ETM7

(Rev 1)

Technical Reference Manual

ARM®

ARM DDI 0158D

ETM7
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
4 February 2000	A	First release.
1 September 2000	B	Second release.
29 May 2001	C	New chapter added containing signal guidelines.
13 June 2001	D	Error corrected in signal guidelines chapter.

Proprietary Notice

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
ARM7 Embedded Trace Macrocell (ETM7) Technical Reference Manual

List of Figures
ETM7 Technical Reference Manual

Preface

This preface introduces the *ARM7 Embedded Trace Macrocell* (ETM7) (Rev 1) and its reference documentation. It contains the following sections:

- *About this document*
  Feedback on page xi.

About this document

Intended audience

This document has been written for hardware and software engineers who wish to incorporate an ARM core based product having instruction and data trace capability into their hardware and/or software design.

Using this manual

This document is organized into the following chapters:

Chapter 1 ***Introduction***

the ETM7 block and functional diagrams.

Chapter 2 ***Accessing ETM7 Registers***

Chapter 3 ***Integrating the ETM7***

Chapter 4 ***Memory Map Decode Interface***

Read this chapter for details of the Memory Map Decode interface used to integrate the ETM7 with memory-mapped peripherals.

Chapter 5 ***ASIC Trace Validation***

ASIC design.

Chapter 6 ***Software Considerations for Trace***

Accessing ETM7 Registers

Appendix A

Appendix B ***Differences between ETM7 versions***

the changes to the programmer's model.

Typographical conventions

bold Highlights ARM processor signal names, and interface elements, such as menu names and buttons. Also used for terms in descriptive lists, where appropriate.

Highlights special terminology, cross-references, and citations.

`typewriter`

`typewriter`

`typewriter italic`

`typewriter bold`

Further reading

`http://www.arm.com`

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

Embedded Trace Macrocell Specification (ARM IHI 0014)

- *Multi-ICE User Guide* (ARM DUI 0048).

Other publications

Trace Port Analysis for ARM ETM Users Guide (Agilent Publications - publication number E5903-97000).

Feedback

Feedback on the ARM7 Embedded Trace Macrocell (ETM7)

Feedback on this book

errata@arm.com

Chapter 1
Introduction

1.1 About the ETM7

ARM9 Embedded Trace Macrocell (ETM9) Technical Reference Manual *Embedded Trace Macrocell* *Trace Port Analyzer* *Embedded Trace Macrocell Specification*. Where the expression ETM appears in the text it refers to a nonspecific ETM (ETM7 or ETM9). This document assumes that the ETM7 is a Rev 1 version. For details of differences between Rev 0 and Rev 1, see Appendix B *Differences between ETM7 versions*

