

a. The Generic exception encoding, b1101, is introduced in ETMv3.3. In previous versions of the ETM architecture this encoding is Reserved 4.

When the processor enters Halting-debug state, the ETM might generate a branch packet indicating a Halting-debug exception. The address in the branch packet is not valid and debuggers must ignore this address. Whether this packet is generated is IMPLEMENTATION SPECIFIC.

Missing components of exception Branch address packets

Not all components of the branch address are output every time. After the branch, the values of any missing components are as shown in Table 7-13.

Table 7-13 Handling of missing Branch address packet components

Component	Handling
High order address bits	Same as previous branch address
Instruction set state	Same as previous branch address
Exception information	No exception
Security level	Same as previous branch address

Branch address packets for change of security state

On processors that support Secure and Non-secure states, the ETM always traces a change of security state by issuing a branch packet with at least one Exception information byte:

- If the original address compression scheme is implemented, a change of security state is traced with a six byte packet, consisting of five address bytes and the Exception information byte. An example of this packet is shown in Figure 7-6 on page 7-15.
- From ETMv3.4, if the alternative address compression scheme is implemented a change of security state is traced with a packet that has between two and five address bytes, followed by the Exception information byte. Examples of these packets are shown in Figure 7-19 on page 7-22 to Figure 7-24 on page 7-24.

Branch address packets for change of processor state

The ETM always traces a change of processor state by issuing a 5 or 6-byte branch packet. These packets are similar to those shown in Figure 7-5 on page 7-15 and Figure 7-6 on page 7-15.

ETMv3.3 introduces support for the Thumb Execution Environment, ThumbEE state. Table 7-14 shows the branch packets for all permitted state changes, and shows which of these changes were supported before ETMv3.3.

Table 7-14 State change packets

State change	Branch packet size	AltISA bit	Notes
ARM to Jazelle, no exception	5 bytes	-[a]	ETMv3.0 or later.
ARM to Jazelle, on an exception	6 bytes	0	ETMv3.0 or later.
ARM to Thumb, no exception	5 bytes	-[a]	ETMv3.0 or later.

Table 7-14 State change packets (continued)

State change	Branch packet size	AltISA bit	Notes
ARM to Thumb, on an exception	6 bytes	0	ETMv3.0 or later.
ARM to ThumbEE[b]	6 bytes	1	From ETMv3.3 only
Jazelle to ARM, no exception	5 bytes	-[a]	ETMv3.0 or later.
Jazelle to ARM, on an exception	6 bytes	0	ETMv3.0 or later.
Thumb to ARM, no exception	5 bytes	-[a]	ETMv3.0 or later.
Thumb to ARM, on an exception	6 bytes	0	ETMv3.0 or later.
Thumb to ThumbEE[b]	6 bytes	1	From ETMv3.3 only
ThumbEE to ARM[b]	6 bytes	0	From ETMv3.3 only
ThumbEE to Thumb[b]	6 bytes	0	From ETMv3.3 only

a. There is no AltISA bit in 5-byte branch packets.
b. With these state changes, the Exception field in the sixth byte of the branch packet (see Figure 7-6 on page 7-15) indicates whether the state-change branch was because of an exception. If there was no exception the value of the Exception field is 4'b0000.

Changes of state that are not indicated explicitly

Not all state changes require a branch packet to be output in the trace, because a direct branch instruction can cause a state change. When such a state change occurs, subsequent branch packets are output with all the bits that have changed since the last broadcast address output in either an I-Sync packet or a branch packet. However, there are cases where a state change is not indicated in the trace.

Table 7-15 shows a sequence of instructions including a direct branch causing a change from ARM to Thumb state:

Table 7-15 Direct branch with change from ARM to Thumb state

Step	Instruction	State (change)	Trace generated
1.	…	ARM state	Trace enabled, I-Sync packet generated indicating ARM state.
2.	`BLX #immed`	(to Thumb state)	No branch packet generated, because this is a direct branch.
3.	`MOVS PC`	(to ARM state)	Indirect branch packet generated.

In this example, Thumb state is not indicated explicitly in the trace stream, because the state change is caused by a direct branch instruction. The branch packet generated at step 3 does not have to be a full 6-byte packet, because the most recent broadcast address, in the I-Sync packet generated at step 1, indicated ARM state.

Table 7-16 shows a sequence of instructions that include changes between Thumb and ThumbEE states:

Table 7-16 Direct branch with changes between Thumb and ThumbEE states

Step	Instruction	State (change)	Trace generated
1.	…	ARM state	Trace enabled, I-Sync packet generated indicating ARM state.
2.	MOVS	(to ThumbEE state)	6-byte branch packet generated indicating ThumbEE state[a].
3.	LEAVEX	(to Thumb state)	No branch packet generated, because this is a direct branch.
4.	MOVS	(to ThumbEE state)	Indirect branch packet generated but does not require 6 bytes.

a. Byte 5 of the packet indicates Thumb state, and the AltISA bit in byte 6 is set to 1.

In this case, the entry to Thumb state at step 3 trace stream is not indicated explicitly in the trace stream, because the state change is caused by a direct branch. The branch packet generated at step 4 does not have to be a full 6-byte packet, because the state entered is the same as that indicated in the most recent broadcast address, that was in the branch packet generated at step 2.

Extended exception handling, from ETMv3.4

The ARMv7-M processor architecture includes a number of features that affect the tracing of exceptions, in particular:

- The architecture supports up to 512 exceptions. These comprise 15 standard exceptions plus up to 496 interrupts.
- Some processor instructions can be *paused for continuation* when an interrupt is received. Processing of a paused instruction is *resumed* on return from the exception.

To permit tracing of these features, ETMv3.4 introduces extensions to the exception branch packets.

Note

These extensions are backwards compatible, and apply regardless of which option for branch address encoding is implemented by the ETM. For details of the two options for branch address encoding, see:

- *Branch address compression, original scheme* on page 7-16
- *Branch packet formats with the alternative address compression scheme* on page 7-18.

The features of the extended exception handling are:

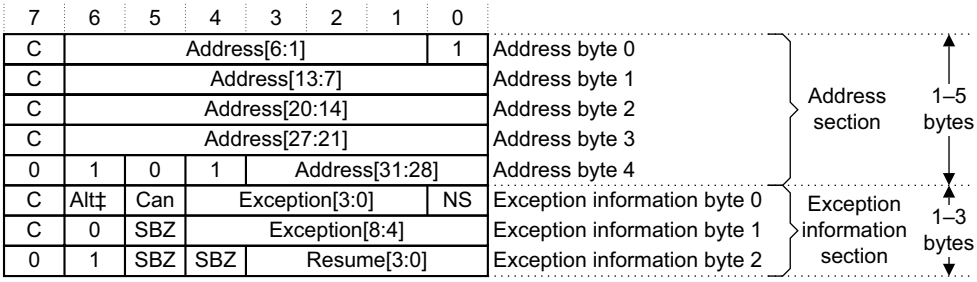
- ETMv3.3 and earlier permits a single *Exception information* byte, for exception and state information. From ETMv3.4, up to three continuation bytes are permitted, and these are referred to as *Exception information bytes* 0 to 2.
- In ETMv3.3, the exception type is encoded in a four bit field, Exception[3:0]. From ETMv3.4 this field can be extended up to nine bits, to give Exception[8:0].

- ETMv3.4 introduces a 4-bit execution resumption field, Resume[3:0].

Figure 7-25 shows a maximum-sized extended exception branch packet, for an exception in Thumb state.

Note

Extended exception branch packets are only generated by processors that conform to the ARMv7-M architecture.



Alt‡ = AltISA bit

C bits are 1 if another byte follows in that section, and 0 for the last byte of a section

Figure 7-25 Extended exception branch packet, shown for Thumb state

In the Exception information section of the packet, the ETM only outputs the bytes that contain nonzero information. This means that:

- if Resume[3:0] is not output its value is assumed to be b0000
- If Excp[8:4] is not output its value is assumed to be b00000.

The rules for the generation of the extended exception branch packet are:

For the Address section:

- If the ETM implements the original scheme for branch address compression, branch addresses for exception packets are never compressed. The Address section of the extended exception branch packet is always all five bytes, as shown in Figure 7-25.
- If the ETM implements the alternative scheme for branch address compression, the address section is compressed as described in *Exception branch packets in the alternative encoding* on page 7-22, and is between two and five bytes long.

Bit [20] of the ETM ID Register indicates which branch address compression scheme is implemented, see *ETM ID Register, ETMv2.0 and later* on page 3-71.

For the Exception information section:

Exception information byte 0 is always output:

- If this is the only Exception information byte output then the C bit is 0, otherwise the C bit is 1.

- If the exception number is 15 or less (b1111 or less) then the value in Exception[3:0] is used to identify the exception, by reference to Table 7-17. In this case, Exception information byte 1 is not output, and Exception[8:4] = b00000.

Exception information byte 1 is only output if the exception number is 16 or greater (b000010000 or greater). If Exception information byte 1 is output:

- If Exception information byte 2 is not output then the C bit is 0, otherwise the C bit is 1.
- The value of Exception[8:0] is used to identify the exception, by reference to Table 7-17.

Exception information byte 2 is only output if the instruction being traced was paused for continuation:

- The value of Resume[3:0], together with the value of the Can bit from Exception information byte 0, gives information about resuming execution, see Table 7-18 on page 7-37.

Note

- When an extended exception branch packet with two Exception information bytes is output, the last byte can be either Exception information byte 1 or byte 2. A debugger can tell which byte it is by checking bit [6]:
  - for Exception information byte 1, bit [6] = 0
  - for Exception information byte 2, bit [6] = 1.
- When only one Extended Information byte is output and the ETM implements the original scheme for branch address compression, the format of the extended exception branch packet is identical to the 6-byte branch packet with extension byte, shown in Figure 7-6 on page 7-15. However, Exception[3:0] must be interpreted by reference to Table 7-17, instead of Table 7-10 on page 7-28. A debugger must know the architecture of the processor it is debugging to know how to interpret Exception[3:0] for a packet in this format. Table 7-10 on page 7-28 corresponds to ARMv7-M processors only.

Table 7-17 Encoding of Exception[8:0] for ARMv7-M processors

Exception[8:0][a]	Meaning	Exception[8:0][a]	Meaning
b000000000 (0x000)	No exception	b000010010 (0x012)	Reserved
b000000001 (0x001)	IRQ1	b000010011 (0x013)	HardFault
b000000010 (0x002)	IRQ2	b000010100 (0x014)	Reserved
b000000011 (0x003)	IRQ3	b000010101 (0x015)	BusFault
b000000100 (0x004)	IRQ4	b000010110 (0x016)	Reserved
b000000101 (0x005)	IRQ5	b000010111 (0x017)	Reserved

Table 7-17 Encoding of Exception[8:0] for ARMv7-M processors (continued)

Exception[8:0][a]	Meaning	Exception[8:0][a]	Meaning
b000000110 (0x006)	IRQ6	b000011000 (0x018)	IRQ8
b000000111 (0x007)	IRQ7	b000011001 (0x019)	IRQ9
b000001000 (0x008)	IRQ0	b000011010 (0x01A)	IRQ10
b000001001 (0x009)	UsageFault	b000011011 (0x01B)	IRQ11
b000001010 (0x00A)	NMI	b000011100 (0x01C)	IRQ12
b000001011 (0x00B)	SVC	b000011101 (0x01D)	IRQ13
b000001100 (0x00C)	Debug Monitor	b000011110 (0x01E)	IRQ14
b000001101 (0x00D)	MemManage	b000011111 (0x01F)	IRQ15
b000001110 (0x00E)	PendSV	b000100000 (0x020)	IRQ16
b000001111 (0x00F)	SysTick	...	...
b000010000 (0x010)	Reserved	b111111110 (0x1FE)	IRQ494
b000010001 (0x011)	Processor reset	b111111111 (0x1FF)	IRQ495

a. To make it easier to relate this table to the diagram of the packet in Figure 7-25 on page 7-34, the binary values in this column are shown with a space between the Exception[8:4] and Exception[3:0] sub-fields.

Note

When tracing an ARMv7-M core, a branch packet might only include a single Exception information byte. However, in this case, the exception is described by Exception[8;0], with:

- Exception[8:5] = b0000
- Exception[3:0] defined by the Exception information byte
- The encoding of Exception[8:0] is defined in Table 7-17 on page 7-35.

Table 7-18 Encoding of the Can bit and Resume[3:0]

Can bit[a]	Resume[3:0][b]	Meaning
0	b0000	Last instruction completed.
0	> b0000	The next instruction is resumed from earlier. Resume[3:0] encodes the last successfully completed transfer. In this case, Exception[8:0] = b000000000.
1	b0000	Last instruction was cancelled.
1	> b0000	The last instruction was paused for continuation. Resume[3:0] encodes the last successfully completed transfer. In this case, Exception[8:0] must be nonzero.

a. Bit [5] of Exception information byte 0.
b. Bits [3:0] of Exception information byte 2.

Possible forms of the Exception information section

The rules for generating the Exception information section of an extended exception branch packet, and the additional architectural requirements given in Table 7-18, mean that only certain combinations of the Exception information bytes are possible. This section describes each possibility, and the circumstances in which it is output.

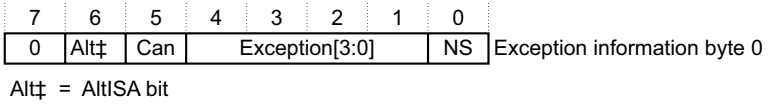


Figure 7-26 Only Exception information byte 0 is output

There are two situations where only Exception information byte 0 is output:

- If there is no exception and no resumption of a *paused for continuation* instruction. In this case Exception[3:0] is zero and the Can bit is zero.
- If an exception with exception number of 15 or less occurs, without pausing an instruction. This can occur with or without cancellation of the previous instruction:
  - Exception[3:0] holds the exception number, see Table 7-17 on page 7-35
  - Can is set to 1 if the previous instruction was cancelled.

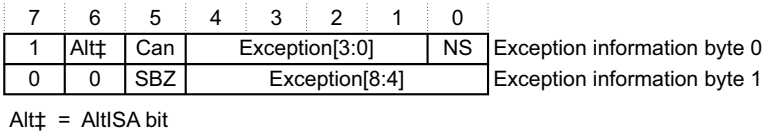


Figure 7-27 Only Exception information bytes 0 and 1 are output

There is only one situation where only Exception information bytes 0 and 1 are output:

- If an exception with exception number greater than 15 occurs without the pausing of an instruction. This can occur with or without cancellation of the previous instruction:
  - Exception[8:0] holds the exception number, see Table 7-17 on page 7-35
  - Can is set to 1 if the previous instruction was cancelled.

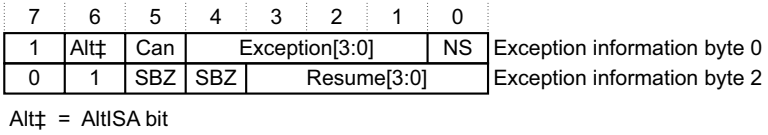


Figure 7-28 Only Exception information bytes 0 and 2 are output

There are two situations where only Exception information bytes 0 and 2 are output:

- If an exception with exception number of 15 or less occurs, and an instruction is paused:
  - Exception[3:0] is nonzero, and holds the exception number, see Table 7-17 on page 7-35
  - Can is set to 1.
- If a *paused for continuation* instruction is resumed:
  - Exception[3:0] is zero, because there is no exception
  - Can is set to 0.

Note

No exception occurs in this situation. However, it is included here because it is traced in the same way as exceptions.

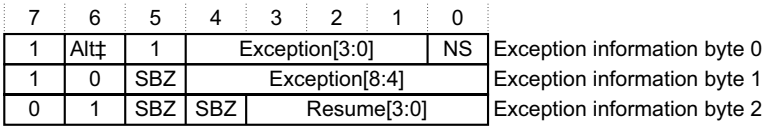


Figure 7-29 All Exception information bytes are output

There is only one situation where all three Exception information bytes are output:

- If an exception with exception number greater than 15 occurs, and an instruction is paused:
  - Exception[8:0] is nonzero, and holds the exception number, see Table 7-17 on page 7-35
  - Can is set to 1.

Extended exception handling in Instruction-only trace

When performing instruction-only tracing, and an instruction is paused for continuation, no trace information is required to indicate where the instruction is resumed. With instruction-only trace:

- if the previous instruction completed:
  - Can = 0
  - Resume[3:0] = b0000
- if the previous instruction is paused for continuation, or restarted, it is treated as if the instruction is cancelled:
  - Can = 1
  - Resume[3:0] = b0000.

This means that, with instruction-only trace, Exception information byte 2 is never output.

Branch address generation

This section describes how a Branch address packet is produced. The sequence is shown in:

- Figure 7-30 on page 7-40 for ARM addresses
- Figure 7-31 on page 7-41 for Thumb addresses
- Figure 7-32 on page 7-42 for Jazelle addresses.

These examples only describe the generation of a complete 5-byte address packet. However, the descriptions of address compression given in this chapter indicate how the 5-byte address is compressed to give shorter address packets when this is appropriate.

The Branch address packet is constructed using the following rules:

1. The address is prefixed with a one in the position of bit [33]. The decompressor uses the position of this bit in the final address to identify whether the code is currently in ARM, Thumb, or Jazelle state.
2. If the destination is an ARM address, it is shifted right by two bits (ARM addresses are word-aligned so the first two bits are always zero).

   If the destination is a Thumb address, it is shifted right by one bit (Thumb addresses are halfword-aligned so the first bit is always zero).

   If the destination is a Jazelle address, it is not shifted (Jazelle addresses are byte-aligned).
3. A one is added to the low end of the packet. This identifies the packet as a Branch address.
4. The value produced (at most 34 bits wide) is divided into 7-bit quantities.

5. An address continue bit is added to each 7-bit fragment. This bit is set to 1 in all but the last byte of the address packet.

This encoding mechanism means that ARM, Thumb, and Jazelle addresses can always be uniquely identified by the high-order bits of the fifth address byte.

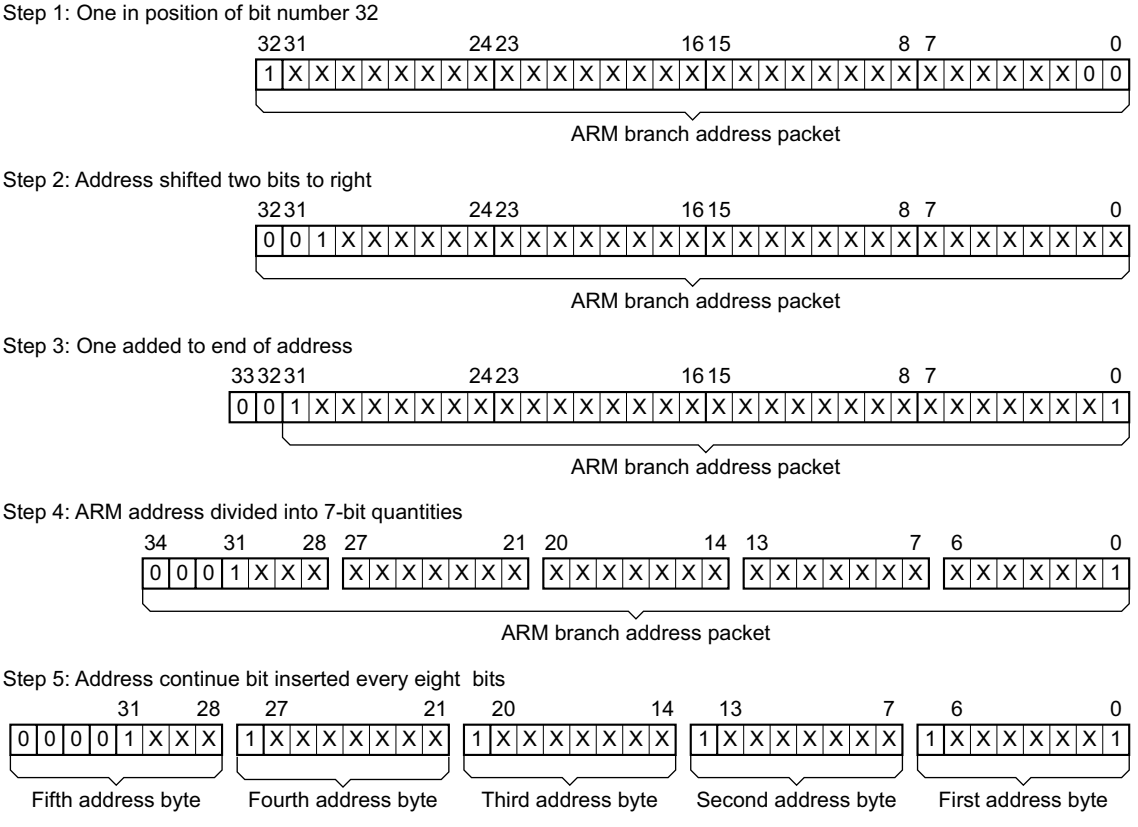


Figure 7-30 Generating an ARM branch address

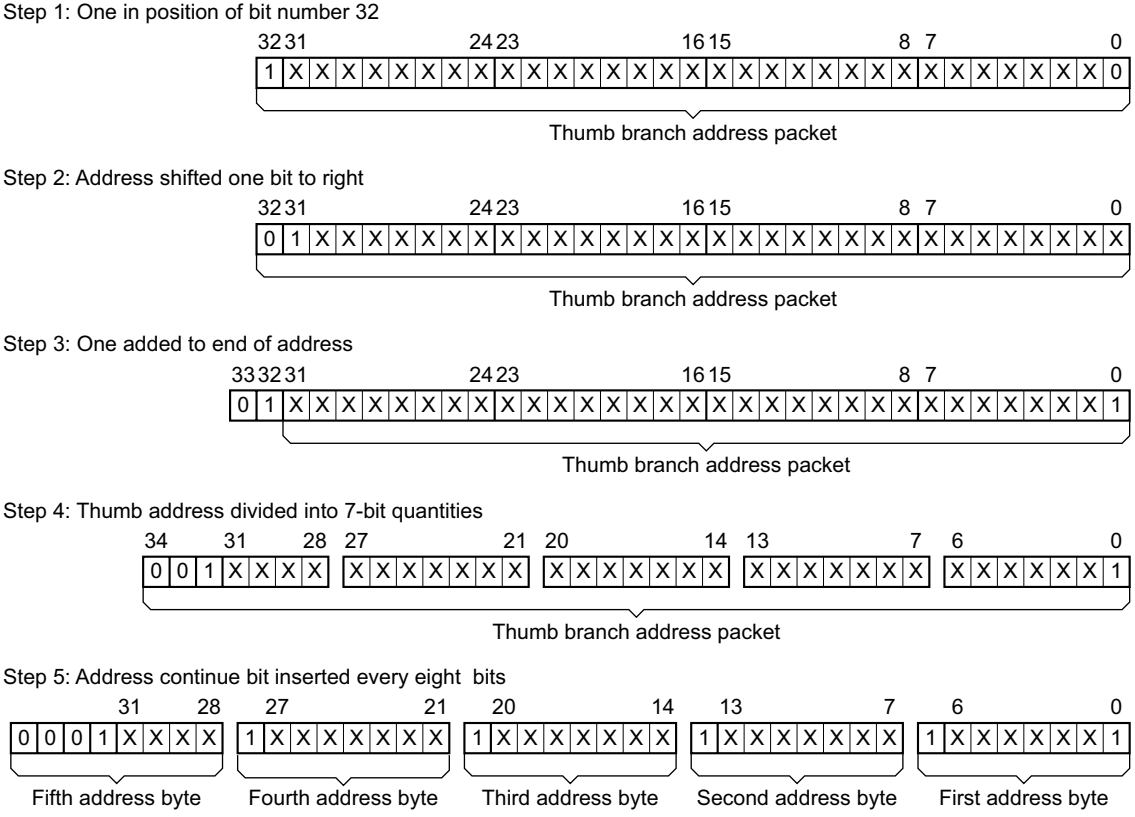


Figure 7-31 Generating a Thumb branch address

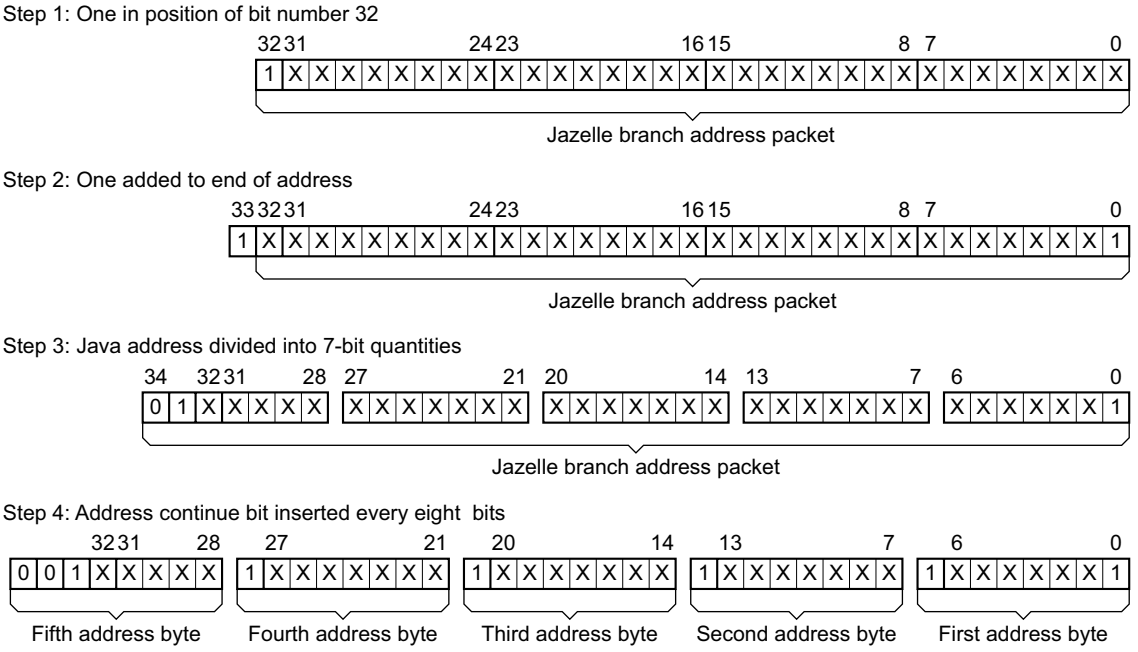


Figure 7-32 Generating a Jazelle branch address

7.3.6 Context ID packets

When the Context ID changes, a Context ID packet is output to give the new value. It comprises the following components:

- Context ID packet header (1 byte)
- Context ID (1-4 bytes).

Figure 7-33 shows a Context ID packet.

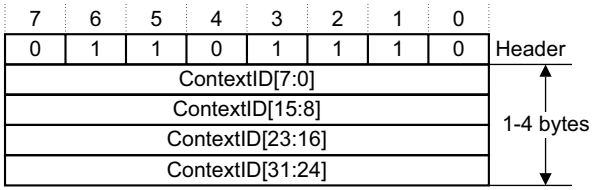


Figure 7-33 Context ID packet

The number of bytes output depends on the ContextIDSize bits, bits [15:14] of the ETM Control Register, `0x00`, see *ETM Control Register* on page 3-20. If Context ID tracing is disabled because these bits are set to b00, Context ID packets are never generated.

If the Context ID is changed by a data transfer that would normally have been traced, and a Context ID packet is output, it is IMPLEMENTATION SPECIFIC whether the Context ID packet is generated instead of or in addition to the normal trace. As a result, when Context ID tracing is enabled, data trace might be missing for an instruction that changes the Context ID.

The Context ID packet is output:

- after tracing all instructions up to the point where the Context ID is changed
- before tracing any instructions that are executed with the new Context ID.

7.4 Data tracing

Data tracing is performed by interleaving packets of data trace with the instruction trace. Most data packets correspond to the most recent *Data instruction* traced. See *Data Instructions* on page 4-22 for definitions of Data Instructions and *Exceptions on Data Instructions* on page 7-55 for special handling around exceptions.

Note

When tracing `LSM` instructions, the ETM generates one data packet for each 32-bit data transfer. For more information see *Tracing LSMs* on page 7-49.

Data tracing features are controlled by the ETM Control Register. For more information, see *ETM Control Register* on page 3-20 and in particular the descriptions of the following bits and fields:

- Data-only mode, bit [20]
- Filter (CPRT), bit [19], and MonitorCPRT, bit [1]
- Suppress data, bit [18]
- Data access, bits [3:2].

From ETMv3.3, it is IMPLEMENTATION DEFINED which data tracing features are provided. For details of the implementation options see *Data tracing options, ETMv3.3 and later* on page 7-54.

This document refers to data tracing being enabled. Data tracing is enabled if at least one of the following is enabled:

- data address tracing
- data value tracing
- CPRT tracing.

7.4.1 Data packet types

Data packets can be any of the following types:

Normal data packet

Used where the data values can be output in order. For more details, see *Normal data packet* on page 7-45.

Out-of-order packets

Used where the data values cannot be output in order. For more details, see *Out-of-order packets* on page 7-47.

Value not traced packet

Used when performing partial tracing of an `LSM` instruction. For more details, see *Value not traced packet* on page 7-50.

Data suppressed packet

Used to prevent FIFO overflow.

Store failed packet

Used to indicate failure of an exclusive store operation.

7.4.2 Normal data packet

The Normal data packet is used for all loads, stores, and CPRT packets that can be output in order. For more information about tracing LSMs see *Tracing LSMs* on page 7-49.

The ETM compresses the data address trace by reducing the number of bits that are output for the address of the data transfer. The same technique is used as for Branch addresses, where a copy of the last data access address is kept and only the low-order bits that have changed are output for the next address. This is particularly effective, for example, if you are viewing data in one small address range, because all the traced data accesses have the same high-order address bits.

A Normal data packet comprises the following contiguous components:

Normal data packet header

Output first. Always present.

Data address Present if both of the following conditions are satisfied:

- data address tracing is enabled in the ETM Control Register
- the A bit is set to 1 in the header.

Data addresses consist of one to five bytes. To enable the decompressor to detect the last byte, bit [7] of each byte is set to 1 if there are more address bytes to follow. Bit [7] is LOW in the last address byte. Whether or not data addresses are traced must be statically determined before tracing begins.

The data address is compressed relative to the last traced data address, in a similar manner to instruction addresses. If a data address of any length is traced, bits that are not output are the same as those in the last traced data address.

Data value Present only if data value tracing is enabled in the ETM Control Register.

Normal data packets correspond to the most recently-traced *data instruction*. This is to support cores where instructions that do not perform a data transfer might execute before a previous transfer completes.

A Normal data packet is shown in Figure 7-34 on page 7-46.

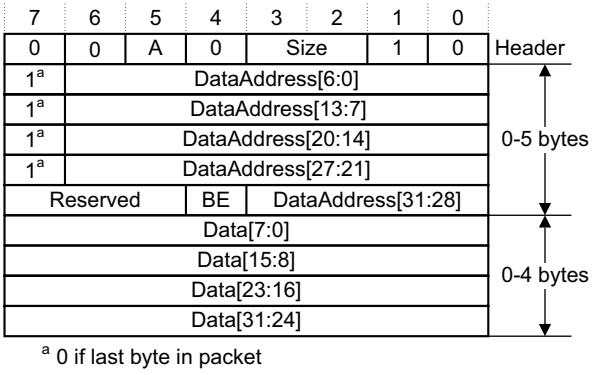


Figure 7-34 Normal data packet for ETMv3.0 and later

The A bit

The A bit of the Normal data packet header shows that a data address is expected if address tracing is enabled. It is set to 1 for the first data packet output for an LSM, and can be set to 1 for subsequent data packets if their addresses are noncontiguous. In ETMv3.0 a new address must be output if it is noncontiguous and the processor is in Jazelle state. In ETMv3.1 and later this applies to ARM, Thumb, and Jazelle states.

Note

- In ETMv3.0 you must take account of the instruction type when determining data addresses. This is because the `SWP` and `SWPB` instructions perform two accesses to the same address, but only the first has an address output.
- The A bit can be set to 1 even if address tracing is disabled. In this case the A bit must be ignored and an address is not present.

BE bit

The BE bit shows that the data was a BE-8 (ARM architecture v6 and later) big-endian transfer, and that the bytes must be reversed to determine the value that was stored in memory. It represents the state of the E bit in the CPSR at the time of the transfer. See *Endian effects and unaligned access* on page 4-19.

The BE bit is traced regardless of whether data value tracing is enabled. However, if a data address is traced and this bit is not output because the address is output in less than 5 bytes, then the value of the BE bit is the same as the value given in the last 5-byte data address traced. If the value of the BE bit changes then a full 5-byte data address is output.

Size bits

The size bits are used for data value compression. They specify the size of the transferred data value. Leading zeros are removed from the value as a simple form of this compression. The encoding combinations of the size bits are listed in Table 7-19.

Table 7-19 Size bit encoding combinations

Encoding	Description
b00	Value = 0, no data value bytes follow
b01	Value < 256, one data value byte follows
b10	Value < 65536, two data value bytes follow
b11	No compression done, four data value bytes follow

7.4.3 Out-of-order packets

ETMv3.x supports cores with non-blocking data caches. A non-blocking data cache enables instructions, including data instructions, to execute underneath an outstanding data transfer. This means that the data cache can return data to the core out-of-order.

This behavior is handled by the Out-of-order placeholder and Out-of-order data header types. These are described in:

- *Out-of-order placeholder*
- *Out-of-order data* on page 7-48.

For more information about tracing LSMs see *Tracing LSMs* on page 7-49.

Out-of-order placeholder

When data cannot be traced in order, an Out-of-order placeholder packet is placed in the FIFO instead of a Normal data packet. Figure 7-35 shows an Out-of-order placeholder packet.

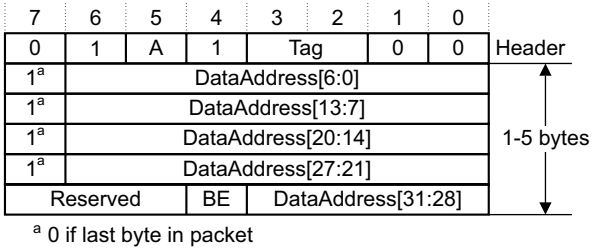


Figure 7-35 Out-of-order placeholder packet

When an Out-of-order placeholder packet is read, the decompression software must identify the data value as an outstanding value. This outstanding value is returned later in the trace.

The A bit

The A bit indicates that a data address is present if data tracing is enabled. See *The A bit* on page 7-46 for more information.

Tag bits

The tag bits are used to identify each load miss. Values b01, b10, and b11 are supported. Encodings corresponding to a tag of b00 correspond to other packets in the header space.

BE bit

The BE bit indicates that the data was a BE-8 transfer. See *BE bit* on page 7-46 for more information.

Out-of-order data

When out-of-order data is returned, the Out-of-order data packet, comprising the Out-of-order data header byte and the data value, is placed in the FIFO.

An Out-of-order data packet never includes a data address.

An Out-of-order data packet is shown in Figure 7-36.

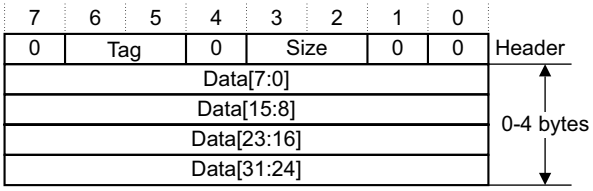


Figure 7-36 Out-of-order data packet for ETMv3.0 and later

An Out-of-order data packet always corresponds to the most recent Out-of-order placeholder packet with the same TT tag value, with the exception of 64-bit values. See *64-bit values* on page 7-49 for more information.

If the decompressor receives an unexpected Out-of-order data packet (that is, an Out-of-order data packet is given without a pending Out-of-order placeholder packet with the same TT tag), it must be ignored. If trace is disabled before the outstanding out-of-order data is returned, this data item is placed in the FIFO as soon as it is available.

Note

In ETMv3, all forms of data, loads, stores, and CPRTs, can be returned out-of-order. In ETMv2, only loads can be returned out-of-order.

Rules for generation of Out-of-order packets

These rules do not affect decompression, but describe how the ETM handles out-of-order trace packets.

Out-of-order placeholder packets are placed in the FIFO if **TraceEnable** and **ViewData** are active at the time that the Out-of-order placeholder is generated, in the same way as Normal data packets.

Out-of-order data packets are placed in the FIFO if and only if the corresponding Out-of-order placeholder packet was traced, with the following exceptions:

- Out-of-order data might be missing following overflow, if the Out-of-order placeholder packet was placed in the FIFO before the overflow occurred.
- Out-of-order data might be missing following restart from debug, if the Out-of-order placeholder packet was placed in the FIFO before the entry to debug state.
- Out-of-order data might be missing following a processor reset, if the Out-of-order placeholder packet was placed in the FIFO before the reset occurred.
- Out-of-order data might be missing if it is returned in the same cycle as a Non-periodic I-sync. This is because of FIFO bandwidth limitations. A periodic I-sync must be delayed if it would cause the loss of an Out-of-order data packet.

Out-of-order data packets are never output without a corresponding Out-of-order placeholder packet. However, lone Out-of-order data packets might be observed at the beginning of the captured trace, because of the loss of the original Out-of-order placeholder packets.

64-bit values

When a miss occurs on a 64-bit value, two out-of-order packets are placed in the FIFO in the same cycle. The decompressor must recognize that these two misses are for a single 64-bit value because both packets have the same tag value and they are consecutive. As with Normal data packets, the data address is present only with the first Out-of-order placeholder packet, and is not present at all if the miss occurs in the middle of an LSM that has already output data packets.

When 64-bit out-of-order data is returned, it is always returned as two separate Out-of-order data packets given in the same cycle. Both packets have the same tag, and are consecutive.

7.4.4 Tracing LSMs

When the first data transfer associated with an LSM is traced, a Normal data packet, or an Out-of-order placeholder packet, is placed in the FIFO. If data address tracing is enabled this packet includes the data address. All subsequent data transfers for that LSM place a packet in the FIFO according to the following rules:

- If a subsequent data transfer is to be traced, a Normal data packet, or Out-of-order placeholder packet is traced, normally without an address. See *The A bit* on page 7-48 for more information.
- If a subsequent data transfer is not to be traced, a Value not traced packet is placed in the FIFO for that transfer, see *Value not traced packet* on page 7-50.

- When data address tracing is enabled, the trace packets for all of the subsequent data transfers do not normally include the data address, because the addresses of the LSM transfers are sequential.

  Exceptions to this are:

  — For a SWP or SWPB instruction where the addresses for the two data transfers are not sequential.

  — On some processors, for an LDM instruction that includes the PC in its registers list. Some processors transfer the PC before the other addresses, and this results in data transfers with non-sequential addresses.

  If data address tracing is enabled, and a data transfer for an LSM is not sequential to the previous transfer for that LSM, the trace packet for the transfer includes the data address.

A compiler can combine adjacent loads or stores into an LSM to speed up execution. In ETMv1, data tracing can be enabled only at the beginning of a *Load/Store Multiple* (LSM) instruction. ETMv2 and ETMv3 can partially trace an LSM and output only the data values that match the filtering criteria. For example, the **ViewData** setting might match only from the third word of the LSM. In this case this third transfer is the first transfer traced for the LSM, and if data address tracing is enabled the trace packet includes the data address of the third word of the LSM.

For more information about tracing LSMs in data-only mode see *Tracing LSM instructions in data-only mode* on page 7-53.

See *Load/Store Multiple (LSM) instructions* on page 4-21 for a list of the LSM instructions.

7.4.5 Value not traced packet

Value not traced packets comprise a Value not traced header byte, followed by an optional data address as shown in Figure 7-37. The decompression software must work backwards from the final data transfer, using the Value not traced packets in combination with the Normal data packets, to determine which of the LSM values were traced.

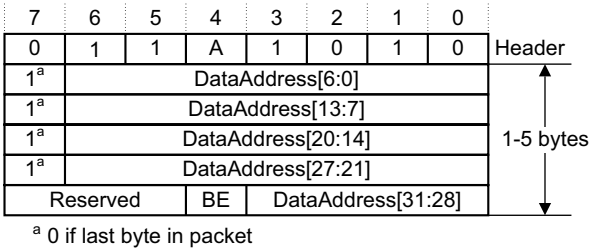


Figure 7-37 Value not traced packet

The data address is output if the address of the transfer is noncontiguous relative to the previous transfer, so that the address of subsequent data transfers in the LSM can be determined if they are traced. It therefore follows the same rules as for Normal data packets. See *The A bit* on page 7-46 for more information.

Note

If an LSM instruction is traced, instruction tracing continues until the LSM completes, even if **TraceEnable** is deasserted before the LSM completes. This means that instructions executed under an LSM are also traced, regardless of whether **TraceEnable** remains asserted, see *Independent load/store unit* on page 2-71.

7.4.6 Data suppressed packet

If enabled, the ETM prevents the output of data trace when the number of free bytes in the FIFO drops below the level set in the FIFOFULL Level Register, 0x0B, by activating the **SuppressData** signal. The following packet types are suppressed when **SuppressData** is asserted:

- Normal data
- Out-of-order data
- Out-of-order placeholder
- Value not traced.

The first such data packet to be generated while data suppression is activated is replaced by a Data suppressed packet. Figure 7-38 shows a Data suppressed packet. Subsequent data packets are deleted entirely and generate no trace while data suppression is still active. Data suppression remains active until a data packet is generated while **SuppressData** is deasserted.

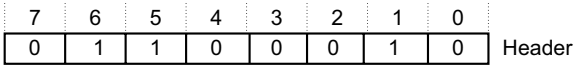


Figure 7-38 Data suppressed packet

Data suppression does not occur and no Data suppressed packet is output if **SuppressData** is asserted and deasserted without the suppression of any data packets. This happens if the number of free bytes in the FIFO briefly drops below the FIFOFULL level during cycles when no data tracing occurs.

If data suppression occurs during a Data Instruction, data suppression must continue until the Data Instruction has completed. Because no Value not traced placeholder packets are output, and data tracing might not have started on the first transfer, it is not possible to determine which transfers in the Data Instruction were traced. As a result you might have to discard all transfers corresponding to the Data Instruction that were traced before the Data suppressed packet.

In ETMv3.2 and earlier, D-sync is required following a restart from data suppression. In other words, the first data address output must be a full 5-byte address. This resynchronization is not required in ETMv3.3 and later.

Synchronization is delayed while **SuppressData** is asserted. This applies to:

- I-sync
- A-sync.

In rare cases this can cause an overflow. See *Synchronization* on page 7-65 for more information.

7.4.7 Store failed packet

ARMv6 supports a new instruction, STREX, that might or might not succeed in storing its value. The trace must indicate if this has been unsuccessful. Figure 7-39 shows a Store failed packet.

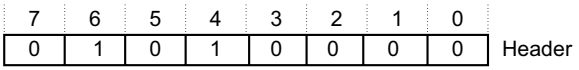


Figure 7-39 Store failed packet

This packet is output immediately following the Normal data or Out-of-order data packet, and indicates that the most recent data transfer was a failed STREX. In other words, no other packets are output between the Normal data or Out-of-order data packet and the Store failed packet. This packet is only output if data value tracing is enabled by setting bit [2] of the ETM Control Register, 0x00, to 1. The data value traced is 0. The data address is output as normal if appropriate.

7.4.8 Jazelle data tracing

Loads and stores that are considered to be useful are traced. This excludes the following:

- stack spills and fills
- array base pointer loads
- array size loads
- loads whose sole purpose is to perform a null-pointer check
- loads from the constant pool used by quicker bytecodes.

The precise list of traced data transfers is IMPLEMENTATION DEFINED. See the appropriate *Technical Reference Manual* for details.

7.4.9 Data aborts

If one or more of the data transfers for a data instruction is aborted by the memory system, a branch address to the Data abort exception vector is traced, indicating that a data abort exception has occurred. It is IMPLEMENTATION SPECIFIC whether the data instruction or data transfers are traced. If the instruction is traced, the branch packet indicates that the instruction is cancelled and all data transfers traced for this instruction must be discarded.

For information about specifying comparator behavior when data aborts occur, see *Exact matching for data address comparisons* on page 2-46.

Asynchronous data aborts

Asynchronous data aborts are so named because the data abort handler cannot determine the instruction that caused the abort, and must therefore usually terminate the entire process. Similarly, it is not possible to determine which instruction caused the asynchronous data abort from the trace.

Asynchronous data aborts can be traced as canceling or not canceling.

Note

In previous versions of this document:

- synchronous aborts were described as precise aborts
- asynchronous aborts were described as imprecise aborts.