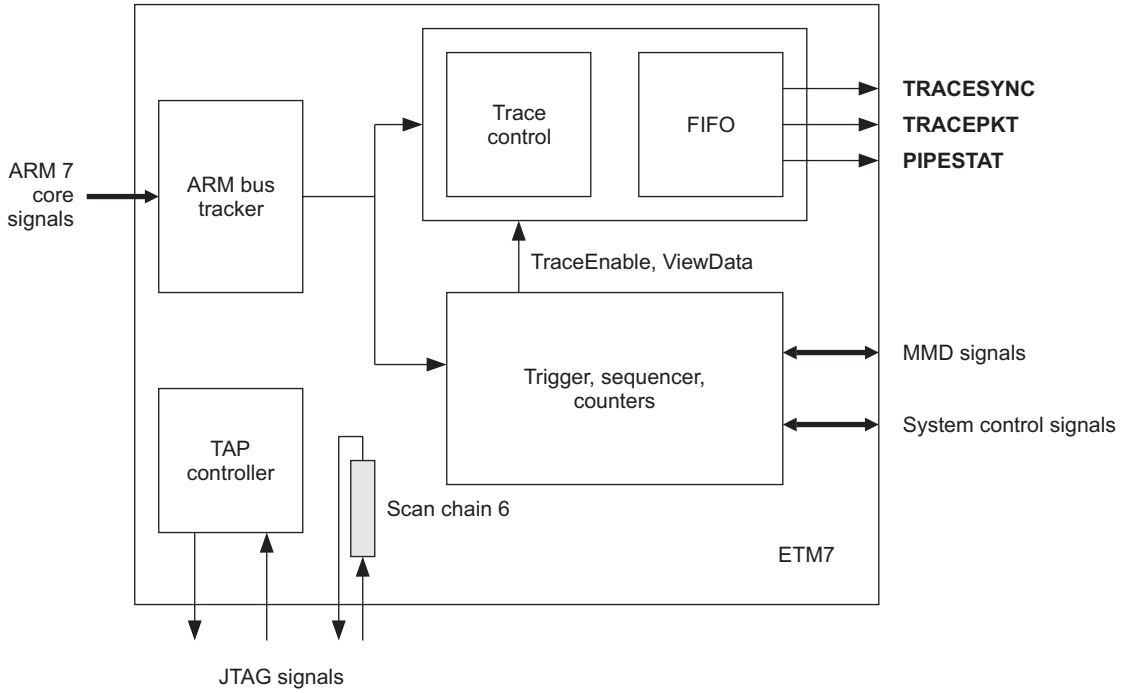

Figure 1-1 Block diagram of the ETM7

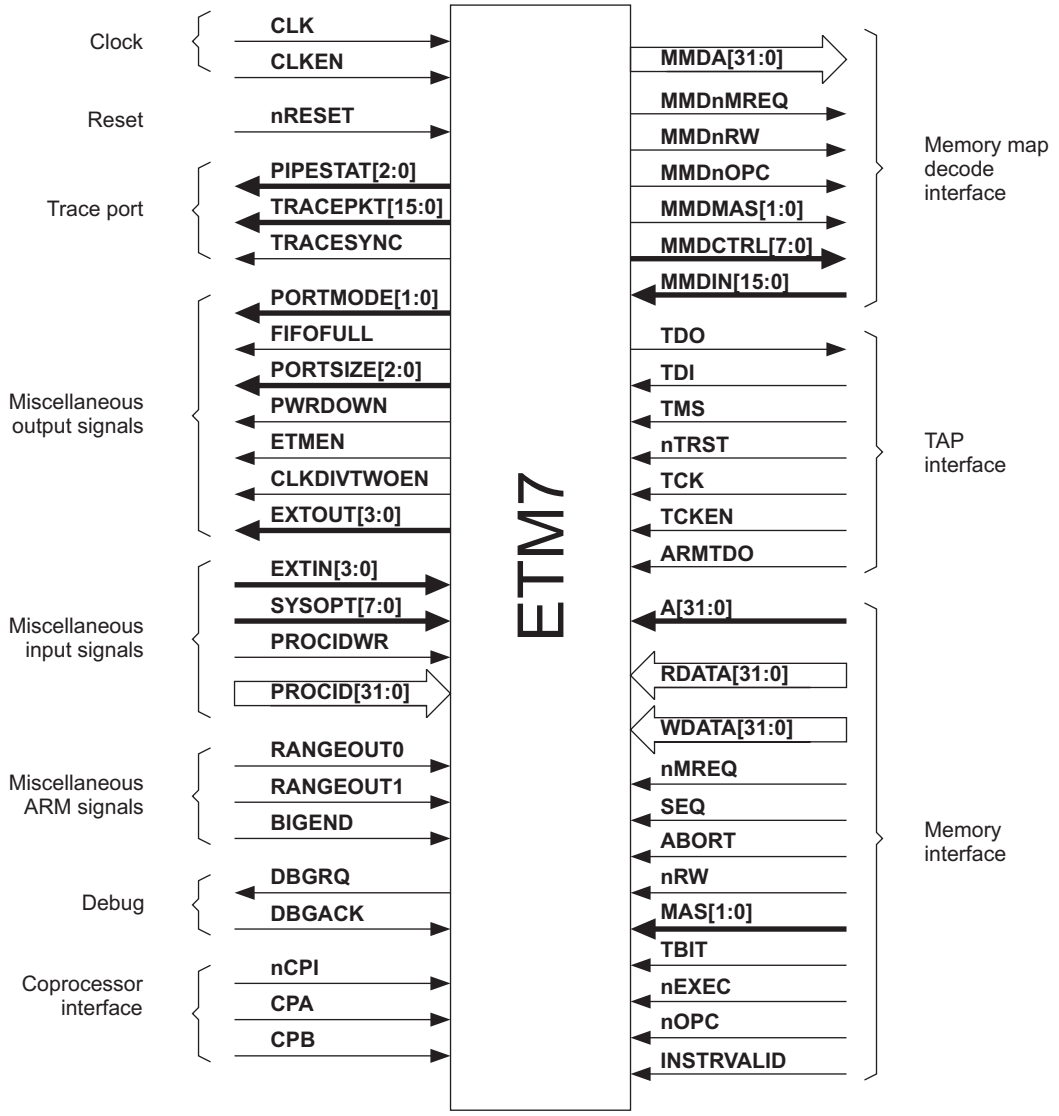

Figure 1-2 Functional diagram of the ETM7

Chapter 2
Accessing ETM7 Registers

Programming and reading ETM7 registers

2.1 TAP interface

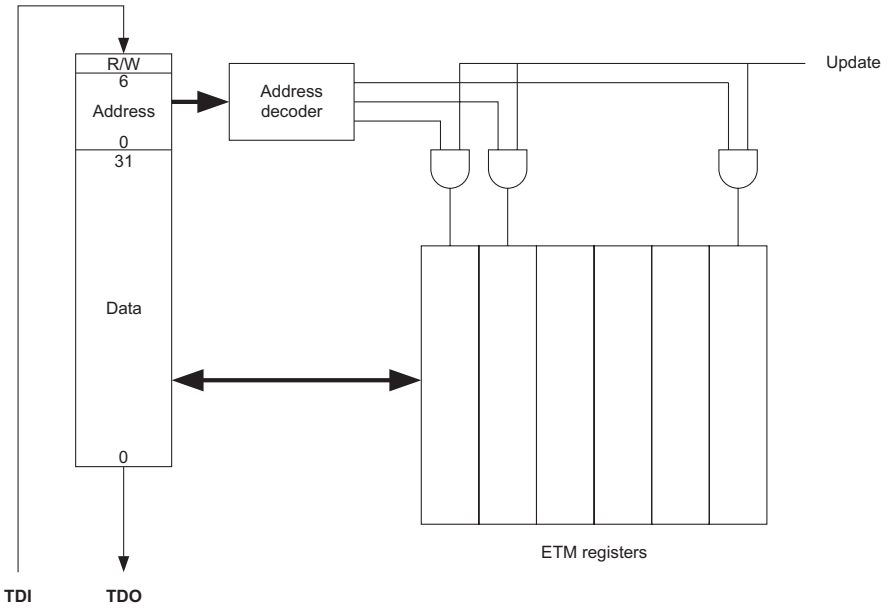

Figure 2-1 ETM7 TAP structure

The ETM7 uses scan chain 6. For details, refer to the
.

2.2 Programming and reading ETM7 registers

All registers in the ETM7 are programmed through a JTAG interface. The interface is an extension of the ARM TAP controller, and is assigned scan chain 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.

The general arrangement of the ETM7 JTAG registers is shown in Figure 2-1 on page 2-2.

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.

For further details of ETM7 registers, see the .

Chapter 3
Integrating the ETM7

3.1 About integrating the ETM7

the PWRDOWN output on page 3-19), or to OR debug requests (see *Debug request output wiring*

3.1.1 ETM port names

ARM interfacing

3.2.1 ETM7 to ARM7 connection guide

Table 3-1 ETM7 to ARM signal connections

ETM7signalname	ARM signal name		
	ARM7TDMI-S	ARM7TDMI	ARM720T (Rev3)
A[31:0]	ADDR[31:0]	A[31:0]	ETMA[31:0]
ABORT	ABORT	ABORT	ETMABORT
ARMTDO	DBGTDO	N/C [a]	N/C a
BIGEND	CFGBIGEND	BIGEND	ETMBIGEND
CLK	CLK	MCLK [b]	ETMCLK
CLKEN	CLKEN	nWAIT	ETMCLKEN
CPA	CPA [c]	CPA [c]	ETMCPA
CPB	CPB [c]	CPB [c]	ETMCPB
DBGACK	DBGACK	DBGACK	ETMDBGACK
DBGRQ	DBGRQ	DBGRQ	DBGRQ
nMREQ	CPnMREQ	nMREQ	ETMnMREQ