7.4.10 Data-only mode, ETMv3.1 and later

Bit [20] of the ETM Control Register, `0x00`, enables instruction trace to be disabled while continuing to output data trace. This is useful if you want to trace updates to a particular data value or a selection of values, but do not have to trace the instruction that caused the update.

For example, if tracing the value of a single word, only 5 bytes are required (the header must be traced, to enable synchronization). The actual requirement is I-sync, a P-header and the data itself, a total of 12 bytes. Therefore, the size of the data traced is over twice what it is expected to be. This proportion is even bigger if the values are suitable for compression-byte values are 2 bytes instead of 9 bytes.

When data-only trace is enabled, only the following header types are produced:

- A-sync
- Normal data
- Out-of-order placeholder
- Out-of-order data
- I-sync (without instruction addresses)
- Context ID
- Trigger.

It is IMPLEMENTATION DEFINED whether Value not traced packets are produced in data-only mode, see *Tracing LSM instructions in data-only mode*.

All other packet types, including branches, P-headers, and Data suppressed, are suppressed. D-sync (periodic trace of a full data address) continues as normal.

From ETMv3.3, data-only mode is only implemented on macrocells that provide a full data tracing implementation. See *Data tracing options, ETMv3.3 and later* on page 7-54 for more information.

Tracing LSM instructions in data-only mode

When an ETM implementation supports data-only mode, it is IMPLEMENTATION DEFINED how `LDM` and `STM` instructions are traced in data-only mode when **Viewdata** is not active for all of the data transfers generated by the instruction. However, one of the following trace sequences must be implemented:

- A Value not traced packet is generated for any word of the transfer for which **ViewData** is not active. See *Value not traced packet* on page 7-50.
- An address packet is output for each traced data transfer for which the data address is not sequential to the previous traced data transfer.

See *Load/Store Multiple (LSM) instructions* on page 4-21 for a list of the LSM instructions.

Possible wrong interpretation of CPRT trace in data-only mode

When you are tracing in data-only mode with data address tracing enabled and CPRT tracing enabled, no addresses are output with any CPRT transfer. This means that, if a CPRT transfer follows immediately after one or more LSM transfers in the trace stream there is nothing in the trace to distinguish the CPRT transfer from the LSM transfers. Therefore, a trace decompression tool might incorrectly associate a CPRT transfer with the address of an earlier LSM transfer.

7.4.11 Data tracing options, ETMv3.3 and later

From ETMv3.3, an ETM implementation can limit the availability of data value and data address tracing. From ETMv3.3, the availability of the following data trace options is IMPLEMENTATION DEFINED:

- data address tracing
- data value tracing
- CPRT tracing
- data-only mode.

However, these options are not independent, and any implementation must provide one of the feature sets listed in Table 7-20.

Table 7-20 Possible feature sets for data tracing, ETMv3.3 and later

Data address tracing	Data value Tracing	CPRT tracing	Data-only mode	Notes
Implemented	Implemented	Implemented	Implemented	As for ETMv3.1 and ETMv3.2
Implemented	Not implemented	Not implemented	Not implemented	-
Not implemented	Implemented	Implemented	Not implemented	-
Not implemented	Not implemented	Not implemented	Not implemented	ViewData registers not implemented.

Note

- Data-only mode, as described in *Data-only mode, ETMv3.1 and later* on page 7-53, is only implemented when all other data tracing features are implemented.
- Context ID tracing, as described in *Context ID packets* on page 7-43, is always available and is not affected by any restriction on the data tracing options that are implemented.
- If an implementation does not provide any of the optional data tracing features then the ViewData registers are not implemented, and reads as zero.

Detecting which data tracing options are available

Debug tools can write and then read the ETM Control Register to find which data tracing options are supported. For details, see *Checking which data tracing options are available, ETMv3.3 and later* on page 3-28.

7.4.12 Exceptions on Data Instructions

If a Data Instruction is canceled by a canceling exception, all data traced with it might be invalid and must be discarded.

7.5 Additional trace features for ARMv7-M cores, from ETMv3.4

The ARMv7-M processor architecture introduces a number of features that require the ETM trace protocol to be extended. ETMv3.4 introduces changes that permit the tracing of these features.

The ARMv7-M architecture features that require additions to the ETM protocol are:

- Support for up to 512 exceptions, see *Support for a large number of exceptions*.
- Some instructions can be *paused for continuation* when interrupted. Processing of a paused instruction is resumed on return from the interrupt. The ETM must be able to:
  - — indicate that an instruction has been paused for continuation
  - — trace the resumed processing of the instruction.

  For more information see *Instructions that can be paused for continuation*.
- The stack is pushed automatically on entry to an exception, and popped on return from the exception handler, see *Automatic stack push on exception entry and pop on exception exit* on page 7-57.
- Return from an exception can be performed by the ARMv7-M-specific implementations of certain instructions, see *Tracing return from an exception* on page 7-59.

Some of the ETMv3.4 changes to provide support for ARMv7-M cores are described in *Branch packets* on page 7-11. These changes are cross-referenced from this section. The other ETMv3.4 changes to support these cores are described fully in this section.

7.5.1 Support for a large number of exceptions

The ARMv7-M architecture supports 15 standard exceptions and up to 496 interrupts, controlled by an NVIC. ETMv3.4 introduces an extension to the *branch with exception* packet format, that permits each of these exceptions to be identified and traced uniquely. This extension is described in *Extended exception handling, from ETMv3.4* on page 7-33.

7.5.2 Instructions that can be paused for continuation

In other ARM processor architectures, there are two ways of responding to an interrupt that occurs during the execution of an LSM instruction:

- The instruction is completed before branching to the interrupt handler.
- The instruction is cancelled before branching to the interrupt handler. In this case, all data transfers are stopped and the results discarded. The instruction must be executed on return from the exception handler.

See *Load/Store Multiple (LSM) instructions* on page 4-21 for details of the instructions to which this applies.

In the ARMv7-M architecture, LSM instructions can be *paused for continuation* during execution. When this happens:

- the position at which the instruction is paused is stored in a processor status register that is pushed onto the stack
- on return from the exception handler, execution of the instruction continues from the point where it was paused.

When an instruction is paused, trace output is generated both when the exception occurs and again when execution of the instruction is resumed. Extended *branch with exception* packets are generated at both points, even though no exception occurs when execution is resumed. These packets are described in *Extended exception handling, from ETMv3.4* on page 7-33. In summary:

- The basic format of the packet generated, both when the exception occurs and when the instruction is resumed, is shown in Figure 7-25 on page 7-34.
- When the exception occurs, the format of the Exception information section of the packet is shown in:
  - Figure 7-28 on page 7-38, if the exception number is 15 or less
  - Figure 7-29 on page 7-38, if the exception number is greater than 15.

  In both cases, the Can bit is 1, Resume[3:0] encodes the last successfully completed transfer, and the Exception[3:0] or Exception[8:0] field holds the exception number.
- When processing of the instruction is resumed, the format of the Exception information section of the packet is shown in Figure 7-28 on page 7-38. The Can bit is 0, Resume[3:0] encodes the last successfully completed transfer index, and the Exception[3:0] field is b0000, indicating that there is no exception.

Tracing continuation of an instruction during instruction-only trace

In instruction-only trace, there is no tracing of the resumed instruction. With instruction-only trace:

- When the exception occurs, the paused instruction is traced as cancelled by the exception branch packet.
- The return from the exception handler is traced as a normal branch, to the instruction that follows the paused instruction in the program execution flow. No Exception information bytes are output.

7.5.3 Automatic stack push on exception entry and pop on exception exit

The ARMv7-M architecture introduces an automatic stack push whenever an exception occurs. When considering data tracing, this is a major difference from all other ARM processor architectures, where data transfers are always initiated by an instruction, and data trace can always be associated with a traced instruction.

In the ARMv7-M architecture, when an exception occurs, eight registers are pushed onto the stack automatically. When execution of the exception handler is complete, the eight registers are popped from the stack and restored. There is a special case extension of this, referred to as *tail-chaining*, when two exceptions are executed back-to-back. In this case, there is no stack pop and subsequent push between the handling of the two events, and the handling of the two tail-chained events is:

1. Before entry to the first event handler the registers are pushed onto the stack.
2. Control passes to the first event handler.
3. When execution of the event handler is complete control passes directly to the second event handler, without any pop, restore and push of the registers.
4. When execution of the second event handler is complete the registers are popped from the stack and restored.

Being able to perform data tracing of the register push and pop operations is important, because this gives access to the contents of the registers. Also, you can use this trace to help to identify the cause of any corruption of the stack by software. Therefore, the ETM must trace the data transfers for the stack push that occurs before the exception handler is entered. These transfers do not have a parent instruction. Instead, an *Exception entry packet* is inserted into the trace stream, to indicate the start of the stack push. The data transfers for the stack push are associated with this packet, not with the previous instruction. These transfers are traced in the same way as a normal STM instruction.

Figure 7-40 shows the format of the Exception entry packet.

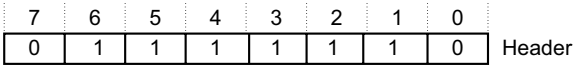


Figure 7-40 Exception entry packet, ETMv3.4 and later

If cycle-accurate tracing is enabled, outputting an Exception entry packet has no effect on the current cycle count, and does not imply that any W, E, or N atoms have occurred.

During instruction-only trace the data transfers for the stack push are not traced, and Exception entry packets are not output.

Tracing might be enabled while a stack push is in progress, either because of exit from an overflow condition or from a normal trace-on occurrence. In this situation, tracing must be enabled immediately:

- the I-Sync packet traces the address of the instruction that caused the exception
- the Exception entry packet is traced immediately after the I-Sync packet
- no P-header is traced for the instruction that caused the exception.

A higher-priority exception might occur during an exception-entry stack push. This results in one of the following two situations, that must be traced as described here:

- The original stack push completes, the first exception is taken, and its vector table entry is loaded. At this point a second stack push is performed, to enter the second exception handler.

  In this case, the trace sequence is:

  — Exception entry packet and data trace for the first stack push

- Exception branch packet for the first exception handler
- P-header for the first instruction of the first exception handler
- Exception entry packet and data trace for the second stack push
- Exception branch packet for the second exception handler
- P-header for the first instruction of the second exception handler.

- The original stack push completes, but the first exception is not taken. The vector table load is performed for the second exception. The first exception is taken only when the higher-priority exception handler has completed. At that point the original exception is traced as if it has just been noticed.

  In this case the trace sequence is:

  - Exception entry packet and data trace for the stack push.
  - Exception branch packet for the second exception handler.
  - P-header for the first instruction of the second exception handler, followed immediately by the trace for all of the code of the second exception handler.

    When the second exception handler has completed there is no stack pop and restore, because this is a tail-chained case.
  - Exception branch packet for the first exception handler
  - P-header for the first instruction of the first exception handler.

  If another higher-priority exception occurs during the execution of the second exception handler then the branch to the first exception handler is not taken until execution of the handler for the new exception has completed.

7.5.4 Tracing return from an exception

In ARM architectures other than ARMv7-M, returning from an exception is performed by executing the special `RFE` instruction, or by moving the value required for the return address into the PC. The ETM traces these returns as simple indirect branches.

The ARMv7-M architecture extends the possible methods of returning from an exception, by extending the meaning of some existing instructions. In simple terms, if one of these instructions results in a transfer of a particular predefined value into the PC then this is treated as a *return from exception* event. This event causes eight registers to be popped from the stack, and has other minor effects. The instructions that can be used in this way are:

- a `POP` or `LDM` that loads into the PC
- an `LDR` with the PC as its destination
- a `BX` with any register.

In ARMv7-M processors, this method is always used to return from an exception. For more information see the *ARMv7-M Architecture Reference Manual*.

These command extensions mean that the effect of a particular command can vary enormously, depending on whether it is used normally or to cause a return from exception. Therefore, ETMv3.4 introduces a new packet that is used to identify the return from exception use of these commands. This is the *Return from exception* packet.

When one of the extended instructions causes a return from exception event, a Return from exception packet is inserted in the trace, between the P-header for the instruction and the branch packet for the branch to the return address. This means that the trace for the command, when causing a return from exception, is:

- P-header for the command
- Return from exception packet
- Branch packet, for branch to the return address.

Figure 7-41 shows the format of the Return from exception packet.

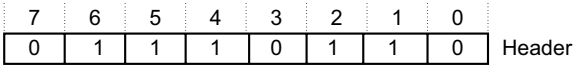


Figure 7-41 Return from exception packet, ETMv3.4 and later

When tracing the use of one of these commands for a Return from exception, the branch packet output depends on whether or not the return is from a tail-chained exception handler, see *Automatic stack push on exception entry and pop on exception exit* on page 7-57 for an explanation of tail-chaining.

- If the exception handler from which control is returning is not tail-chained then the branch packet is a normal indirect branch.
- If the exception handler is tail-chained then an exception branch packet is output. This packet describes the second exception.

Note

A Return from exception packet is always output when one of the extended commands is used to cause a return from exception, regardless of any tail-chaining effects. This means that a Return from exception packet is not always associated with a pop and restore of the registers, because there is no pop and restore on a return from a tail-chained exception handler.

If cycle-accurate tracing is enabled, outputting the Return from exception packet has no effect on the current cycle count, and does not imply the occurrence of any W, E or N atoms.

Tracing might be enabled while a stack pop is in progress, either because of exit from an overflow condition or from a normal trace-on occurrence. In this situation, tracing must be enabled immediately:

- the I-Sync packet traces the address of the exception return instruction
- the Return from exception packet is traced immediately after the I-Sync packet
- no P-header is traced for the exception return instruction.

If a new higher priority exception stops the stack pop by preemption, the branch to the new exception handler must indicate that the last instruction was cancelled, with no resumption information. This means that the Can bit is set to 1, and Resume[3:0] = b0000. This indicates that the Return from exception *packet*

was cancelled, but the return from exception *instruction* was not cancelled. The presence of a Return from exception packet in the trace output stream indicates that the instruction causing the return from exception completed.

Data tracing of return from exception

If data tracing is enabled, the data transfers of the stack pop must be traced. However, these transfers do not have a parent instruction. When a stack pop is performed, the data transfers are associated with the appropriate Return from exception packet. These transfers are traced in the same way as a normal `LDM` instruction.

Note

As stated earlier in this section, a Return from exception is always output when one of the extended commands is used to cause a return from exception, regardless of any tail-chaining effects. However, no stack pop is performed on a return from a tail-chained exception. This means that, when data tracing is enabled, there can be Return from exception packets with no associated data transfers.

7.6 Behavior of EmbeddedICE inputs, from ETMv3.4

In ETMv3.3 and earlier, if an ETM implementation supported EmbeddedICE watchpoint comparator inputs then it provided two EmbeddedICE inputs. From ETMv3.4, the number of EmbeddedICE watchpoint comparator inputs is IMPLEMENTATION DEFINED, between 0 and 8, and is indicated by bits [19:16] of the Configuration Code Extension Register, see *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

In addition, ETMv3.4 defines an optional read/write register that permits dynamic control of the behavior of the EmbeddedICE watchpoint comparator inputs, see *EmbeddedICE Behavior Control Register, ETMv3.4 and later* on page 3-79. Bit [21] of the Configuration Code Extension Register is set to 1 when the EmbeddedICE Behavior Control Register is implemented. ETMv3.4 also specifies default behavior of the EmbeddedICE watchpoint inputs, in different contexts, that must be implemented when the EmbeddedICE Behavior Control Register is not implemented, see *Default behavior of EmbeddedICE watchpoint inputs* on page 7-63.

Additional information about the behavior of the EmbeddedICE inputs is given in the following sections:

- *EmbeddedICE watchpoint comparator input behavior*
- *Implementation of pulse and latch behavior of EmbeddedICE inputs* on page 7-63
- *EmbeddedICE input usage examples* on page 7-64.

7.6.1 EmbeddedICE watchpoint comparator input behavior

Providing control of the behavior of the EmbeddedICE inputs, and specifying different default behavior of these inputs in different contexts, makes these signals more useful for controlling tracing.

For example, when used for the **TraceEnable** event, the normal requirement is that the controlling input is held between comparisons, to ensure that the **TraceEnable** state is held through a range of addresses:

- when an instruction address is used for comparison in the EmbeddedICE logic, it is preferable to maintain the **TraceEnable** event until the next instruction is traced
- when a data address is used for comparison in the EmbeddedICE logic, it is preferable to maintain the **TraceEnable** event until the next data transfer.

In contrast, when an EmbeddedICE input is used as an input to the trace start/stop block, it is preferable for the input to be pulsed for a single cycle. This avoids the possibility, for example, that a stop signal might be missed because a start signal from an EmbeddedICE input is being maintained.

To take account of these different requirements, the EmbeddedICE Behavior Control Register enables a debugger to program the behavior of each EmbeddedICE watchpoint input, as pulsed or latched, depending on the current use of each input. For more information see *EmbeddedICE Behavior Control Register, ETMv3.4 and later* on page 3-79.

If the EmbeddedICE Behavior Control Register is not implemented then the behavior of the EmbeddedICE watchpoint inputs must differ for different resources, as defined in *Default behavior of EmbeddedICE watchpoint inputs* on page 7-63.

7.6.2 Default behavior of EmbeddedICE watchpoint inputs

When the EmbeddedICE Behavior Control Register is not implemented, Table 7-21 defines the required behavior of the EmbeddedICE watchpoint input connection to the different ETM resources.

Table 7-21 Default behavior of EmbeddedICE watchpoint comparator inputs

Resource driven by EmbeddedICE input	Behavior of input
Trigger event	Pulse
TraceEnable event	Latch
Trace start/stop block input	Pulse
ViewData event	Latch
Counter enable or reload	Pulse
Sequencer state change	Pulse
External output	Pulse

Note

Debuggers can read bit [21] of the Configuration Code Extension Register to discover whether the EmbeddedICE Behavior Control Register is implemented:

- if the register is not implemented the debugger can assume the behavior of the EmbeddedICE watchpoint comparator inputs matches Table 7-21
- if the register is implemented the debugger must configure the behavior of each EmbeddedICE input, as appropriate for the use it is making of the input.

7.6.3 Implementation of pulse and latch behavior of EmbeddedICE inputs

Correct implementation of configurable EmbeddedICE watchpoint comparator inputs requires control signals that indicate the sampling point for each input. For each input, the input is sampled at the appropriate point indicated by the control signals. The sampling depends on the value of the corresponding bit in the Behavior Control Register. If this bit is:

0 If the signal is sampled HIGH, the EmbeddedICE input is asserted for a single cycle from the point where it is sampled.

1 The EmbeddedICE signal is latched to the sampled value, and held until the cycle before the next sample point.

7.6.4 EmbeddedICE input usage examples

These are examples of how EmbeddedICE watchpoint comparator inputs might be use, and how the appropriate input must be configured for each use:

- If an EmbeddedICE watchpoint comparator input is used to count the number of instructions executed at a particular address, the EmbeddedICE input must *pulse* for one cycle each time the EmbeddedICE logic matches the required address. Therefore, the appropriate bit of the EmbeddedICE Behavior Control Register must be set to 0 to indicate that the EmbeddedICE input must be pulsed.
- If an EmbeddedICE watchpoint comparator input is used to count the number of cycles spent in a particular range of instruction addresses, the EmbeddedICE input must latch between each cycle where the EmbeddedICE logic compares an instruction address with the required range. Therefore, the appropriate bit of the EmbeddedICE Behavior Control Register must be set to 1 to indicate that the EmbeddedICE input must be latched.
- If an EmbeddedICE watchpoint comparator input is used to include a particular range of trace addresses using **TraceEnable**, the EmbeddedICE input must latch between each comparison. Therefore, the appropriate bit of the EmbeddedICE Behavior Control Register must be set to 1 to indicate that the EmbeddedICE input must be latched.

For details of configuring the EmbeddedICE Behavior Control Register, see *EmbeddedICE Behavior Control Register, ETMv3.4 and later* on page 3-79.

7.7 Synchronization

There are three forms of synchronization, that occur periodically to enable correct synchronization. Different synchronizations might not occur together. The three forms are described in the following sections:

- *A-sync, alignment synchronization*
- *I-sync instruction synchronization* on page 7-66
- *D-sync, data address synchronization* on page 7-75.

7.7.1 Frequency of synchronization

Each form of synchronization must occur in a specified frequency, that depends on the ETM architecture version:

ETMv3.0 Synchronization must occur every *n* cycles.

ETMv3.1 and later

Synchronization must occur every *n* bytes of trace.

Where *n* is the value of the Synchronization Frequency Register, `0x78`. The implementation might delay synchronization in special cases by up to another *n* cycles, usually to prevent overflow.

An overflow occurs if periodic synchronization does not occur in a period of twice the synchronization frequency.

7.7.2 A-sync, alignment synchronization

Periodically a sequence of five or more A-sync P-headers, b0000 0000, are output, followed by the binary value b1000 0000. This is equivalent to a string of 47 or more 0 bits followed by a 1.

To synchronize, the decompressor must search for this sequence, that cannot occur in any other way. While trace capture devices are usually byte-aligned, this might not be the case for sub-byte ports. Therefore the decompressor must realign all data following the A-sync sequence if required.

The next byte is a header, that can be of any type.

For example, in a byte-aligned system, an A-sync sequence followed by a single **E** P-header might be represented, in hexadecimal bytes, as `00 00 00 00 00 80 84`. However, the same sequence offset by 1 bit, from capture from a 1-bit port, might be represented, in hexadecimal bytes, as `01 00 00 00 00 40 42`. This does not occur if the trace capture device is completely accurate over the course of the trace run. However, if it captures one cycle incorrectly, either capturing an extra cycle or missing one out because of instability on the **TRACECTL** signal then all subsequent captures are offset. By accommodating for the new alignment in each byte, the error can be localized.

Note

The trace is normally byte-aligned if any of the following are true:

- The width of **TRACEDATA** is a multiple of 8 bits, that is, not a 4-bit port.
- The trace is embedded in the CoreSight formatting protocol.

- The trace port was inactive when the Trace Capture Device was connected, and there have been no errors in the capture.

Loss of byte alignment is rare, and most decompressors are slower at decompressing misaligned trace. However, all decompressors must be able to cope with misaligned trace.

The A-sync packet is output in the trace stream when required and the generation of other packets does not affect it. This means that it is output even when **TraceEnable** is LOW.

7.7.3 I-sync instruction synchronization

When the decompressor finds an A-sync sequence, it must search for an I-sync packet. This provides synchronization of the following parts of the trace:

- instruction address
- instruction set state
- address of previous data instruction, if it is still executing
- Context ID.

The I-Sync packet includes a code that gives the reason for the output of the I-Sync, see *Reason codes* on page 7-74. The possible reasons are:

- periodic synchronization
- tracing enabled
- tracing restarted after an overflow
- the processor has exited debug state.

If the code indicates periodic synchronization, the I-sync packet is called a Periodic I-sync packet. Otherwise, it is called a Non-periodic I-sync packet.

There are five forms of I-sync packet:

- Normal I-sync packet, see *Normal I-sync packet* on page 7-67
- Normal with cycle count I-sync packet, see *Normal I-sync with cycle count packet* on page 7-68
- *Load/Store in Progress* (LSiP) I-sync packet, see *Load/Store in Progress (LSiP) I-sync packet* on page 7-70
- Load/Store in Progress (LSiP) with cycle count I-sync packet, see *Load/Store in Progress (LSiP) I-sync with cycle count packet* on page 7-72
- Data-only I-sync packet, see *Data-only I-sync packet* on page 7-74.

Use of I-sync packets in cycle-accurate mode

When tracing in cycle-accurate mode, a cycle count is required for every Non-periodic I-sync packet to indicate the number of cycles (W atoms) since the last P-header packet prior to the I-sync packet. This is output in one of the following ways:

- The I-sync packet, followed by a Cycle count packet before the next Non-periodic I-sync packet. The Cycle count packet might not be present if there is a subsequent ETM FIFO overflow, and in this case the cycle count is unknown.
- The I-sync packet is a Normal I-sync with cycle count packet.
- The I-sync packet is a *Load/Store in Progress* (LSiP) I-sync with cycle count packet.

For details of the possible use of I-sync packets for tracing long gaps in trace during cycle-accurate tracing, see *Tracing long gaps in cycle-accurate trace* on page 7-79.

Normal I-sync packet

Figure 7-42 shows the format of a Normal I-sync packet.

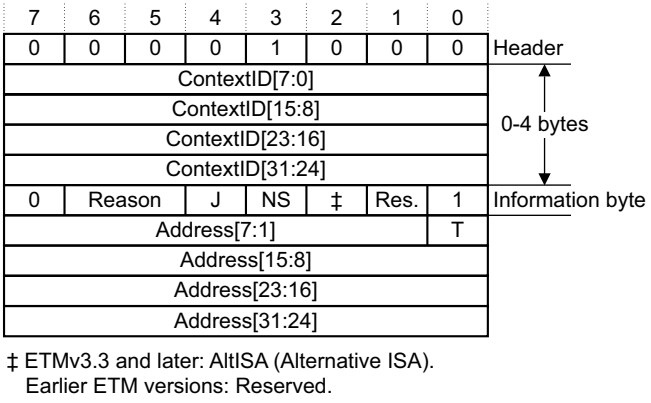


Figure 7-42 Normal I-sync packet

A normal I-sync packet comprises the following contiguous components:

I-sync header Indicates that this is an I-sync packet.

Context ID The number of Context ID bytes traced (0-4) is statically determined by ETM Control Register bits [15:14]. For more information on Context ID, see *Context ID packets* on page 7-43.

I-sync information byte

In this byte:

- Bit [7] distinguishes between Normal and LSiP I-sync packets.

- Bits [6:5] are a 2-bit reason code, see *Reason codes* on page 7-74 for more information.
- Bit [4] is set to 1 if the processor is in Jazelle state.
- Bit [3] is set to 1 if the processor is in a Non-secure state.
- From ETMv3.3, bit [2] is the Alternative instruction set (AltISA) bit. See *The Alternative instruction set bit, ETMv3.3 and later* for more information.

Instruction address

The instruction address is always four bytes and is not compressed.

Bit [0] is the Thumb bit and is set to 1 if the processor is in Thumb state. However, see *The Alternative instruction set bit, ETMv3.3 and later* for the interpretation of this bit in ETMv3.3 and later.

The Alternative instruction set bit, ETMv3.3 and later

From ETMv3.3, bit [2] of the Information byte is the Alternative instruction set (AltISA) bit. From ETMv3.3, to find state of the processor you must consider:

- the J bit, bit [4] of the Information byte
- the T bit, bit [0] of the Address field
- the AltISA bit, bit [2] of the Information byte.

Table 7-22 shows how the values of these bits correspond to the different processor states.

Table 7-22 Processor state information in I-sync packets, ETMv3.3 and later

J bit	T bit	AltISA bit	Alignment	Processor State
0	0	0	Word	ARM.
0	0	1	Word	Not used. Reserved combination of J, T and AltISA.
0	1	0	Halfword	Thumb.
0	1	1	Halfword	ThumbEE.
1	X	0	Byte	Jazelle.
1	X	1	Byte	Not used. Reserved combination of J, T and AltISA.

Normal I-sync with cycle count packet

A Normal I-sync with cycle count packet is equivalent to a Normal I-sync packet followed by a Cycle count packet. The cycle count indicates the number of cycles (W atoms) that have occurred since the last P-header packet.

A Normal I-sync with cycle count packet is never output with a reason code of periodic synchronization, because the cycle count is used to indicate the gap between trace regions, and when periodic synchronization is performed there is no gap in the trace.

Figure 7-43 shows the format of a Normal I-sync with cycle count packet.

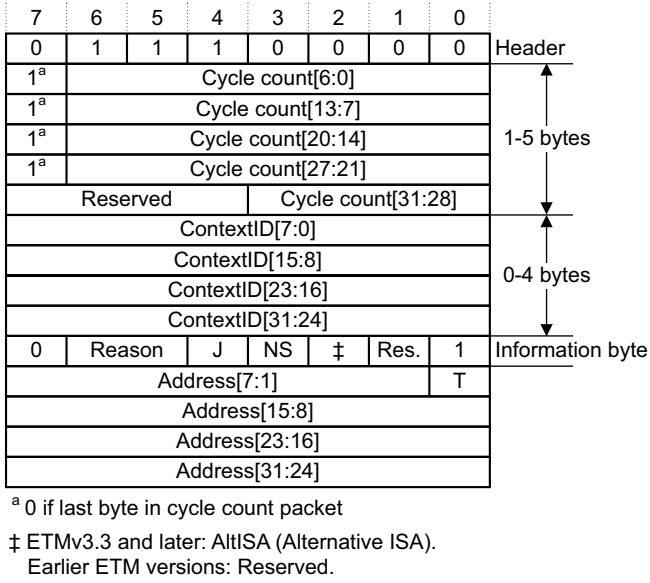


Figure 7-43 Normal I-sync with cycle count packet

A normal I-sync packet with cycle count comprises the following contiguous components:

I-sync header Indicates that this is an I-sync packet.

Cycle count The number of cycles since the last P-header. For more information on P-headers, see *Cycle information, for cycle-accurate tracing* on page 7-9.

Context ID The number of Context ID bytes traced (0-4) is statically determined by ETM Control Register bits [15:14]. For more information on Context ID, see *Context ID packets* on page 7-43.

I-sync information byte

In this byte:

- Bit [7] distinguishes between Normal and LSiP I-sync packets.
- Bits [6:5] are a 2-bit reason code, see *Reason codes* on page 7-74 for more information.
- Bit [4] is set to 1 if the processor is in Jazelle state.
- Bit [3] is set to 1 if the processor is in a Non-secure state.

- From ETMv3.3, bit [2] is the Alternative instruction set (AltISA) bit. See *The Alternative instruction set bit, ETMv3.3 and later* on page 7-68 for more information.

Instruction address

The instruction address is always four bytes and is not compressed.

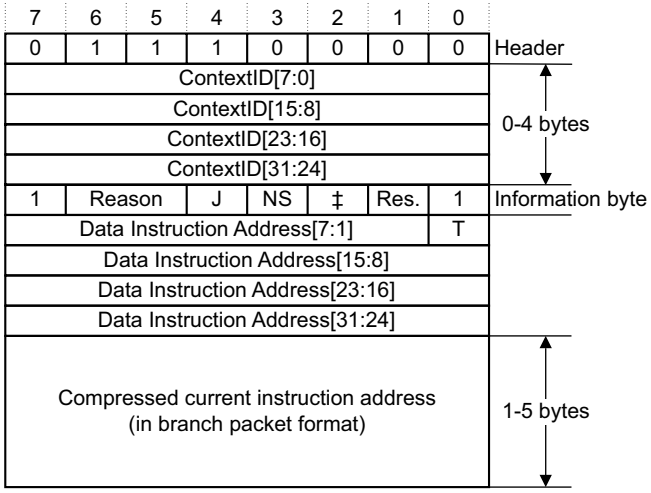
Bit [0] is the Thumb bit and is set to 1 if the processor is in Thumb state. However, see *The Alternative instruction set bit, ETMv3.3 and later* on page 7-68 for the interpretation of this bit in ETMv3.3 and later.

Load/Store in Progress (LSiP) I-sync packet

LSiP I-sync packets occur only when the following conditions occur simultaneously:

- Trace is enabled in the middle of a data instruction. *Definitions* on page 4-21 lists the data instructions.
- Another instruction is currently executing.

An LSiP I-sync packet is never output with a reason code of periodic synchronization because this I-sync packet is only used when tracing is started. Figure 7-44 shows the format of an LSiP I-sync packet.



‡ ETMv3.3 and later: AltISA (Alternative ISA). Reserved in earlier ETM versions.

Figure 7-44 LSiP I-sync packet

An LSiP I-sync packet comprises the following contiguous components:

I-sync header

Indicates that this is an I-sync packet.

Context ID The number of Context ID bytes traced (0-4) is statically determined by ETM Control Register bits [15:14]. For more information on Context ID, see *Context ID packets* on page 7-43.

I-sync information byte

In this byte:

- Bit [7] distinguishes between Normal and LSiP I-sync packets.
- Bits [6:5] are a 2-bit reason code, see *Reason codes* on page 7-74 for more information.
- Bit [4] is set to 1 if the processor is in Jazelle state.
- Bit [3] is set to 1 if the processor is in a Non-secure state.
- From ETMv3.3, bit [2] is the Alternative instruction set (AltISA) bit. See *The Alternative instruction set bit, ETMv3.3 and later* on page 7-68 for more information.

Data instruction address

This is the address of the data instruction executing in parallel with the current instruction.

Bit [0] is the Thumb bit and is set to 1 if the processor is in Thumb state. However, see *The Alternative instruction set bit, ETMv3.3 and later* on page 7-68 for the interpretation of this bit in ETMv3.3 and later.

Execution of the instruction at this address is implied, as if an **E** atom was traced after this address and before the current instruction addresses.

Compressed current instruction address

The address for the instruction currently executing (1-5 bytes) is compressed using the technique that is used for Branch addresses, see *Branch packets* on page 7-11. The exception vector format is not used in this case. A 6-byte address might be used here to indicate a change in security level.

This instruction address is compressed relative to the full address from the data instruction address. The next instruction atom is for the instruction pointed to by the compressed current instruction address and tracing begins in the normal way from this point forwards.

The LSiP I-sync packet type enables correct tracing of all instructions that touch a particular data address or data value. Without it, the data instruction cannot be properly traced based on the data address.

If trace is enabled in the middle of a data instruction, another instruction has since executed and left the pipeline but no instruction is currently executing, a Normal I-sync packet is output, giving the address of the data instruction. A Branch address packet is output giving the address of the next instruction to execute before it is traced.

Note

Instructions occurring underneath the data instruction are traced even if tracing is programmed to turn on only during the data instruction itself. Similarly, if tracing starts because of the instruction address of an instruction that executes underneath a data instruction, an LSiP I-sync packet is still output.

Load/Store in Progress (LSiP) I-sync with cycle count packet

An LSiP I-sync with cycle count packet is equivalent to an LSiP I-sync packet followed by a Cycle count packet. The cycle count packet indicates the number of cycles (W atoms) that have occurred since the last P-header packet.

An LSiP I-sync with cycle count packet is never output with a reason code of periodic synchronization because this I-sync packet is only used when tracing is started.

Figure 7-45 shows the format of an LSiP I-sync with cycle count packet.

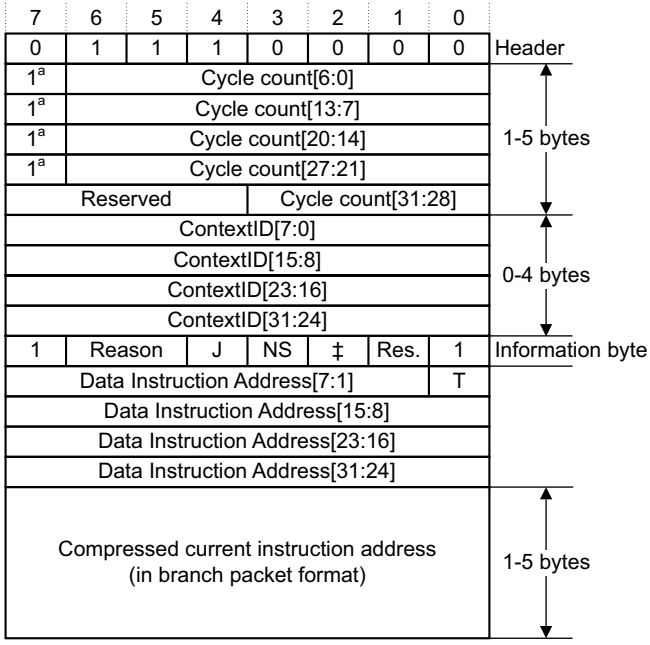


Figure 7-45 LSiP I-sync with cycle count packet

An LSiP I-sync with cycle count packet comprises the following contiguous components:

I-sync header

Indicates that this is an I-sync packet.

Cycle count The number of cycles since the last P-header. For more information on P-headers, see *Cycle information, for cycle-accurate tracing* on page 7-9.

Context ID The number of Context ID bytes traced (0-4) is statically determined by ETM Control Register bits [15:14]. For more information on Context ID, see *Context ID packets* on page 7-43.

I-sync information byte

In this byte:

- Bit [7] distinguishes between Normal and LSiP I-sync packets.
- Bits [6:5] are a 2-bit reason code, see *Reason codes* on page 7-74 for more information.
- Bit [4] is set to 1 if the processor is in Jazelle state.
- Bit [3] is set to 1 if the processor is in a Non-secure state.
- From ETMv3.3, bit [2] is the Alternative instruction set (AltISA) bit. See *The Alternative instruction set bit, ETMv3.3 and later* on page 7-68 for more information.

Data instruction address

This is a fixed 4-byte address, the address of the data instruction executing in parallel with the current instruction.

Bit [0] is the Thumb bit and is set to 1 if the processor is in Thumb state. However, see *The Alternative instruction set bit, ETMv3.3 and later* on page 7-68 for the interpretation of this bit in ETMv3.3 and later.

Execution of the instruction at this address is implied, as if an **E** atom was traced after this address and before the current instruction addresses.

Compressed current instruction address

The address for the instruction currently executing (1-5 bytes) is compressed using the technique that is used for Branch addresses, see *Branch packets* on page 7-11. The exception vector format is not used in this case.

This instruction address is compressed relative to the full address from the data instruction address. The next instruction atom is for the instruction pointed to by the compressed current instruction address, and tracing begins in the normal way from this point. A 6-byte address might be used here to indicate a change in security level.

The LSiP I-sync packet type enables correct tracing of all instructions that touch a particular data address or data value. Without it, the data instruction cannot be properly traced based on the data address.

If trace is enabled in the middle of a data instruction, and another instruction has since executed and left the pipeline, but no instruction is currently executing, a Normal I-sync packet is output, giving the address of the data instruction. A Branch address packet is output giving the address of the next instruction to execute before it is traced.

Note

Instructions occurring underneath the data instruction are traced even if tracing is programmed to turn on only during the data instruction itself. Similarly, if tracing starts because of the instruction address of an instruction that executes underneath a data instruction, an LSiP I-sync packet is still output.

Data-only I-sync packet

In data-only mode, data-only I-sync packets are output instead of the other forms of I-sync packets. Figure 7-46 shows the format of a data-only I-sync packet.

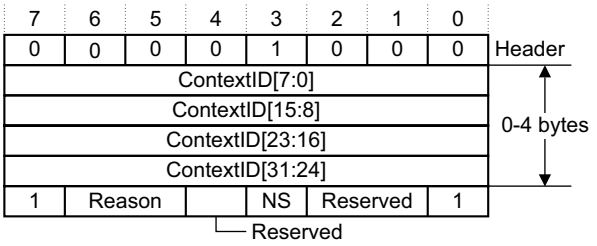


Figure 7-46 Data-only I-sync packet

A data-only I-sync packet comprises the following contiguous components:

I-sync header Indicates that this is an I-sync packet.

Context ID The number of Context ID bytes traced (0-4) is statically determined by ETM Control Register bits [15:14]. For more information on Context ID, see *Context ID packets* on page 7-43.

I-sync information byte

This includes a 2-bit reason code. There is no bit to indicate whether the processor is in Jazelle state. Bit [3] is set to 1 if the processor is in Non-secure state.

Reason codes

Reason codes are encoded as the information byte in the I-sync packet header. The reason codes are listed in Table 7-23.

Table 7-23 ETMv3 reason codes

Value	Description
b00	Periodic I-sync.
b01	Tracing enabled.
b10	Tracing restarted after overflow. This takes precedence over b01. This is not output following exit from debug state. See *ETM Status Register, ETMv1.1 and later* on page 3-33.
b11	ARM processor has exited from debug state. This code is only used if the first non-prohibited instruction following exit from debug state is traced.

7.7.4 D-sync, data address synchronization

D-sync provides synchronization of data addresses to enable compressed data addresses to be reliably decompressed. It is achieved by periodically outputting a full 5-byte address. This is output as part of the Normal data and Out-of-order placeholder packets as described in *Data tracing* on page 7-44.

D-sync occurs for the following reasons:

- the first data address is output following a trace gap
- periodically, as specified in *Frequency of synchronization* on page 7-65.

D-sync is not required on the first data transfer after a periodic I-sync packet.

It is usual that a single counter is used for both D-sync and I-sync, and that the counter values are staggered to reduce the likelihood of overflow.

7.8 Trace port interface

The behavior of the trace port interface is described in:

- *Trigger*
- *Ignore* on page 7-77
- *FIFO draining* on page 7-77.

An ETM might not output trace directly onto a trace port, but instead onto an on-chip bus for routing to an on-chip trace buffer or more complex trace port. One example is the CoreSight AMBA *Advanced Trace Bus* (ATB), described in the *CoreSight Architecture Specification*. The ATB enables trace from multiple trace sources, one of which might be an ETM, to be combined and output over one trace port or captured in an on-chip trace buffer. In these systems, triggers must be indicated on the trace port using the mechanism supported by the trace port, and the ETM must be capable of indicating the trigger condition to the trace port interface unit.

7.8.1 Trigger

The trigger is indicated by outputting the trigger packet header. Figure 7-47 shows the trigger packet header.

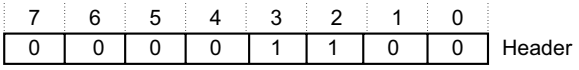


Figure 7-47 Trigger packet

In multi-byte ports this must be output on **TRACEDATA[7:0]**, and Ignore packets can be inserted into the trace to ensure that this is the case.

The trigger packet is output even if the ETM is recovering from an overflow.

TRACECTL is asserted on this cycle. This enables the TPA to detect that the trigger is zero.

Note

Bit [0] of the trigger header is always output on **TRACEDATA[0]**.

In sub-byte ports, **TRACECTL** is asserted on the first cycle of the trigger packet output, and deasserted on the remaining cycles, to make sure that all cycles are captured.

In ETMv3.0, the trigger is output within a few cycles of the cycle it occurred on. In ETMv3.1 and later, the trigger is output in a cycle-accurate manner.

The trigger packet is output in the trace stream when the trigger event occurs and is unaffected by the generation of other trace packets. This means that it is output even when **TraceEnable** is LOW.

7.8.2 Ignore

The Ignore packet header has no effect. Figure 7-48 shows the Ignore packet header.

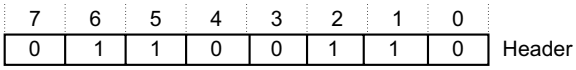


Figure 7-48 Ignore packet

It can be used in unused bytes of the trace port if trace must be output when there is insufficient trace to fill the entire port.

7.8.3 FIFO draining

When there is no trace to output then **TRACECTL** and **TRACEDATA[0]** are both asserted. During any cycle in which trace is output, **TRACECTL** is deasserted.

Usually no trace is output unless there is sufficient trace to fill the entire width of **TRACEDATA**. This makes the most efficient use of the space in the TCD where **TRACEDATA** is wider than 8 bits.

If trace is output when there is not sufficient trace to fill **TRACEDATA** then Ignore headers are output on the unused upper bytes. The circumstances where this is permitted are:

- When the next packet to be output is a trigger, so that the trigger header can appear on **TRACEDATA[7:0]**.
- When A-sync is output. The implementation might choose not to take advantage of this permitted case.
- When the Programming bit is set to 1. This is to make sure that, when trace is disabled at the end of a trace run, all remaining trace is drained.

During Trace Disabled, **TRACEDATA[1]** must also be asserted. This is ignored by the TPA.

7.9 Tracing through regions with no code image

Decompressing the trace requires the code image to be available. However, often the code image is not available for some areas of memory, for example system libraries, and it is not practical to filter all these regions out. These are referred to as *unknown regions*. The decompressor can resume tracing when an indirect branch occurs to a known region, without having to wait for the next synchronization point. The protocol is designed to enable the length of each packet to be determined without reference to the code image, so that alignment synchronization is not lost. The following information must continue to be monitored:

Branch addresses These must be monitored to keep track of the last output address, used to compress branch addresses.

Data addresses These must be monitored so that the first data address can be decompressed.

Context IDs These can still be traced.

Cycle-accurate information is unaffected.

When tracing from a known region to an unknown region, data corresponding to the last data instruction in the known region must be discarded if the last data instruction did not have all of its data traced.

This is because the first data traced in the unknown region might correspond to the last data instruction in the known region, or to an instruction at the beginning of the unknown region. Alternatively, the decompressor can discard all data corresponding to the last data instruction in the known region whenever an unknown region is encountered.

7.10 Cycle-accurate tracing

When profiling the execution of critical code sequences, it is often useful if you can observe the exact number of cycles that a particular code sequence takes to execute. To perform this *cycle-accurate tracing*, you must set bit [12] of the ETM Control Register to 1, see *ETM Control Register* on page 3-20.