SEQ	CPSEQ	SEQ	ETMSEQ
MAS[1:0]	**SIZE[1:0]**	**MAS[1:0]**	**ETMMAS[1:0]**
nCPI	**CPnCPI**	**nCPI**	**ETMnCPI**
nEXEC	**DBGnEXEC**	**nEXEC**	**ETMnEXEC**
nOPC	**CPnOPC**	**nOPC**	**ETMnOPC**
nRESET	**ETMnRESET** [d]	**ETMnRESET** d	**ETMnRESET** d
nRW	**WRITE**	**nRW**	**ETMnRW**
nTRST	**DBGnTRST**	**nTRST**	**nTRST**
PROCID[31:0]	-	-	**ETMPROCID[31:0]**
PROCIDWR	-	-	**ETMPROCIDWR**
RANGEOUT0	**DBGRNG[0]**	**RANGEOUT0**	**RANGEOUT[0]**
RANGEOUT1	**DBGRNG[1]**	**RANGEOUT1**	**RANGEOUT[1]**
RDATA[31:0]	**RDATA[31:0]**	**D[31:0]/DIN[31:0]** [e]	**ETMD[31:0]** e
TBIT	**CPTBIT**	**TBIT**	**ETMTBIT**
TCK	**CLK**	**TCK**	**TCK**
TCKEN	**DBGTCKEN**	V_{DD}	V_{DD}
TDI	**DBGTDI**	**TDI**	**TDI**
TDO	N/C a	**SDOUTBS**	**SDOUTBS**
TMS	**DBGTMS**	**TMS**	**TMS**
WDATA[31:0]	**WDATA[31:0]**	**D[31:0]/DOUT[31:0]** e	**ETMD[31:0]** e
INSTRVALID	**DBGINSTRVALID** [f]	**INSTRVALID** [f]	V_{DD} [f]

a. See *TAP interface wiring* on page 3-13
b. See *CLK and CLKEN* on page 3-10
c. See *Coprocessor control connections* on page 3-8
d. See *ETM reset* on page 3-11
e. See *Use of bidirectional buses* on page 3-5

f. See *INSTRVALID*

INSTRVALID

If you are using an ARM processor in which this signal is not available, (such as Rev 0 to 3a of ARM7TDMI or Rev 0 to 1 of ARM7TDMI-S) you must tie this ETM input HIGH.

3.2.3 Use of bidirectional buses

You can use the ETM7 in the ARM7TDMI and ARM720T (Rev3) system using a single bidirectional data bus. The **D[31:0]** bus must be connected to both the ETM7 **RDATA[31:0]** and **WDATA[31:0]** buses. In this case the ETM7 samples the data bus correctly for both read and write accesses. See Figure 3-1 on page 3-6 for details of an example system using bidirectional buses.

3.2.4 Coprocessor data bus connections

Figure 3-1 on page 3-6 shows the coprocessor data bus connections for a unidirectional bus based system having an ETM7 and an ARM7TDMI-S or ARM7TDMI core.

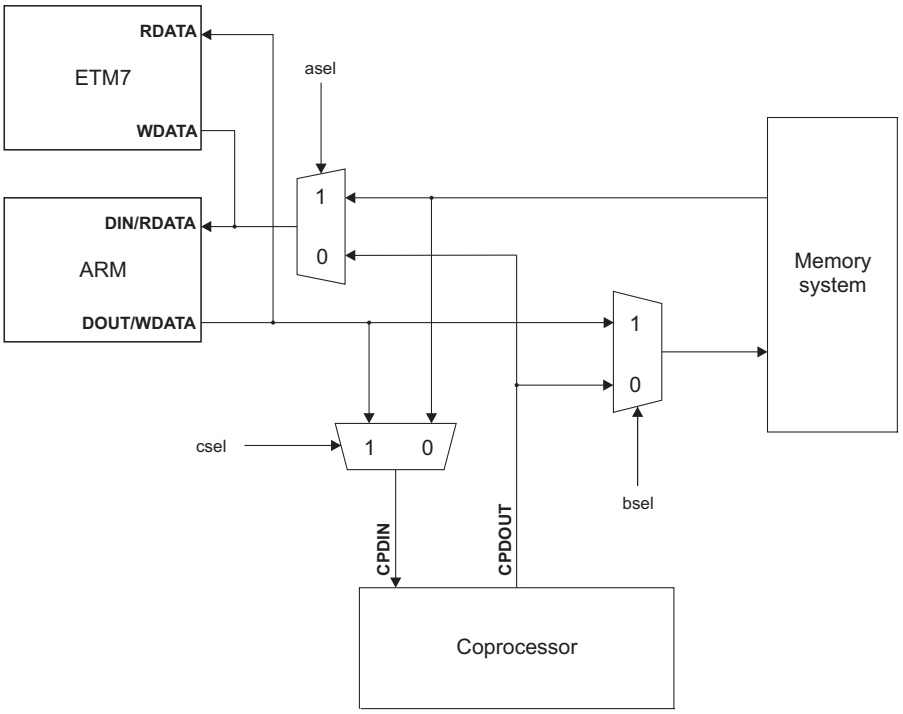

Figure 3-1 Coprocessor unidirectional data bus connections

```
        assign asel = ~(cprt | (cpdt & nRW_r));assign bsel = ~cpdt;assign csel
= cprt;  assign cpdt = ~nMREQ_r & ~CPA_r2 & nOPC_r;assign cprt = nMREQ_r &
SEQ_r;
```