For more information about cycle-accurate tracing in ETMv3 see:

- *P-header encodings in cycle-accurate mode* on page 7-6
- *Cycle information, for cycle-accurate tracing* on page 7-9
- *Cycle count packet* on page 7-10.

7.10.1 Tracing long gaps in cycle-accurate trace

There can be long gaps in the execution of instructions by the processor, for example if the processor is in a Wait For Interrupt or Wait For Event condition. In cycle-accurate mode, these gaps can be traced in three ways:

1. No trace is output during the gap. When execution resumes, a non-periodic I-sync packet with cycle count is generated, see *Normal I-sync with cycle count packet* on page 7-68.

2. No trace is output during the gap. When execution resumes, a non-periodic I-sync packet is generated, see *Normal I-sync packet* on page 7-67. At some short time after this, a cycle count packet is generated, see *Cycle count packet* on page 7-10.

3. A W atom is output for each cycle of the gap, see *P-header encodings in cycle-accurate mode* on page 7-6.

   This method has the disadvantage of increasing the amount of trace generated.

Which of these methods is used is IMPLEMENTATION SPECIFIC. A particular implementation might use more than one of these methods.

7.10.2 Support for cycle-accurate tracing, ETMv3.3 and later

From ETMv3.3, whether an ETM macrocell supports cycle-accurate tracing is IMPLEMENTATION DEFINED. Debug tools can write and then read the ETM Control Register to find whether cycle-accurate tracing is supported. For details, see *Checking support for cycle-accurate tracing, ETMv3.3 and later* on page 3-28.

7.11 ETMv2 and ETMv3 compared

This section describes how some of the concepts in the ETMv2 protocol are represented in the ETMv3 protocol.

7.11.1 ETMv2 PIPESTAT encodings and ETMv3 P-headers compared

ETMv3.0 provides alternative mechanisms for indicating the trigger and trace disabled conditions. These mechanisms replace the TR and WT pipeline status codes. When data appears in the trace stream, it always corresponds to the most recent cycle or instruction. This means it is not necessary to indicate whether data follows, and therefore the remaining 14 pipeline status conditions reduce to seven possibilities. These can be represented as a combination of the following three P-header atoms:

- **W** is a cycle boundary, all pipeline status conditions indicate this
- **E** is an instruction that passed its condition codes test
- **N** is an instruction that failed its condition codes test.

Table 7-24 shows the mappings.

Table 7-24 Mappings from pipeline status to P-header atoms

Pipeline status	Atoms
Instruction executed (IE), instruction executed with data (DE)	**W**,**E**
Instruction not executed (IN), instruction not executed with data (DN)	**W**,**N**
Wait (WT), wait with data (DW)	**W**
Trigger (TR), trace disabled (TD)	Not applicable
Branch phantom taken plus instruction executed (PTIE), branch phantom taken plus instruction executed with data (PTDE)	**W**,**E**,**E**
Branch phantom taken plus instruction not executed (PTIN), branch phantom taken plus instruction not executed with data (PTDN)	**W**,**E**,**N**
Branch phantom not taken plus instruction executed (PNIE), branch phantom not taken plus instruction executed with data (PNDE)	**W**,**N**,**E**
Branch phantom not taken plus instruction not executed (PNIN), branch phantom not taken plus instruction not executed with data (PNDN)	**W**,**N**,**N**

7.11.2 ETMv2 TFO packets and ETMv3 I-sync packets compared

I-sync packets in ETMv3 are equivalent to TFO packets in ETMv2 with the following differences:

- They are preceded by an I-sync P-header.

- The Context ID comes before the information byte. This is to prevent the A-sync value occurring five times in succession.
- Bit [0] of the information byte is always set to 1, to prevent an information byte of 0 conflicting with A-sync.

The presence of a periodic I-sync packet does not imply an instruction executed as in ETMv2.x. Instead, a periodic I-sync can occur at any time. The only time that an I-sync implies the execution of an instruction is when an LSiP I-sync packet is output, where the execution of the LSiP instruction is implied as in ETMv2.

The instruction address always gives the address of the next instruction to be executed, even for periodic synchronization. Previously, periodic synchronization gave the address of the instruction just executed.

Chapter 8
Trace Port Physical Interface

This chapter describes the external pin interface, timing, and connector type required for the trace port on a target system. It contains the following sections:

8.1 Target system connector

The specified target system connector is the AMP Mictor connector. For high-speed tracing through a demultiplexed, half-speed port using 8 or 16 **TRACEPKT** bits, two Mictor connectors are required.

The AMP Mictor connector is a high-density matched-impedance connector. This connector has several important attributes:

- direct connection to a logic analyzer probe using a high-density adapter cable with termination (for example, HPE5346A from Agilent)
- matching impedance characteristics, enabling the same connector to be used up to 200MHz, and possibly higher
- a large number of ground fingers to ensure good signal integrity
- inclusion of the run-time control (JTAG) signals on the connector, enabling a single debug connection to the target.

Table 8-1 shows the AMP part numbers for the four possible connectors.

Table 8-1 Connector part numbers

AMP part number	Description
2-767004-2	Vertical, surface mount, board to board/cable connector
767054-1	Vertical, surface mount, board to board/cable connector
767061-1	Vertical, surface mount, board to board/cable connector
767044-1	Right angle, straddle mount, board to board/cable connector

The choice of connector used depends on factors such as board thickness and cost. Contact AMP connector distributors for details of connectors to meet application-specific requirements.

8.2 Target connector pinouts

This section contains details of the target system connector pinouts as follows:

- *Single target connector pinout* on page 8-4
- *Dual target connector pinout* on page 8-7
- *Multiplexed trace port, single target connector pinout (ETMv1.x and ETMv2.x)* on page 8-8
- *Demultiplexed trace port target connector pinout* on page 8-10
- *Signal descriptions* on page 8-11
- *Connector placement* on page 8-14
- *Dual connector placement* on page 8-15
- *Half-rate clocking mode* on page 8-17.

Note

The target connector pinout described in release C or later of the ETM specification is changed from that given in releases A and B.

All 38 pins are specified in the tables in this section.

The connector supports:

- up to 20 trace information pins, depending on the ETM architecture version
- one trace clock pin
- one external trigger pin
- one voltage reference pin
- one voltage supply pin
- nine JTAG interface pins.

In multiplexed mode, each pin carries two ETM signals. In demultiplexed mode, each signal is split across two pins.

Port modes are described in *Trace port clocking modes* on page 2-67.

8.2.1 Assignment of trace information pins between ETM architecture versions

The names used for the trace signals depend on the ETM architecture version. In the rest of this chapter trace signals are assigned as shown in Table 8-2.

Table 8-2 Trace signal names

Trace signal	ETMv1	ETMv2	ETMv3
Trace signal 1	**PIPESTAT[0]**	**PIPESTAT[0]**	**TRACEDATA[0]**
Trace signal 2	**PIPESTAT[1]**	**PIPESTAT[1]**	**TRACECTL**
Trace signal 3	**PIPESTAT[2]**	**PIPESTAT[2]**	**Logic 1**

Table 8-2 Trace signal names (continued)

Trace signal	ETMv1	ETMv2	ETMv3
Trace signal 4	**TRACESYNC**	**PIPESTAT[3]**	**Logic 0**
Trace signal 5	**TRACEPKT[0]**	**TRACEPKT[0]**	**Logic 0**
Trace signal 6	**TRACEPKT[1]**	**TRACEPKT[1]**	**TRACEDATA[1]**
Trace signal 7	**TRACEPKT[2]**	**TRACEPKT[2]**	**TRACEDATA[2]**
Trace signal 8	**TRACEPKT[3]**	**TRACEPKT[3]**	**TRACEDATA[3]**
Trace signal 9	**TRACEPKT[4]**	**TRACEPKT[4]**	**TRACEDATA[4]**
Trace signal 10	**TRACEPKT[5]**	**TRACEPKT[5]**	**TRACEDATA[5]**
Trace signal 11	**TRACEPKT[6]**	**TRACEPKT[6]**	**TRACEDATA[6]**
Trace signal 12	**TRACEPKT[7]**	**TRACEPKT[7]**	**TRACEDATA[7]**
Trace signal 13	**TRACEPKT[8]**	**TRACEPKT[8]**	**TRACEDATA[8]**
Trace signal 14	**TRACEPKT[9]**	**TRACEPKT[9]**	**TRACEDATA[9]**
Trace signal 15	**TRACEPKT[10]**	**TRACEPKT[10]**	**TRACEDATA[10]**
Trace signal 16	**TRACEPKT[11]**	**TRACEPKT[11]**	**TRACEDATA[11]**
Trace signal 17	**TRACEPKT[12]**	**TRACEPKT[12]**	**TRACEDATA[12]**
Trace signal 18	**TRACEPKT[13]**	**TRACEPKT[13]**	**TRACEDATA[13]**
Trace signal 19	**TRACEPKT[14]**	**TRACEPKT[14]**	**TRACEDATA[14]**
Trace signal 20	**TRACEPKT[15]**	**TRACEPKT[15]**	**TRACEDATA[15]**

8.2.2 Single target connector pinout

The pinout for a single-processor ETM target connector is shown in Table 8-3.

Table 8-3 Single target connector pinout

Pin	Signal name	Pin	Signal name	Pin	Signal name	Pin	Signal name
38	Trace signal 1	37	Trace signal 13	36	Trace signal 2	35	Trace signal 14
34	Trace signal 3	33	Trace signal 15	32	Trace signal 4	31	Trace signal 16
30	Trace signal 5	29	Trace signal 17	28	Trace signal 6	27	Trace signal 18

Table 8-3 Single target connector pinout (continued)

Pin	Signal name	Pin	Signal name	Pin	Signal name	Pin	Signal name
26	Trace signal 7	25	Trace signal 19	24	Trace signal 8	23	Trace signal 20
22	Trace signal 9	21	**nTRST**[a]	20	Trace signal 10	19	**TDI**[a]
18	Trace signal 11	17	**TMS**[a]	16	Trace signal 12	15	**TCK**[a]
14	**VSupply**[a]	13	**RTCK**[a]	12	**VTRef**	11	**TDO**[a]
10	**EXTTRIG**[a]	9	**nSRST**[a]	8	**DBGACK**[a]	7	**DBGRQ**[a]
6	**TRACECLK**	5	**GND**	4	No-connect	3	No-connect
2	No-connect	1	No-connect	-	-	-	-

a. Run control signal.

Note

Pins 1, 2, 3, and 4 *must* be true no-connects. For designs with fewer than 16 trace data pins, pin 5 and any unused trace signal pins *must* be connected to ground on the target board.

Pipeline status seen by old TPAs, ETMv3.0 upwards

The pipeline status seen by old TPAs is listed in Table 8-4.

Table 8-4 Pipeline status seen by old TPAs

ETMv3.x			ETMv2.x	ETMv1.x
TRACECTL	TRACEDATA[0]	Simulated PIPESTAT[3:0]	mnemonic	mnemonic
0	X[a]	b010x	WT/DW	BE/BD
1	0	b0110	TR	TR
1	1	b0111	TD	TD

a. Can be 0 or 1.

For more information on trigger and trace disabled conditions see *Decoding required by trace capture devices* on page 4-7

Wider trace ports, ETMv3.0 upwards

If you require a wider trace port than that shown in *Single target connector pinout* on page 8-4 then a second connector must be used.

Up to 16 extra bits of **TRACEDATA** are supported on the second connector. If the second connector is used:

- **TRACEPKT[15:0]** is equivalent to **TRACEDATA[31:16]**
- **PIPESTAT[3:0]** is wired to b0100.

Table 8-5 shows the single target connector pinout for ETMv3.x.

Table 8-5 Second target connector pinout ETMv3.x

Pin	Signal name	Pin	Signal name	Pin	Signal name
38	**TRACEDATA[16]**	37	**TRACEDATA[24]**	36	**Logic 0**
35	**TRACEDATA[25]**	34	**Logic 1**	33	**TRACEDATA[26]**
32	**Logic 0**	31	**TRACEDATA[27]**	30	**Logic 0**
29	**TRACEDATA[28]**	28	**TRACEDATA[17]**	27	**TRACEDATA[29]**
26	**TRACEDATA[18]**	25	**TRACEDATA[30]**	24	**TRACEDATA[19]**
23	**TRACEDATA[31]**	22	**TRACEDATA[20]**	21	No-connect
20	**TRACEDATA[21]**	19	No-connect	18	**TRACEDATA[22]**
17	No-connect	16	**TRACEDATA[23]**	15	No-connect
14	No-connect	13	No-connect	12	**VTRef**
11	No-connect	10	No-connect	9	No-connect
8	No-connect	7	No-connect	6	**TRACECLK**
5	**GND**	4	No-connect	3	No-connect
2	No-connect	1	No-connect	-	-

8.2.3 Dual target connector pinout

The pinout for a dual-processor ETM target connector is shown in Table 8-6.

Table 8-6 Dual target connector pinout

Pin	Signal name	Pin	Signal name	Pin	Signal name	Pin	Signal name
38	Trace signal 1_A	37	Trace signal 1_B	36	Trace signal 2_A	35	Trace signal 2_B
34	Trace signal 3_A	33	Trace signal 3_B	32	Trace signal 4_A	31	Trace signal 4_B
30	Trace signal 5_A	29	Trace signal 5_B	28	Trace signal 6_A	27	Trace signal 6_B
26	Trace signal 7_A	25	Trace signal 7_B	24	Trace signal 8_A	23	Trace signal 8_B
22	Trace signal 9_A	21	**nTRST**	20	Trace signal 10_A	19	**TDI**
18	Trace signal 11_A	17	**TMS**	16	Trace signal 12_A	15	**TCK**
14	**VSupply**	13	**RTCK**	12	**VTRef**	11	**TDO**
10	**EXTTRIG**	9	**nSRST**	8	**DBGACK**	7	**DBGRQ**
6	**TRACECLK_A**	5	**TRACECLK_B**	4	No connect	3	No connect
2	No connect	1	No connect	-	-	-	-

Note

Pins 1, 2, 3, and 4 *must* be true no-connects. For designs with less than 16 trace data pins, unused **TRACEPKT** pins *must* be connected to ground.

The **TRACECLK** connections differ based on:

- whether the two trace ports are synchronous or asynchronous
- whether you want to use multiple TPAs or a dual module logic analyzer for trace collection.

Asynchronous trace ports

If the two trace ports are asynchronous, **TRACECLK_A** and **TRACECLK_B** are driven separately by the ASIC.

Synchronous trace ports

If you want to use a logic analyzer for collecting the trace, it is normally possible to configure both modules to use the same clock. In this situation it is recommended that you use **TRACECLK_A** and connect pin 5 to GND.

If you intend to use two TPAs to collect the trace from the two trace ports, **TRACECLK_A** and **TRACECLK_B** are both required. It is recommended that you drive these from two separate pins on your ASIC. This has the advantage that you can run the trace ports asynchronously, without having to make changes to the PCB or the ASIC pinout.

If it is not possible to use separate pins, you can drive both from the same pin provided that:

- the ASIC pad drivers are capable of driving the increased load
- both PCB tracks are series-terminated as close as possible to the pin of the ASIC.

8.2.4 Multiplexed trace port, single target connector pinout (ETMv1.x and ETMv2.x)

Note

This option is supported only in ETMv1.x and ETMv2.x.

The pinout for a multiplexed trace port, single-processor target connector is shown in Table 8-7.

Table 8-7 Multiplexed trace port, single target connector pinout

Pin	Signal name	Pin	Signal name
38	**PIPESTAT[0]** + **TRACESYNC** in ETMv1.x **PIPESTAT[0]** + **PIPESTAT[3]** in ETMv2.x	37	No connect
36	**PIPESTAT[1] + TRACEPKT[1]**	35	No connect
34	**PIPESTAT[2] + TRACEPKT[2]**	33	No connect
32	**TRACEPKT[0,3]**	31	No connect
30	**TRACEPKT[4,5]**	29	No connect
28	**TRACEPKT[6,7]**	27	No connect
26	**TRACEPKT[8,9]**	25	No connect
24	**TRACEPKT[10,11]**	23	No connect
22	**TRACEPKT[12,13]**	21	**nTRST**
20	**TRACEPKT[14,15]**	19	**TDI**
18	No connect	17	**TMS**
16	No connect	15	**TCK**
14	**VSupply**	13	**RTCK**
12	**VTRef**	11	**TDO**

Table 8-7 Multiplexed trace port, single target connector pinout (continued)

Pin	Signal name	Pin	Signal name
10	**EXTTRIG**	9	**nSRST**
8	**DBGACK**	7	**DBGRQ**
6	**TRACECLK**	5	**GND**
4	No connect	3	No connect
2	No connect	1	No connect

Table 8-8 shows the edges that you must use to sample the pairs of signals in a multiplexed trace port connector.

Table 8-8 Paired signals in a multiplexed trace port connector

Connector groups		Signals sampled on the rising edge of TRACECLK	Signals sampled on the falling edge of TRACECLK
These signals are paired for a 4-pin trace port connector	These signals are paired for a 6-pin trace port connector	**PIPESTAT[0]**	**TRACESYNC** in ETMv1 **PIPESTAT[3]** in ETMv2
		PIPESTAT[1]	**TRACEPKT[1]**
		PIPESTAT[2]	**TRACEPKT[2]**
		TRACEPKT[0]	**TRACEPKT[3]**
-		**TRACEPKT[4]**	**TRACEPKT[5]**
		TRACEPKT[6]	**TRACEPKT[7]**
	-	**TRACEPKT[8]**	**TRACEPKT[9]**
		TRACEPKT[10]	**TRACEPKT[11]**
		TRACEPKT[12]	**TRACEPKT[13]**
		TRACEPKT[14]	**TRACEPKT[15]**

Note

Pins 1, 2, 3, and 4 *must* be true no-connects. Pin 5 and all unused **TRACEPKT** pins *must* be connected to ground on the target board.

8.2.5 Demultiplexed trace port target connector pinout

A demultiplexed trace port requires twice the number of output pins as a standard trace port, because it runs at half the clock rate. For a 4-bit demultiplexed trace port these can be accommodated using a single connector.

The pinout for this is shown in Table 8-9. When decompressing the trace, the data from **PIPESTAT_B**, **TRACESYNC_B**, and **TRACEPKT_B** must be read before the data from **PIPESTAT_A**, **TRACESYNC_A**, and **TRACEPKT_A**.

Table 8-9 Demultiplexed 4-bit connector pinout

Pin	Signal name	Pin	Signal name
38	**PIPESTAT_A[0]**	37	**PIPESTAT_B[0]**
36	**PIPESTAT_A[1]**	35	**PIPESTAT_B[1]**
34	**PIPESTAT_A[2]**	33	**PIPESTAT_B[2]**
32	**TRACESYNC_A** in ETMv1 **PIPESTAT_A[3]** in ETMv2	31	**TRACESYNC_B** in ETMv1 **PIPESTAT_B[3]** in ETMv2
30	**TRACEPKT_A[0]**	29	**TRACEPKT_B[0]**
28	**TRACEPKT_A[1]**	27	**TRACEPKT_B[1]**
26	**TRACEPKT_A[2]**	25	**TRACEPKT_B[2]**
24	**TRACEPKT_A[3]**	23	**TRACEPKT_B[3]**
22	No connect	21	**nTRST**
20	No connect	19	**TDI**
18	No connect	17	**TMS**
16	No connect	15	**TCK**
14	**VSupply**	13	**RTCK**
12	**VTRef**	11	**TDO**
10	**EXTTRIG**	9	**nSRST**
8	**DBGACK**	7	**DBGRQ**
6	**TRACECLK**	5	**GND**
4	No connect	3	No connect
2	No connect	1	No connect

The pinouts for 8-bit or 16-bit demultiplexed trace ports look identical to two single processor connectors. You must not connect the run control signal pins on the second connector. These signals are:

- **VSupply**
- **EXTTRIG**
- **DBGACK**
- **nTRST**
- **TDI**
- **TMS**
- **TCK**
- **RTCK**
- **TDO**
- **nSRST**
- **DBGRQ**.

8.2.6 Signal descriptions

For details of the **TRACECLK**, **TRACESYNC**, **PIPESTAT**, and **TRACEPKT** output signals, see Chapter 4 *Signal Protocol Overview*.

The following sections describe the signals on the target connector pins:

- *EXTTRIG input*
- *VTRef output* on page 8-12
- *VSupply output* on page 8-12
- *nTRST input* on page 8-12
- *TDI input* on page 8-12
- *TMS input* on page 8-12
- *TCK input* on page 8-13
- *RTCK output* on page 8-13
- *TDO output* on page 8-13
- *nSRST input* on page 8-13
- *DBGRQ input* on page 8-13
- *DBGACK output* on page 8-13
- *VDD input* on page 8-13.

EXTTRIG input

EXTTRIG is an optional signal. It is intended to be an input to one of the external inputs on the ETM. Depending on the design, ETM external triggers might not be available on the ASIC external pins. In this case the **EXTTRIG** has no function. It is recommended that this pin is pulled to a defined state.

VTRef output

The **VTRef** signal is intended to supply a logic-level reference voltage to enable debug equipment to adapt to the signalling levels of the target board. It does *not* supply operating current to the debug equipment. Target boards must supply a voltage that is nominally between 1V and 5V. With ±10% tolerance, this is minimum 0.9V, maximum 5.5V. The target board must provide a sufficiently low DC output impedance so that the output voltage does not change by more than 1% when supplying a nominal signal current (±0.4mA). Debug equipment that connects to this signal must interpret it as a signal rather than a power supply pin and not load it more heavily than a signal pin. The recommended maximum source or sink current is ±0.4mA.

VSupply output

The **VSupply** signal enables the target board to supply operating current to debug equipment so that an additional power supply is not required. This might not be used by all debug equipment. The VDD power rail typically drives the pin on the target board. Target board documentation must indicate the **VSupply** pin voltage and the current available. Target boards must supply a voltage that is nominally between 2V and 5V. With ±10% tolerance, this is minimum 1.8V, and maximum 5.5V. A target board that drives this pin must provide a minimum of 250mA, and 400mA is recommended. Debug equipment must indicate the required supply voltage range and the current consumption over that range. This enables you to determine whether an external power supply is required to power the debug equipment. Target boards might have a limited amount of current available for external debug equipment, so a backup mechanism to power the debug equipment must be provided where **VSupply** is not connected, or is insufficient. For some hardware, this signal is unused.

nTRST input

The **nTRST** signal is an open collector input from the run control unit to the **Reset** signal on the target JTAG port. This pin must be pulled HIGH on the target to avoid unintentional resets when there is no connection.

Note

Board logic must ensure that there is a LOW pulse on the **nTRST** pin of the target ASIC at power up.

TDI input

TDI is the Test Data In signal from the run control unit to the target JTAG port. It is recommended that you pull this pin to a defined state.

TMS input

TMS is the Test Mode Select signal from the run control unit to the target JTAG port. This pin must be pulled up on the target so that the effect of any spurious **TCK**s when there is no connection is benign.

TCK input

TCK is the Test Clock signal from the run control unit to the target JTAG port. It is recommended that this pin is pulled to a defined state.

RTCK output

RTCK is the Return Test Clock signal from the target JTAG port to the run control unit. Some targets, such as ARM7TDMI-S™ processor, must synchronize the JTAG port to internal clocks. To assist in meeting this requirement, you can use a returned (and re-timed) **TCK** to dynamically control the **TCK** rate.

TDO output

This signal is the Test Data Out from the target JTAG port to the run control unit.

nSRST input

This is an open collector output from the run control unit to the target system reset. This might also be an input to the run control unit so that a reset initiated on the target can be reported to the debugger.

You must pull this pin HIGH on the target to avoid unintentional resets when there is no connection.

DBGRQ input

The **DBGRQ** signal is used by the run control unit as a debug request signal to the target processor. It is recommended that this pin is pulled to a defined state. This signal is rarely implemented as a pin on the target ASIC. Use of this pin is not recommended for the dual-target connector.

You must pull this pin LOW on the target to avoid unintentional debug requests when there is no run control unit connected.

DBGACK output

The **DBGACK** signal is used by some run control units to detect entry or exit from debug state. This signal is rarely implemented as a pin on the target ASIC. Use of this pin is not recommended for the dual-target connector.

VDD input

V_{DD} is a logic level 1 signal for compatibility with TPAs designed for ETMv1 or ETMv2 only. It is normally equivalent to **VTRef**.

8.3 Connector placement

This section describes:

- *Connector orientation*
- *Dual connector placement* on page 8-15.

8.3.1 Connector orientation

The connector can be oriented on the target system as shown in Figure 8-1. This shows the view from above the PCB with the trace connector mounted near to the edge of the board. This enables the TPA to minimize the physical intrusiveness of the target interconnect, that can be PCB-to-PCB, to ensure signal integrity.

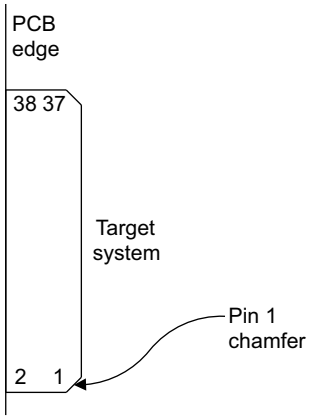


Figure 8-1 Recommended connector orientation

8.3.2 Dual connector placement

Where two connectors are used it is recommended that they are placed in line, separated by 1.35 inches, as shown in Figure 8-2.

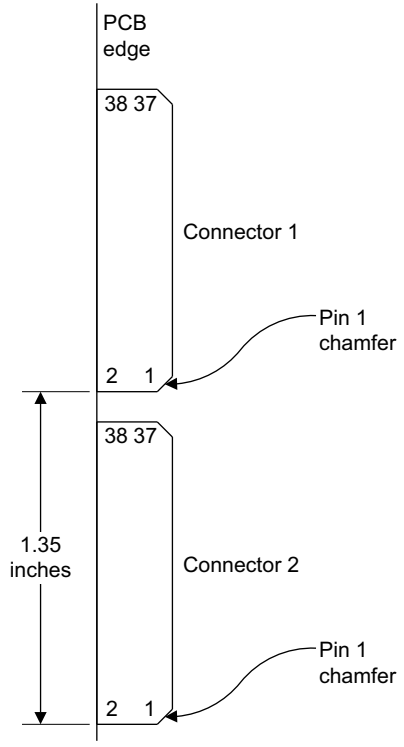


Figure 8-2 Recommended dual connector orientation

8.4 Timing specifications

There are no inherent restrictions on operating frequency, other than ASIC pad technology and TPA limitations. ASIC designers must provide a **TRACECLK** as symmetrical as possible, and with set up and hold times as large as possible. TPA designers must conversely be able to support a **TRACECLK** as asymmetrical as possible, and require set up and hold times as short as possible. The following timing specifications are given as a guide for a TPA that supports **TRACECLK** frequencies up to around 100MHz.

Note

Actual processor clock frequencies vary according to application requirements and the silicon process technologies used. The maximum operating clock frequencies attained by ARM devices increases over time as a result.

If you adhere to the timing described here, you can use any ARM-approved TPA. Figure 8-3 depicts the timing for **TRACECLK**.

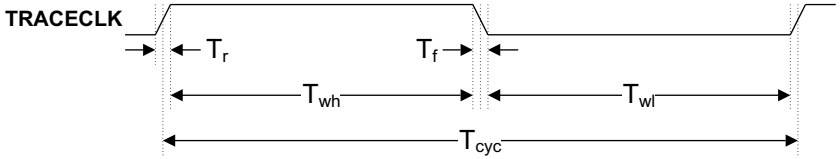


Figure 8-3 TRACECLK specification

Table 8-10 shows details of the timing requirements for **TRACECLK** parameters.

Table 8-10 TRACECLK timing requirements

Parameter	Minimum	Description
T_{cyc}	Frequency dependent	Clock period
T_{wl}	2ns	LOW pulse width
T_{wh}	2ns	HIGH pulse width

Table 8-11 shows rise and fall time requirements for all ETM clock and data signals.

Table 8-11 Rise and fall time requirements

Parameter	Maximum	Description
T_r	3ns	Clock and data rise time
T_f	3ns	Clock and data fall time

Figure 8-4 on page 8-17 shows the setup and hold requirements of the trace data pins with respect to **TRACECLK**.

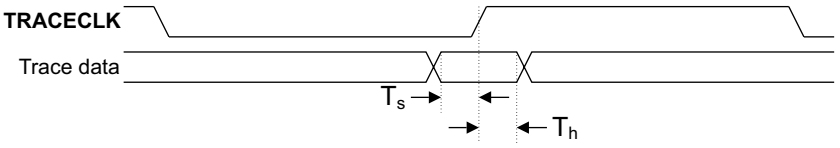


Figure 8-4 Trace data specification

Table 8-12 shows the timing requirements for Figure 8-4.

Table 8-12 Trace port setup and hold requirements

Parameter	Minimum	Description
T_s	3ns	Data setup
T_h	2ns	Data hold

8.4.1 Half-rate clocking mode

When half-rate clocking is used, the trace data signals are sampled by the TPA on both the rising and falling edges of **TRACECLK**, where **TRACECLK** is half the frequency of the clock shown in Figure 8-4.

8.5 Signal level specifications

Debug equipment must be able to deal with a wide range of signal voltage levels. Typical ASIC operating voltages can range from 1V to 5V, although 1.8V to 3.3V is common.

8.6 Other target requirements

It is important that you keep the trace length differences as small as possible to minimize skew between signals. Crosstalk on the trace port must be kept to a minimum as it can cause erroneous trace results. Stubs on these traces can cause UNPREDICTABLE responses, especially at high frequencies, so it is recommended that no stubs exist on the trace lines. If stubs are necessary, you must make them as small as possible.

The trace port clock line (**TRACECLK**) must be series terminated as close as possible to the pins of the driving ASIC.

The maximum capacitance that is presented by the trace connector, cabling, and interfacing logic must be less than 15pF.

For processor frequencies greater than 100MHz you must take great care in the design of the input/output pads, chip package, PCB layout, and connections to the chosen TPA. You are recommended to use SPICE modeling.

8.7 JTAG control connector

Some JTAG controller products use a different connector to that specified in *Target system connector* on page 8-2. Therefore you must use either a second connector for the chosen JTAG controller, or an adapter board connected to the specified connector.

Chapter 9
Tracing Dynamically Loaded Images

This chapter describes software issues relating to the ETMs. It contains the following sections:

9.1 About tracing dynamically-loaded code

When a debugger is debugging a system, it communicates mainly in terms of accesses to addresses in memory or virtual memory. It translates between these addresses and the locations in the code images loaded on the system. This means that the debugger can present a symbolic or source-level view of the code running on the system.

In a simple statically-linked and loaded system, a single image is run to describe the mapping of target addresses as image locations. To perform debugging, the debugger requires only the name of the code image. However, many systems, including operating systems such as Windows CE, Linux, or Symbian OS, load part or all of their software dynamically. This can have several effects:

- the address at which an image is loaded might not be known until it is loaded
- at different times, different images might be loaded at the same address
- in a complex system, the debugger might not know what images are candidates to be loaded until they are loaded.

To debug systems like these, the debugger must be able to examine the target, to determine what images are loaded and from where they are loaded.

The problem is more complex when using trace, because trace data is historical information. Any embedded trace solution requires an image of the code that was executed to be available to the trace decompression software of the debugger, otherwise the debugger cannot decode the trace.

The compression algorithm used for trace conserves data bandwidth by broadcasting only the minimum of address information. This means that, given a (compressed) address issued by the trace port, the tools must be able to know what instructions are at and around that point. This enables the target address of direct branches (`B` and `BL` instructions in the case of code in ARM state) to be inferred. This is difficult with, for example, virtual memory and software paging, because the debugger is unlikely to know where the code is executed from.

To resolve this problem, ETM uses Context IDs. These require both software and hardware support, as described in:

- *Software support for Context ID* on page 9-5
- *Hardware support for Context ID* on page 9-6.

Note

In addition to the support for Context ID described in this chapter, from ETMv3.3 there is combined hardware and software support for saving the complete debug configuration. Although this is intended to enable the configuration to be saved and restored when an ETM macrocell is power-cycled, it might be used for other purposes. See *Power-down support, ETMv3.3 and later* on page 3-119 for details of this support.

9.1.1 Simple overlay support

A system for supporting simple overlays is possible that does not require specific support in the debugger. This solution is based on the requirement that the memory space into which the overlays are loaded exists in multiple places in the memory map as shown in Figure 9-1. That is, some of the unused address bits are *don't care* when determining the memory to be accessed.

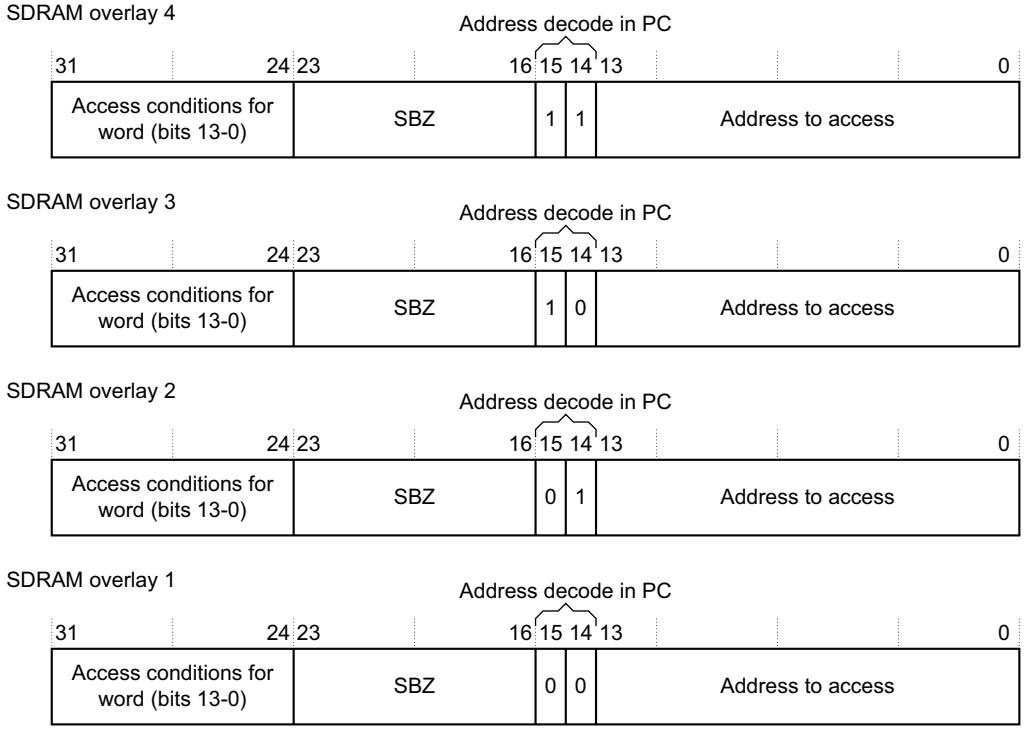


Figure 9-1 SDRAM overlay examples

For example, if you have 16KB of SRAM, bits [13:0] of the address determine the 32-bit word to access and bits [31:24] determine when to access that particular block. However, if bits [15:14] are in the address decoder, four copies of the memory block exist in the memory map. In other words the same word can be accessed using four different addresses, that is, when bits [15:14] of the address are b00, b01, b10, or b11 as shown in Figure 9-2 on page 9-4.

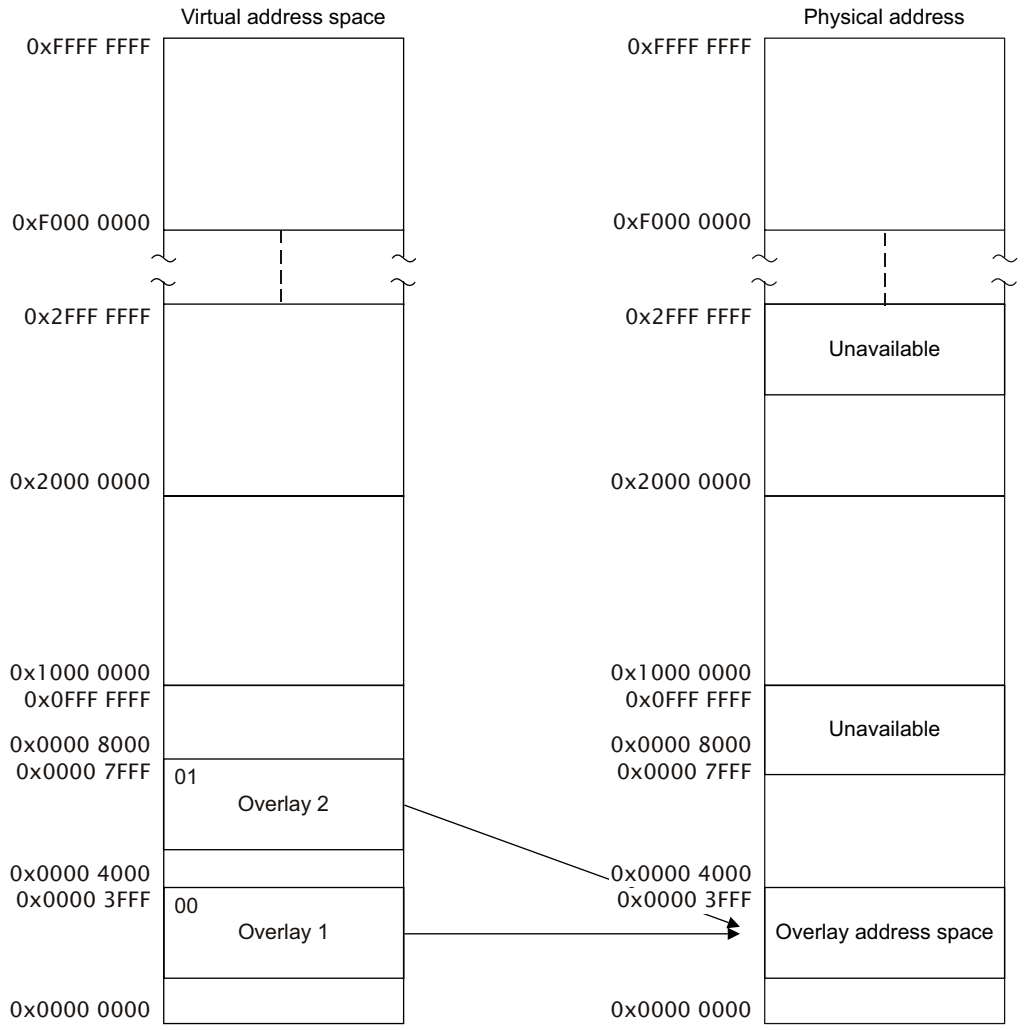


Figure 9-2 Memory map and overlay physical address space

9.2 Software support for Context ID

When the operating system switches between binary images, or virtual memory spaces, it must update the value in the Context ID register that is part of coprocessor 15. The debugger must have access to a mapping file specifying the Context ID that correlates to each binary image. With this information, the debugger can then associate each binary image with the correct part of the trace.

9.3 Hardware support for Context ID

A variable-length Context ID value is output whenever trace is enabled, and as part of the periodic synchronization packet. This enables the current Context ID value to be passed to the debugger. You can also filter out unwanted trace based on the current Context ID using programmable trigger resources.

To support tracing when only a partial binary image is available, the compression protocol maintains synchronization even as the ETM branches into unknown code regions. When the code jumps back into a region for which the code image is available trace is decompressable immediately.

Appendix A
ETM Quick Reference information

This appendix contains quick-reference information for some key aspects of the ETM. It contains the following sections:

A.1 ETM Event Resources

This section contains quick-reference information about configuring the ETM event resources. It contains the following sections:

- *Resource identification and event encoding*
- *Resource control registers* on page A-5.

A.1.1 Resource identification and event encoding

An ETM event is a Boolean combination of ETM resources. An event is encoded in a 17-bit Event Register as shown in Figure A-1.

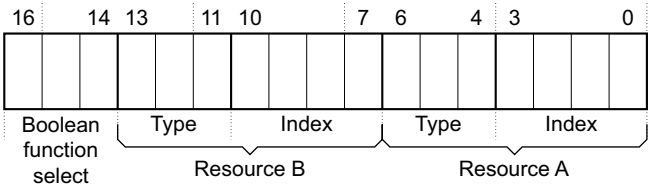


Figure A-1 Writing to an Event Register

Table A-1 shows the encodings used for resources in Event Registers.

Table A-1 Resource identification encoding

Resource type[a]	Index values[a]	Description of resource type
b000	0-15	Single address comparator. (Produces = and >= outputs. The >= output is used only as part of the address range comparison.)
b001	0-7	Address range comparison. Uses pairs of address comparators.
	8-11	Instrumentation resource 1-4. Software-controlled resources, see *Instrumentation resources, from ETMv3.3* on page 2-63. Only available in ETMv3.3 and later.
b010	0-7[b]	EmbeddedICE module watchpoint comparators, if implemented[c].
b011	0-15	Memory map decoder, if implemented[d].
b100	0-3	Counter at zero.

Table A-1 Resource identification encoding (continued)

Resource type[a]	Index values[a]	Description of resource type
b101	0-2	Sequencer in states 1-3.
	3-7	Reserved.
	8-10	Context ID comparator 1-3, ETMv2.0 and later.
	11-14	Reserved.
	15	Trace start/stop resource, ETMv2.0 and later[e].
b110	0-3	External inputs 1-4.
	4-7	Reserved.
	8-11	Extended external input selectors 1-4, ETMv3.1 and later.
	12	Reserved.
	13	Core is in Non-secure state.
	14	Trace prohibited by core.
	15	Hard-wired input (always true).
b111	-	Reserved.

a. The Resource type is bits [6:4] of the 7-bit resource identifier, and the Index value is bits [3:0] of the identifier. Sometimes, the combined 7-bit resource identifier is called the Resource number.
b. 0-7 in ETMv3.4 and later, 0 and 1 only in ETMv3.3 and earlier. See Footnote [c].
c. EmbeddedICE module watchpoint comparators are not implemented in all ETMs. For more information see *EmbeddedICE watchpoint comparators* on page 2-8. In ETMv3.4 and later there can be up to eight EmbeddedICE watchpoint comparators, with index values 0 to 7. In earlier ETMs, if the Embedded ICE watchpoint comparators are implemented there are always two comparators, with index values 0 and 1.
d. Memory map decoders are not implemented in all ETMs. For more information see *Memory map decoder (MMD)* on page 2-8.
e. The trace start/stop resource is driven by the trace start/stop block, that is not implemented on all ETMs. For more information see *The trace start/stop block* on page 2-23.

Table A-2 shows the encodings for the Boolean operations to be applied to the event resources.

Table A-2 Boolean function encoding for events

Encoding	Function
b000	A
b001	NOT(A)
b010	A AND B
b011	NOT(A) AND B
b100	NOT(A) AND NOT(B)

Table A-2 Boolean function encoding for events (continued)

Encoding	Function
b101	A OR B
b110	NOT(A) OR B
b111	NOT(A) OR NOT(B)

Note

To permanently enable or disable an event, you must specify external input 16, using either function A or NOT (A).

Table A-3 shows the locations of the 17-bit Event Registers.

Table A-3 Locations of ETM event registers

Location	Description
0x02	Trigger Event
0x08	**TraceEnable** Event
0x0C	**ViewData** Event
0x54	Enabling event for counter 1 (Counter 1 Enable)
0x55	Enabling event for counter 2 (Counter 2 Enable)
0x56	Enabling event for counter 3 (Counter 3 Enable)
0x57	Enabling event for counter 4 (Counter 4 Enable)
0x58	Counter Reload Event for counter 1
0x59	Counter Reload Event for counter 2
0x5A	Counter Reload Event for counter 3
0x4B	Counter Reload Event for counter 4
0x60	Sequencer State Transition event for State 1 to State 2
0x61	Sequencer State Transition event for State 2 to State 1
0x62	Sequencer State Transition event for State 2 to State 3
0x63	Sequencer State Transition event for State 3 to State 1

Table A-3 Locations of ETM event registers (continued)

Location	Description
0x64	Sequencer State Transition event for State 3 to State 2
0x65	Sequencer State Transition event for State 1 to State 3
0x68	External Output Event for external output 1
0x69	External Output Event for external output 2
0x6A	External Output Event for external output 3
0x6B	External Output Event for external output 4

A.1.2 Resource control registers

This section contains register tables for ETM resource control. These are Table A-4 to Table A-28 on page A-12.

Table A-4 ASIC Control Register, 0x003

Bit number	Description
[7:0]	ASIC control

Table A-5 Trace Start/Stop Resource Control Register, 0x006

Bit number	Description
[31:16]	When a bit is set to 1, it selects a single address comparator 16 to 1 as stop addresses. For example, bit [16] set to 1 selects single address comparator 1.
[15:0]	When a bit is set to 1, it selects a single address comparator 16 to 1 as start addresses. For example, bit [0] set to 1 selects single address comparator 1.

Table A-6 TraceEnable Control 1 Register, `0x009`

Bit number	Description
[25]	Trace start/stop enable: **0** Tracing is unaffected by the trace start/stop logic (ETMv1.2 and later). **1** Tracing is controlled by trace on and off addresses.
[24]	Include/exclude control: **0** Include. The specified resources indicate the regions in which tracing can occur. When outside this region tracing is prevented. **1** Exclude. The resources specified in bits [23:0] and in the TraceEnable Control 2 Register indicate regions to be excluded from the trace. When outside an exclude region, tracing can occur.
[23:8]	When a bit is set to 1, it selects a memory map decode 16 to 1 for include/exclude control. For example, bit [8] set to 1 selects MMD 1.
[7:0]	When a bit is set to 1, it selects an address range comparator 8 -1 for include/exclude control. For example, bit [0] set to 1 selects address range comparator 1.

Table A-7 TraceEnable Control 2 Register, `0x007`

Bit number	Description
[15:0]	When a bit is set to 1, it selects single address comparator 16 to 1 for include/exclude control. For example, bit [0] set to 1 selects single address comparator 1.

Table A-8 FIFOFULL Region Register, `0x00A`

Bit number	Description
[24]	Include/exclude control: **0** Include. The specified resources indicate the regions in which **FIFOFULL** can be asserted. When outside these regions, **FIFOFULL** cannot be asserted. **1** Exclude. The resources specified in bits [23:0] indicate the regions in which **FIFOFULL** cannot be asserted. When outside these regions **FIFOFULL** can be asserted.
[23:8]	When a bit is set to 1, it selects memory map decode 16 to 1 for include/exclude control. For example, bit [8] set to 1 selects MMD 1.
[7:0]	When a bit is set to 1, it selects address range comparator 8-1 for include/exclude control. For example, bit [0] set to 1 selects address range comparator 1.

Table A-9 FIFOFULL Level Register, `0x00B`

Bit number	Access	Description
[7:0]	Write-only (ETMv1.x) Read-only (ETMv2.x)	The number of bytes left in the FIFO, below which the **FIFOFULL** signal is asserted

Table A-10 ViewData Control 1 Register, `0x00D`

Bit number	Description
[31:16]	When a bit is set to 1, it selects single address comparator 16 to 1 for exclude control. For example, bit [16] set to 1 selects single address comparator 1.
[15:0]	When a bit is set to 1, it selects single address comparator 16 to 1 for include control. For example, bit [0] set to 1 selects single address comparator 1.

Table A-11 ViewData Control 2 Register, `0x00E`

Bit number	Description
[31:16]	When a bit is set to 1, it selects memory map decode 16 to 1 for exclude control. For example, bit [16] set to 1 selects MMD 1.
[15:0]	When a bit is set to 1, it selects memory map decode 16 to 1 for include control. For example, bit [0] set to 1 selects MMD 1.

Table A-12 ViewData Control 3 Register, `0xo0F`

Bit number	Description
[16]	Exclude-only control: **0** Mixed mode. **ViewData** operates in a mixed mode, and both include and exclude resources can be programmed. **1** Exclude-only mode. **ViewData** is programmed only in an excluding mode. If none of the excluding resources match, tracing can occur.
[15:8]	When a bit is set to 1, it selects address range comparator 8 -1 for exclude control. For example, bit [8] set to 1 selects address range comparator 1.
[7:0]	When a bit is set to 1, it selects address range comparator 8-1 for include control. For example, bit [0] set to 1 selects address range comparator 1.

Table A-13 Address Comparator Value Registers, `0x010-0x01F`

Bit number	Description
[31:0]	Address value

Table A-14 Address Access Type Registers, `0x020-0x02F`

Bit number	Description	
[11:10] (ETMv3.2)	Secure mode control:	
	b00	Security level ignored.
	b01	Match only if in Non-secure state.
	b10	Match only if in Secure state.
	b11	Reserved.
[9:8] (ETMv2.0 and later)	Context ID comparator control:	
	b00	Ignore Context ID comparators.
	b01	Address comparator matches only if Context ID comparator value 1 matches.
	b10	Address comparator matches only if Context ID comparator value 2 matches.
	b11	Address comparator matches only if Context ID comparator value 3 matches.
[7] (ETMv2.0 and later)	Exact match bit. Specifies comparator behavior when exceptions occur. See Table A-15 on page A-9 and Table A-16 on page A-10.	

Table A-14 Address Access Type Registers, 0x020-0x02F (continued)

Bit number	Description	
[6:5]	Data value comparison control:	
	b00	No data value comparison.
	b01	Address matches only if data value matches.
	b10	Reserved.
	b11	Address matches only if data value does not match (ETMv1.2 and later).
[4:3]	Size:	
	b00	Jazelle instruction or byte data.
	b01	Thumb instruction or halfword data.
	b10	Reserved.
	b11	ARM instruction or word data.
	See *Comparator access size* on page 2-36.	
[2:0]	Access type:	
	b000	Instruction fetch.
	b001	Instruction execute.
	b010	Instruction executed and passed condition code test (ETMv1.2 and later).
	b011	Instruction executed and failed condition code test (ETMv1.2 and later).
	b100	Data load or store.
	b101	Data load.
	b110	Data store.
	b111	Reserved.

Table A-15 Exact match bit settings for instruction accesses

Exact match bit	Instruction canceled	Instruction not canceled
0	Comparator matches	Comparator matches
1	Comparator does not match	Comparator matches

Table A-16 Exact match bit settings for data accesses

Data comparator present?	Exact match bit	Cache hit	Cache miss	Data abort
Yes	0	Comparator matches if data value matches	Comparator matches	Comparator matches
Yes	1	Comparator matches if data value matches	Comparator waits	Comparator does not match
No	0	Comparator matches	Comparator matches	Comparator matches
No	1	Comparator matches	Comparator matches (ETMv3.0 and earlier) Comparator waits (ETMv3.1 and later)	Comparator does not match

Table A-17 Data Comparator Value Registers, `0x030-0x03F`

Bit number	Description
[31:0]	Data value

Table A-18 Data Comparator Mask Registers, `0x040-0x04F`

Bit number	Description
[31:0]	Data mask

Table A-19 Counter Reload Value Registers, `0x050-0x053`

Bit number	Description
[15:0]	Counter reload value

Table A-20 Counter Enable Registers, `0x054-0x057`

Bit number	Description
[17]	Count enable source in ETMv1.0. When 0, the counter is continuously enabled and decrements every cycle. When 1, the count enable event is used to enable the counter. It is recommended that bit [17] is always set to b1and that the count enable event is used to control counter operation. In ETMv2.0 and later, this bit has no effect and is always one.
[16:0]	Count enable event.

Table A-21 Counter Value Registers, 0x05C-0x05F

Bit number	Description
[15:0]	Current counter value

Table A-22 Current Sequencer State Register, 0x067

Bit number	Description	
[1:0]	Possible values are:	
	b00	State 1.
	b01	State 2.
	b10	State 3.

Table A-23 External Output Event Registers, 0x068-0x06B

Bit number	Description
[16:0]	External output event

Table A-24 Locations of the Context ID Comparator Value Registers

Context ID Comparator value	Location
1	0x6C
2	0x6D
3	0x6E

Table A-25 Context ID Comparator Value Registers, 0x06C-0x06E

Bit number	Description
[31:0]	Context ID value

Table A-26 Context ID Comparator Mask Register, 0x06F

Bit number	Description
[31:0]	Context ID mask value

Table A-27 Synchronization Frequency Register, `0x078`

Bit number	Description
[11:0]	Cycle count value. Default value is 1024.

Table A-28 Extended External Input Selection Register, `0x07B`

Bit number	Description
[31:24]	Fourth extended external input selector
[23:16]	Third extended external input selector
[15:8]	Second extended external input selector
[7:0]	First extended external input selector

A.2 Summary of branch packets, ETMv3.0 and later

Table A-29 shows all possible reasons for branches and summarizes the trace packet content for each case. It only applies to ETMv3.0 and later. For full details of branch packets see:

- *Instruction tracing* on page 7-5, for ETMv3.x
- *Instruction tracing with ETMv2* on page 6-22, for ETMv2.x
- *Instruction tracing in ETMv1* on page 5-9, for ETMv1.x.

Table A-29 Full list of branch packets with content summary, ETMv3.0 and later

Reason for branch	Packet contents
Normal branch: With no exception and no change of instruction set state or security state.	1-5 bytes of address
Instruction set change, without an exception: ARM to Thumb, ARM to Jazelle, Thumb to ARM, Jazelle to ARM.	5 bytes of address
Instruction set change, without an exception[a]: ARM to ThumbEE, Thumb to ThumbEE, ThumbEE to ARM, ThumbEE to Thumb.	5 bytes of address 1 Exception information byte[b]
Exception, without an instruction set change. Any change in the security state has no effect on the packet length[c].	2-5 bytes of address[d] 1-3 Exception information bytes[b]
Security state change, with no exception and no instruction set state change.	2-5 bytes of address[d] 1 Exception information byte
Security state change, with instruction set state change but no exception.	5 bytes of address 1 Exception information byte
Return from exception, with no continuation of instruction[e]	1-5 bytes of address[e]
Return from exception, with continuation of instruction[e]	1-5 bytes of address 2 Exception information bytes[e]

a. Only supported from ETMv3.3.
b. Before ETMv3.4, there is a maximum of one Exception information byte.
c. The security state is always output in Exception information byte 0, and therefore any change in the security state does not affect the packet length.
d. In ETMv3.3 and earlier, all 5 bytes of address information are always output.
e. Only supported from ETMv3.4.

A.3 Summary of IMPLEMENTATION DEFINED ETM features

This section lists the ETM features that are IMPLEMENTATION DEFINED, for ETMv3.4. It also indicates how, for a particular ETM, you can check the actual implementation of each feature. See the descriptions of the different features for information about their support in ETM versions before ETMv3.4.

Table A-30 lists the ETM features where it is IMPLEMENTATION DEFINED either:

- the number of times the feature is implemented
- the size of the feature.

With all of these features except for the Trace port size, the minimum permitted value is 0, indicating that the feature is not supported in the ETM implementation.

Table A-30 ETMv3.4 features with IMPLEMENTATION DEFINED number of instances or size

Feature	Permitted values	Value given by
Address comparators	0-8 pairs	Bits [3:0] of the ETM Configuration Code Register.[a]
Data value comparators	0-8	Bits [12:16] of the ETM Configuration Code Register.[a]
EmbeddedICE watchpoint comparators	0-8	Bits [19:16] of the Configuration Code Extension Register.[b]
Context ID comparators	0-3	Bits [25:24] of the ETM Configuration Code Register.[a]
Counters	0-4	Bits [15:13] of the ETM Configuration Code Register.[a]
Sequencer	0, 1	Bit [16] of the ETM Configuration Code Register.[a]
Memory Map decoder inputs	0-16	Bits [12:8] of the ETM Configuration Code Register.[a]
External inputs	0-4	Bits [19:17] of the ETM Configuration Code Register.[a]
External outputs	0-4	Bits [22:20] of the ETM Configuration Code Register.[a]
Extended external input bus width	0-255	Bits [10:3] of the Configuration Code Extension Register.[b]
Extended external input selectors	0-4	Bits [2:0] of the Configuration Code Extension Register.[b]
Instrumentation resources	0-4	Bits [15:13] of the Configuration Code Extension Register.[b]
Trace port size	See text	ETM Control Register bits [21,6:4]. See *ETM port size encoding* on page 3-26.

a. See *ETM Configuration Code Register* on page 3-29.
b. See *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

Table A-31 lists the features that are optional in an ETMv3.4 implementation. This means that, in an ETMv3.4 implementation, it is IMPLEMENTATION DEFINED whether each of these features is supported.

Table A-31 Optional features in ETMv3.4

Implementation of	Check for support by
FIFOFULL control	Reading bit [23] of the ETM Configuration Code Register.[a] See also *Checking whether data suppression is supported, ETMv3.3 and later* on page 2-34.
Trace Start/Stop block	Reading bit [26] of the ETM Configuration Code Register.[a]
Trace all branches	Testing whether you can set bit [8] of the ETM Control Register to 1.[b]
Cycle-accurate trace	Writing 1 to bit [12] of the ETM Control Register, see *Checking support for cycle-accurate tracing, ETMv3.3 and later* on page 3-28.
Data trace options[c]	Writing 1s to bits [20:18,3:1] of the ETM Control Register, see *Checking which data tracing options are available, ETMv3.3 and later* on page 3-28.
Data address comparison	Reading bit [12] of the Configuration Code Extension Register.[d]
EmbeddedICE behavior control	Reading bit [21] of the Configuration Code Extension Register.[d]
EmbeddedICE inputs to Trace Start/Stop block	Reading bit [20] of the Configuration Code Extension Register.[d]
Alternative address compression	Reading bit [20] of the ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71.
OS Lock mechanism	Reading bit [0] of the OS Lock Status Register, see *OS Lock Status Register (OSLSR), ETMv3.3 and later* on page 3-82.
Secure non-invasive debug	Reading bits [3:2] of the Authentication Status Register, see *Authentication Status Register (AUTHSTATUS), ETMv3.2 and later* on page 3-92.
Context ID tracing	Testing whether you can set bits [15:14] of the ETM Control Register to b11.[b]

a. See *ETM Configuration Code Register* on page 3-29.
b. See *ETM Control Register* on page 3-20.
c. Data address tracing, data value tracing, CPRT tracing, data-only trace mode.
d. See *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.

In addition to the information in Table A-30 on page A-14 and Table A-31 on page A-15:

- It is IMPLEMENTATION DEFINED which combinations of Port size and Port mode are supported. To test whether a particular combination is supported:
  - write the required values to bits [21,6:4] (Port size) and bits [13,17:16] of the ETM Control Register.
  - read bits [11:10] of the System Configuration Register to see if the selected port mode and port size are supported.
- Before ETMv3.3, some of the behavior of the address range comparators is IMPLEMENTATION DEFINED when the Exact match bit of the Address Access Type Register is set to 1. For more information see *Behavior of address comparators* on page 2-49.

Appendix B
Architecture Version Information

This appendix describes the major architecture changes for the ETM. It contains the following sections:

B.1 ETMv1

This section describes the major changes between the ETMv1 architecture versions. These changes are described in:

- *ETMv1.0 to ETMv1.1*
- *ETMv1.1 to ETMv1.2*
- *ETMv1.2 to ETMv1.3* on page B-3.

B.1.1 ETMv1.0 to ETMv1.1

The changes implemented between ETMv1.0 and ETMv1.1 are described in:

- *Programmer's model*.

Programmer's model

Changes to the programmer's model are:

- introduction of ETM Status Register, see *ETM Status Register, ETMv1.1 and later* on page 3-33
- use of bit [13], half-rate clocking, in the ETM Control Register, see *ETM Control Register* on page 3-20.

B.1.2 ETMv1.1 to ETMv1.2

The changes implemented between ETMv1.1 and ETMv1.2 are described in:

- *Controlling tracing*
- *Programmer's model*
- *Signal protocol* on page B-3.

Controlling tracing

Changes to controlling tracing are:

- introduction of trace start and stop control, see *Derived resources* on page 2-10
- instruction executed and condition code test passed option to single address comparators, see *Single address comparators* on page 2-5
- instruction executed and condition code test failed option to single address comparators, see *Single address comparators* on page 2-5.

Programmer's model

Changes to the programmer's model are:

- introduction of System Configuration Register, see *System Configuration Register, ETMv1.2 and later* on page 3-35

- introduction of Trace Start/Stop Resource Control Register, see *Trace Start/Stop Resource Control Register, ETMv1.2 and later* on page 3-37
- introduction of **TraceEnable** Control 2 Register, see *TraceEnable Control 2 Register, ETMv1.2 and higher* on page 3-38
- use of bits [17:16], port mode, in the ETM Control Register, see *ETM Control Register* on page 3-20
- use of bits [15:14], ContextIDsize, in the ETM Control Register, see *ETM Control Register* on page 3-20
- use of bit [2], current status of the trace start/stop resource, in the ETM Status Register, see *ETM Status Register, ETMv1.1 and later* on page 3-33
- use of bit [1], **TCK**-synchronized version of the ETM Programming bit, in the ETM Status Register, see *ETM Status Register, ETMv1.1 and later* on page 3-33
- use of bit [25], Trace start/stop enable, in the **TraceEnable** Control 1 Register, see *TraceEnable Control 1 Register* on page 3-39
- change to use of bits [6:5], Data value comparison control, in the Address Access Type Register
- change to use of bits [2:0], Access type, in the Address Access Type Register.

Signal protocol

Changes to the signal protocol are:

- introduction of Context ID tracing, see *Context ID tracing* on page 4-16
- change to branch executed pipeline status (addition of Context ID information), see *Pipeline status and trace packet association in ETMv1* on page 5-8
- change to branch reason codes, *Branch reason codes* on page 5-11.

B.1.3 ETMv1.2 to ETMv1.3

The changes implemented between ETMv1.2 and ETMv1.3 are described in:

- *Programmer's model*
- *Signal protocol* on page B-4.

Programmer's model

The change to the programmer's model is:

- use of bit [8], **FIFOFULL** supported, in System Configuration Register, see *System Configuration Register, ETMv1.2 and later* on page 3-35.

Signal protocol

Changes to the signal protocol are:

- use of bit [8], **FIFOFULL** supported, in System Configuration Register, see *System Configuration Register, ETMv1.2 and later* on page 3-35
- Java code support, see *Java code* on page 5-9 and *Tracing Java code, ETMv1.3 only* on page 5-20.

B.2 ETMv2

This section describes the major changes between the ETMv1 and ETMv2 architecture versions. These changes are described in:

- *ETMv1.3 to ETMv2.0*
- *ETMv2.0 to ETMv2.1* on page B-6.

B.2.1 ETMv1.3 to ETMv2.0

The changes implemented between ETMv1.3 and ETMv2.0 are described in:

- *Controlling tracing*
- *Programmer's model*
- *Signal protocol* on page B-6.

Controlling tracing

Changes to controlling tracing are:

- use of Context ID comparators for trace filtering, see *Context ID comparators* on page 2-8
- optional start/stop trace resource, see *Derived resources* on page 2-10 and *Trace start/stop resource* on page 2-11
- consideration is given to advanced cores, see *Considerations for advanced cores, ETMv2 and later only* on page 2-69.

Programmer's model

Changes to the programmer's model are:

- introduction of read/write to bits [7:0] of the **FIFOFULL** Level Register, see *FIFOFULL Level Register* on page 3-43
- introduction of ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71
- use of bit [19], Filter (CPRT), in the ETM Control Register, see *ETM Control Register* on page 3-20
- change to functionality of bit [8], Branch output, in the ETM Control Register, see *ETM Control Register* on page 3-20
- use of bit [26], Trace start/stop block detection, in the ETM Configuration Code Register, see *ETM Configuration Code Register* on page 3-29
- use of bits [9:8], Context ID comparator control, in the Address Access Type Register
- use of bit [7], Exact match bit, in the Address Access Type Register
- change to functionality of bit [17], Count enable source, in the Counter Enable Register, see *Counter Enable Registers* on page 3-59

- introduction of Context ID comparator registers, see *Context ID comparator registers, ETMv2.0 and later* on page 3-66
- introduction of Synchronization Frequency Register, see *Synchronization Frequency Register, ETMv2.0 and later* on page 3-69.

Signal protocol

Changes to the signal protocol are:

- addition of pipeline status pin, see *ETMv1.x and ETMv2.x signals* on page 4-4 and *ETMv2 pipeline status signals* on page 6-2
- changes to generating and analyzing the trace, see *Rules for generating and analyzing the trace in ETMv2* on page 6-7
- introduction of *Trace Fifo Offsets* (TFOs), see *Trace FIFO offsets* on page 6-14
- introduction of trace packet types, see *Trace packet types* on page 6-8.

B.2.2 ETMv2.0 to ETMv2.1

The changes implemented between ETMv2.0 and ETMv2.1 are described in:

- *Programmer's model*
- *Signal protocol* on page B-7.

Programmer's model

Changes to the programmer's model are:

- change to functionality of bits [15:14], ContextIDsize, in the ETM Control Register, see *ETM Control Register* on page 3-20
- change to functionality of bits [6:4], Port size, in the ETM Control Register, see *ETM Control Register* on page 3-20
- change to functionality of bit [1], Monitor (CPRT), in the ETM Control Register, see *ETM Control Register* on page 3-20
- change to functionality of bits [31:16], Specific implementation code, in the System Configuration Register, see *System Configuration Register, ETMv1.2 and later* on page 3-35
- use of bit [17], No fetch comparisons, in the System Configuration Register, see *System Configuration Register, ETMv1.2 and later* on page 3-35
- use of bit [16], No load data, in the System Configuration Register, see *System Configuration Register, ETMv1.2 and later* on page 3-35
- change to functionality of bits [31:24], Implementer code, in the ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71.

Signal protocol

The change to the signal protocol is:

- support of imprecise data aborts, see *Imprecise data aborts, ETMv2.1 and later* on page 6-27.

B.3 ETMv3

This section describes the major changes between the ETMv2 and ETMv3 architecture versions. These changes are described in:

- *ETMv2.1 to ETMv3.0*
- *ETMv3.0 to ETMv3.1* on page B-9
- *ETMv3.1 to ETMv3.2* on page B-10
- *ETMv3.2 to ETMv3.3* on page B-11
- *ETMv3.3 to ETMv3.4* on page B-12.

B.3.1 ETMv2.1 to ETMv3.0

The changes implemented between ETMv2.1 and ETMv3.0 are described in:

- *Programmer's model*
- *Signal protocol.*

Programmer's model

Changes to the programmer's model are:

- control of CPRT tracing is by bit [19], Filter CPRT), and bit [1], MonitorCPRT, of the ETM Control Register, see *ETM Control Register* on page 3-20 and *Filter Coprocessor Register Transfers (CPRT) in ETMv3.0 and later* on page 2-28
- use of bit [18], Suppress data, in the ETM Control Register, see *ETM Control Register* on page 3-20 and *FIFOFULL Level Register* on page 3-43
- change to functionality of bit [7], Stall processor, in the ETM Control Register, see *ETM Control Register* on page 3-20
- introduction of trace collection on both clock edges, see *System Configuration Register, ETMv1.2 and later* on page 3-35.

Signal protocol

Changes to the signal protocol are:

- removal of **PIPESTAT** bus, see *ETMv3.x signals* on page 4-5
- new header types, see *Packet types* on page 7-3
- replacement of the FIFOFULL mechanism, see *Data suppressed packet* on page 7-51
- support for Jazelle, see *Jazelle data tracing* on page 7-52.

B.3.2 ETMv3.0 to ETMv3.1

The changes implemented between ETMv3.0 and ETMv3.1 are described in:

- *Programmer's model*
- *Signal protocol*.

Programmer's model

Changes to the programmer's model are:

- use of bit [20], Instruction trace disable in the ETM Control Register, see *ETM Control Register* on page 3-20
- change to functionality of the MMD Control Register, now named the ASIC Control Register, see *ASIC Control Register* on page 3-32
- change to functionality of bits [4:3], Watch size changed from size mask, in the Address Access Type Register.
- coprocessor access, see *Coprocessor access, ETMv3.1 and later* on page 3-4
- extended external inputs, see *Extended external input selectors* on page 2-12
- read/write access to registers, see *The ETM registers* on page 3-11.

Signal protocol

Changes to the signal protocol are:

- data only trace is enabled, see *Data-only mode, ETMv3.1 and later* on page 7-53 and *ETM Control Register* on page 3-20
- store misses are enabled, see *Definitions* on page 4-21
- support for `STREX` (ARMv6 and later), see *Store failed packet* on page 7-52
- The BE bit shows that the data was a BE-8 (ARMv6 and later) big-endian transfer, and that the bytes must be reversed to determine the value that was stored in memory, see *BE bit* on page 7-46.

B.3.3 ETMv3.1 to ETMv3.2

The changes implemented between ETMv3.1 and ETMv3.2 are described in:

- *Programmer's model* on page B-9
- *Signal protocol* on page B-9.

Programmer's model

Changes to the programmer's model include:

- Use of bit [19], Security Extensions in the ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71.
- Use of bit [18], 32-bit Thumb instructions, in the ETM ID Register, see *ETM ID Register, ETMv2.0 and later* on page 3-71.
- Use of bits [11:10], Security mode control, in the Address Access Type Registers, see *Address Access Type Registers* on page 3-51.
- Addition of event resource `0x6D`, indicating that the processor is in Non-secure state, in *Resource identification* on page 3-108.
- Addition of event resource `0x6E`, indicating that tracing is prohibited, in *Resource identification* on page 3-108.
- Support for memory-mapped register access, in *Memory-mapped access, ETMv3.2 and later* on page 3-6
- Core select functionality for sharing an ETM between two or more processors, in *ETM Control Register* on page 3-20 and *System Configuration Register, ETMv1.2 and later* on page 3-35.
- Modification to ETM Status Register, 0x04. The Programming bit does not read as set to 1 until the FIFO is empty, see *ETM Status Register, ETMv1.1 and later* on page 3-33.
- Addition of CoreSight Trace ID Register, `0x80`, in *CoreSight Trace ID Register, ETMv3.2 and later* on page 3-80.
- Implementation of CoreSight programmer's model, `0x3C0-0x3FF`, in *CoreSight support* on page 3-10.
- Clarification of behavior of IMPLEMENTATION SPECIFIC registers, `0x70-0x77`, in *implementation specific registers* on page 3-68.

Signal protocol

Changes to the signal protocol include:

- Addition of branch address continuation byte to indicate extended exception information and security level, in *Exception branch addresses packets* on page 7-24.

B.3.4 ETMv3.2 to ETMv3.3

The changes implemented between the ETMv3.2 and ETMv3.3 releases of this *Technical Reference Manual* are described in:

- *Programmer's model*
- *Signal protocol*
- *Clarification of descriptions of features from earlier ETM versions* on page B-12.

Programmer's model

Changes to the programmer's model include:

- The use of bit [24], Instrumentation resource control, in the ETM Control Register. See *ETM Control Register* on page 3-20.
- The use of bit [12] in the Configuration Code Extension Register to indicate that data address comparisons are not supported. See *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.
- The addition of an additional permitted value for the Core family field, bits [15:12], of the ETM ID Register. See *ETM ID Register, ETMv2.0 and later* on page 3-71.

  This additional value, 4'b1111, specifies that the core family is defined elsewhere. From ETMv3.3, this is the usual value of this field for ETM macrocells.
- The provision of power-down support. See:
  - *Power-down support, ETMv3.3 and later* on page 3-119
  - *Operating System Save and Restore Registers, ETMv3.3 and later* on page 3-81.
- The addition of new ARM and Thumb instructions to control the Instrumentation resources. See *Instructions for controlling the Instrumentation resources* on page 2-64.
- When a coprocessor interface to the ETM registers is implemented, it is now possible to access all of the ETM registers. See *Full access model, ETMv3.3 and later* on page 3-5.

Signal protocol

Changes to the signal protocol include:

- Addition of the format 4 P-header in cycle-accurate mode. See *P-header encodings in cycle-accurate mode* on page 7-6.
- Addition of optional Instrumentation resource functionality, that provides new event resources that can be controlled from software. See *Instrumentation resources, from ETMv3.3* on page 2-63, and *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76.
- Addition of support for the Thumb Execution Environment. See:
  - *Branch packets* on page 7-11
  - *Branch address packets for change of processor state* on page 7-31

  - — *Direct and indirect branches* on page 4-9.

- From ETMv3.3, it is IMPLEMENTATION DEFINED whether an ETM macrocell supports:
  - — data value and data address tracing, see *Data tracing options, ETMv3.3 and later* on page 7-54
  - — data suppression, see *Checking whether data suppression is supported, ETMv3.3 and later* on page 2-34
  - — cycle-accurate tracing, see *Support for cycle-accurate tracing, ETMv3.3 and later* on page 7-79
  - — data address comparisons, see *No data address comparator option, ETMv3.3 and later* on page 2-7.

Clarification of descriptions of features from earlier ETM versions

Changes in the ETMv3.3 issue of the Architecture Specification that do not describe changes in the ETM architecture include:

- Additional information on the tracing of the Thumb `CZB` instruction, see *Thumb CBZ and CBNZ instructions* on page 4-13.
- Clarification of the section *Branch address generation* on page 7-39.
- Clarification of **TraceEnable** behavior, indicating that data address comparators must not be used in **TraceEnable** exclude regions. See *TraceEnable and filtering the instruction trace* on page 2-20.
- Re-organization of information about address comparators. In particular, a lot of information that previously appeared in the section *Address comparator registers* on page 3-50 has been moved into the section *Address comparators* on page 2-36. This change provides a more convenient description of the address comparators in a single location.

For details of additional changes that are relevant to ETMv3.3 see *Clarification of descriptions of features from earlier ETM versions* on page B-14.

B.3.5 ETMv3.3 to ETMv3.4

The changes implemented between the ETMv3.3 and ETMv3.4 releases of this *Technical Reference Manual* are described in:

- *Programmer's model*
- *Signal protocol* on page B-13
- *Clarification of descriptions of features from earlier ETM versions* on page B-14.

Programmer's model

Changes to the programmer's model include:

1. The Synchronization Frequency Register can be implemented as a read-only register. This is because of a change in the signal protocol that permits an implementation to use a fixed trace synchronization frequency of 1024, see Item 1 in the section *Signal protocol* on page B-13.

For details, see *Synchronization Frequency Register, ETMv2.0 and later* on page 3-69.

2. The ETM ID Register is extended so that bit [20] indicates the encoding used for branch packets. See *ETM ID Register, ETMv2.0 and later* on page 3-71.

   Item 4 in the section *Signal protocol* summarizes the signal protocol change that permits the implementation of an alternative encoding for branch packets.

3. The Configuration Code Extension Register is extended, and two additional registers are introduced, to support additional features of the trace start/stop block and the EmbeddedICE watchpoint inputs, see:

   - *Configuration Code Extension Register, ETMv3.1 and later* on page 3-76
   - *Trace Start/Stop EmbeddedICE Control Register, ETMv3.4 and later* on page 3-78
   - *EmbeddedICE Behavior Control Register, ETMv3.4 and later* on page 3-79. This register is optional.

   Item 5 in the section *Signal protocol* summarizes the signal protocol change that correspond to these changes in the Programmer's model.

Signal protocol

Changes to the signal protocol include:

1. An ETM implementation can use a fixed trace synchronization frequency of 1024. In this case the Synchronization Frequency Register is implemented as a read-only register. For more information see Item 1 in the section *Programmer's model* on page B-12.

2. An ETM implementation for a processor core that complies with the ARMv7-M architecture must implement extensions to the exception branch packets, to support the extended extension information from these processors. See *Extended exception handling, from ETMv3.4* on page 7-33.

3. An ETM implementation for a processor core that complies with the ARMv7-M architecture must implement two new packet types, for *exception entry* and *return from exception*. See:

   - *Automatic stack push on exception entry and pop on exception exit* on page 7-57
   - *Tracing return from an exception* on page 7-59.

4. An ETM implementation can implement an alternative encoding for all branch packets. This alternative encoding provides address compression, where appropriate, when tracing exception branches. See *Branch packet formats with the alternative address compression scheme* on page 7-18.

   Note

   This alternative encoding is not backwards-compatible. From ETMv3.4, it is IMPLEMENTATION DEFINED whether an ETM supports the original branch packet encoding or the new alternative encoding. See Item 1 in the section *Clarification of descriptions of features from earlier ETM versions* on page B-14 for more information about the original branch packet encoding.

5. The Trace Start/Stop block is enhanced, to permit the use of EmbeddedICE watchpoint inputs as start or stop signals to the block. In addition, the number of EmbeddedICE watchpoint inputs becomes IMPLEMENTATION DEFINED, to any value between 0 and 8. In previous versions of the ETM architecture, this number is fixed as two. Also, an implementation must either permit the behavior of these inputs to be configured, or follow specific requirements for their behavior. See:
   - *The trace start/stop block* on page 2-23
   - *Behavior of EmbeddedICE inputs, from ETMv3.4* on page 7-62.

   See item 3 in the section *Programmer's model* on page B-12 for details of the register changes associated with these changes.

Clarification of descriptions of features from earlier ETM versions

Changes in the ETMv3.4 issue of the Architecture Specification that do not describe changes to the ETM architecture include:

1. To complement the explanation of the alternative encoding of branch packets, referred to in Item 4 in the section *Signal protocol* on page B-13, additional explanation is given for the original scheme for encoding branch packets. See *Branch address compression, original scheme* on page 7-16.

2. Clarification of the operation of data value comparators, see *Operation of data value comparators* on page 2-56.

3. A summary of the usual ETM register access modes for ETM implementations is given in *ETM register access models* on page 3-7.

4. A description of the access controls that can apply to ETM register accesses is given in *Access permissions for ETM registers* on page 3-128.

Glossary

This glossary describes some of the terms used in technical documents from ARM Limited.

Abort A mechanism that indicates to a core that the value associated with a memory access is invalid. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a prefetch or data abort, and an internal or external abort.

See also Data abort, External abort and Prefetch abort.

A-sync *See* Alignment synchronization.

Aligned A data item stored at an address that is divisible by the number of bytes that defines the data size is said to be aligned. Aligned words and halfwords have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore stipulate addresses that are divisible by four and two respectively.

Alignment synchronization (A-sync) header
A sequence of bytes that enables the decompressor to byte-align the trace stream and determine the location of the next header.

Address Packet Offset (APO)
In ETMv1 the *Address Packet Offset* (APO) is used by the decompressor to synchronize between the pipeline status signals (**PIPESTAT**) and the trace packet signals (**TRACEPKT**).

APO *See* Address Packet Offset

ARM instruction
A word that specifies an operation for an ARM processor in ARM state to perform. ARM instructions are word-aligned.

See also ARM state, Thumb instruction, ThumbEE instruction.

ARM state
An operating state of the processor, in which it executes 32-bit ARM instructions.

See also ARM instruction, Thumb state, ThumbEE state, Jazelle architecture.

BE-8 Big-endian view of memory in a byte-invariant system.

See also BE-32, LE, Byte-invariant and Word-invariant.

BE-32 Big-endian view of memory in a word-invariant system.

See also BE-8, LE, Byte-invariant and Word-invariant.

Big-endian
Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

See also Little-endian and Endianness.

Big-endian memory
Memory in which:

- a byte or halfword at a word-aligned address is the most significant byte or halfword in the word 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 memory.

Branch folding
A technique where, on the prediction of most branches, the branch instruction is completely removed from the instruction stream presented to the execution pipeline. Branch folding can significantly improve the performance of branches, taking the CPI for branches below 1.

Branch phantom
The condition codes of a predicted taken branch.

See also Branch folding.

Branch prediction
The process of predicting if conditional branches are to be taken or not in pipelined processors. Successfully predicting if branches are to be taken enables the processor to prefetch the instructions following a branch before the condition is fully resolved. Branch prediction can be done in software or by using custom hardware. Branch prediction techniques are categorized as static, in which the prediction decision is decided before run time, and dynamic, in which the prediction decision can change during program execution.

Breakpoint
A mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted the programmer to enable inspection of register contents, memory locations, and/or variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested.

See also Watchpoint.

Byte-invariant
In a byte-invariant system, the address of each byte of memory remains unchanged when switching between little-endian and big-endian operation. When a data item larger than a byte is loaded from or stored to memory, the bytes making up that data item are arranged into the correct order depending on the endianness of the memory access.

The ARM architecture supports byte-invariant systems in ARMv6 and later versions. When byte-invariant support is selected, unaligned halfword and word memory accesses are also supported. Multi-word accesses are expected to be word-aligned.

See also Word-invariant.

Context The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the physical address range that it can access in memory and the associated memory access permissions.

See also Fast context switch.

Context ID
A 32-bit value accessed through CP15 register 13 that is used to identify and differentiate between different code streams.

CoreSight
The infrastructure for monitoring, tracing, and debugging a complete system on chip.

CPI *See* Cycles per instruction.

CPSR *See* Current Program Status Register.

Current Program Status Register (CPSR)
The register that holds the current operating processor status.

Cycles Per instruction (CPI)
Cycles per instruction (or clocks per instruction) is a measure of the number of computer instructions that can be performed in one clock cycle. This figure of merit can be used to compare the performance of different CPUs against each other. The lower the value, the better the performance.

Data abort
An indication from a memory system to the core of an attempt to access an illegal data memory location. An exception must be taken if the processor attempts to use the data that caused the abort.

See also Abort, External abort, and Prefetch abort.

Data instruction
An instruction that passed its condition code test and might have caused a data transfer, for example `LDM` or `MRC`.

Data synchronization (D-sync)
Data addresses output in full to enable decompression of partial addresses output in the future.

Debugger
A debugging system that includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.
An application that monitors and controls the operation of a second application. Usually used to find errors in the application program flow.

D-Sync *See* Data synchronization.

Embedded Trace Buffer (ETB)
The ETB provides on-chip storage of trace data using a configurable sized RAM.

Embedded Trace Macrocell (ETM)
A hardware macrocell that, when connected to a processor core, outputs instruction and data trace information on a trace port. The ETM provides processor driven trace through a trace port compliant to the ATB protocol.

EmbeddedICE logic
An on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

Endianness
Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

See also Little-endian and Big-endian.

ETB *See* Embedded Trace Buffer.

ETM *See* Embedded Trace Macrocell.

Event 1 (Simple): An observable condition that can be used by an ETM to control aspects of a trace.

2 (Complex): A boolean combination of simple events that is used by an ETM to control aspects of a trace.

Event resource
A configurable ETM resource such as an address comparator or a counter. Used when configuring an *event*.

Exception
A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.

Exception vector
See Interrupt vector.

External abort

An indication from an external memory system to a core that the value associated with a memory access is invalid. An external abort is caused by the external memory system as a result of attempting to access invalid memory.

See also Abort, Data abort and Prefetch abort.

Fast Context Switch Extension (FCSE)

Modifies the behavior of an ARM memory system to enable multiple programs running on the ARM processor to use identical address ranges, while ensuring that the addresses they present to the rest of the memory system differ. From ARMv6, use of the FCSE is deprecated, and the FCSE is optional in ARMv7.

FCSE *See* Fast Context Switch Extension.

Half-rate clocking (in ETM)

Dividing the trace clock by two so that the TPA can sample trace data signals on both the rising and falling edges of the trace clock. The primary purpose of half-rate clocking is to reduce the signal transition rate on the trace clock of an ASIC for very high-speed systems.

I-sync *See* Instruction synchronization.

IMPLEMENTATION DEFINED

The behavior is not architecturally defined, but must be defined and documented by individual implementations.

IMPLEMENTATION SPECIFIC

The exact behavior is not architecturally defined, and does not have to be documented by individual implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

Imprecise Tracing

A filtering configuration where instruction or data tracing can start or finish earlier or later than expected. Most cases cause tracing to start or finish later than expected.

For example, if **TraceEnable** is configured 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** configuration always results in imprecise tracing.

See the descriptions of **TraceEnable** and **ViewData** in Chapter 2 *Controlling Tracing*.

Instruction synchronization (I-sync)

Full output of the current instruction address and Context ID on which later trace is based.

Interrupt vector

One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt handler.

Jazelle architecture
The ARM Jazelle architecture extends the Thumb and ARM operating states by adding a Jazelle state to the processor. Instruction set support for entering and exiting Java applications, real-time interrupt handling, and debug support for mixed Java/ARM applications is present. When in Jazelle state, the processor fetches and decodes Java bytecodes and maintains the Jazelle operand stack.

See also ARM state, Thumb state, ThumbEE state.

Jazelle RCT (Jazelle Runtime Compiler Target)
An extension to the ARM architecture targeting execution environments, such as Java or .NET Compact Framework. Jazelle RCT provides enhanced support for Ahead-Of-Time (AOT) and Just-In-Time (JIT) compilation. It extends the Thumb instruction set, and introduces a new processor state, ThumbEE.

See also ThumbEE state.

Joint Test Action Group (JTAG)
The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.

JTAG *See* Joint Test Action Group.

LE Little endian view of memory in both byte-invariant and word-invariant systems.

See also Byte-invariant and Word-invariant.

Little-endian
Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

See also Big-endian and Endianness.

Little-endian memory
Memory in which:

- a byte or halfword at a word-aligned address is the least significant byte or halfword in the word at that address
- a byte at a halfword-aligned address is the least significant byte in the halfword at that address.

See also Big-endian memory.

Macrocell
A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as a processor, an ETM, and a memory block) plus application-specific logic.

Match Resources match for one or more cycles when the condition they have been programmed to check for occurs.

Nested Vectored Interrupt Controller (NVIC)
This is an interrupt controller that forms part of the ARMv7-M architecture.

NVIC *See* Nested Vectored Interrupt Controller.

P-header
Provides pipeline status information as part of the data stream without using dedicated **PIPESTAT** signals.

Packet A number of bytes of related data, consisting of a header byte and zero or more payload bytes.

Packet header
The first byte of an ETM *packet* that specifies the packet type and how to interpret the following bytes in the packet.

Prefetch abort
An indication from a memory system to the core that an instruction has been fetched from an illegal memory location. An exception must be taken if the processor attempts to execute the instruction. A prefetch abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

See also Data abort, External abort and Abort.

Prohibited region
A period of core execution during which tracing is not permitted, for example because the processor is in Secure state.

RAZ *See* Read-As-Zero fields.

Read-As-Zero fields (RAZ)
Appear as zero when read.

Reserved
A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces UNPREDICTABLE results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are IMPLEMENTATION SPECIFIC. All reserved bits not used by the implementation must be written as zero and are Read-As-Zero.

TAP *See* Test Access Port.

TCD *See* Trace Capture Device.

Test Access Port (TAP)
The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

TFO *See* Trace FIFO Offset.

Thumb instruction
One or two halfwords that specify an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned. In the original Thumb instruction set, all instructions are 16-bit. Thumb-2 technology, introduced in ARMv6T2, makes it possible to extend the original Thumb instruction set with many 32-bit instructions.

See also ARM instruction, Thumb state, ThumbEE instruction.

Thumb state
An operating state of the processor, in which it executes 16-bit and 32-bit Thumb instructions.

See also ARM state, Thumb instruction, ThumbEE state, Jazelle architecture.

ThumbEE instruction
One or two halfwords that specify an operation for an ARM processor in ThumbEE state to perform. ThumbEE instructions must be halfword-aligned.

ThumbEE is a variant of the Thumb instruction set that is designed as a target for dynamically generated code, that is, code compiled on the device either shortly before or during execution from a portable bytecode or other intermediate or native representation.

See also ARM instruction, Thumb instruction, ThumbEE state.

ThumbEE state
An operating state of the processor, in which it executes 16-bit and 32-bit ThumbEE instructions.

See also ARM state, Thumb state, ThumbEE instruction, Jazelle architecture.

TPA *See* Trace Port Analyzer.

Trace Capture Device (TCD)
A generic term for Trace Port Analyzers, logic analyzers, and Embedded Trace Buffers.

Trace FIFO Offset
ETMv2 generates *Trace FIFO Offsets* (TFO) to enable the decompressor to synchronize the pipeline status (**PIPESTAT**) and FIFO output (**TRACEPKT**) signals. For more information see *Trace FIFO offsets* on page 6-14.

Trace packet header
Indicates the type of trace packet being output on the **TRACEPKT** pins, and specifies how to interpret the subsequent bytes of the trace packet.

Trace port
A port on a device, such as a processor or ASIC, that is used to output trace information.

Trace Port Analyzer (TPA)
A hardware device that captures trace information output on a trace port. This can be a low-cost product designed specifically for trace acquisition, or a logic analyzer.

Unaligned
A data item stored at an address that is not divisible by the number of bytes that defines the data size is said to be unaligned. For example, a word stored at an address that is not divisible by four.

See also Aligned.

UNDEFINED
Indicates an instruction that generates an Undefined Instruction exception.

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. UNKNOWN values must not be documented or promoted as having a defined value or effect.

UNPREDICTABLE

Means that the behavior of the ETM cannot be relied on. Such conditions have not been validated. When applied to the programming of an event resource, the only effect of the UNPREDICTABLE behavior is that the output of that event resource is UNKNOWN.

UNPREDICTABLE behavior can affect the behavior of the entire system, because the ETM can cause the core to enter debug state, and external outputs can be used for other purposes.

Virtual address

Is an address generated by an ARM processor. For processors that implement a *Protected Memory System Architecture* (PMSA), the virtual address is identical to the physical address

Watchpoint

A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to enable inspection of register contents, memory locations, and variable values when memory is written, to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested.

See also Breakpoint.