Note

cpdt LDC STC

cprt

```
always @(posedge CLK)if (CLKEN)begin              nMREQ_r <= CPnMREQ; //Output
from ARM7TDMI-S          SEQ_r   <= CPSEQ; // Output from ARM7TDMI-SnOPC_r  <=
CPnOPC; // Output from ARM7TDMI-SnRW_r   <= WRITE; // Output from ARM7TDMI-S
CPA_r   <= CPA; // Input to ARM7TDMI-SCPA_r2  <= CPA_r;end
```

CPA

```
always @(negedge MCLK)if (nWAIT)begin             nMREQ_r <= nMREQ; // Output
from ARM7TDMI            SEQ_r   <= SEQ;   // Output from ARM7TDMInOPC_r  <=
nOPC;  // Output from ARM7TDMInRW_r   <= nRW;   // Output from ARM7TDMICPA_r2
<= CPA_r;        end

always @(posedge MCLK)if (nWAIT)CPA_r   <= CPA;     // Input to ARM7TDMI
```

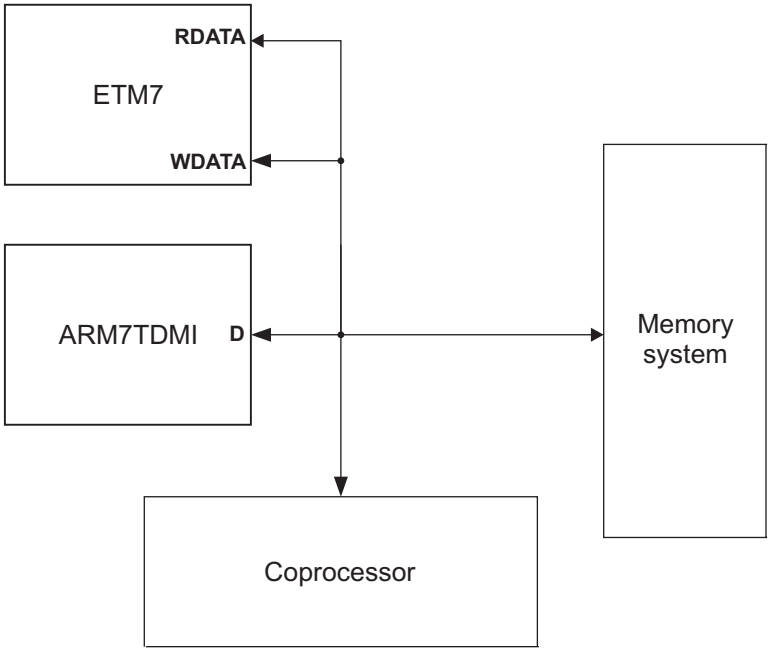

Figure 3-2 Coprocessor bidirectional data bus connections

3.2.5 Coprocessor control connections

CPA CPB

Note

CPA CPB

CPA CPB

ARM7TDMI

CPA CPB MCLK nWAIT
CPA CPB

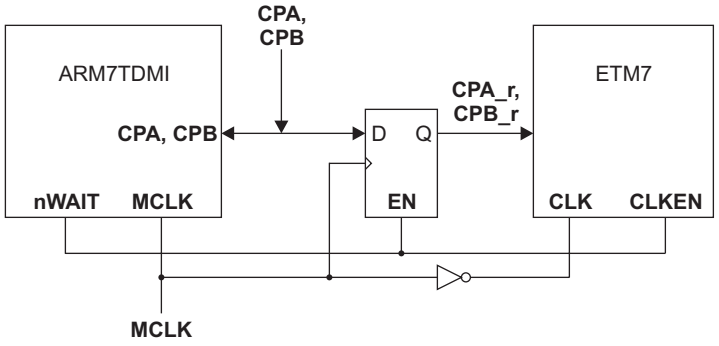

Figure 3-3 Retiming CPA and CPB in ARM7TDMI

ARM7TDMI-S

CPA CPB CLK CLKEN
CPA CPB

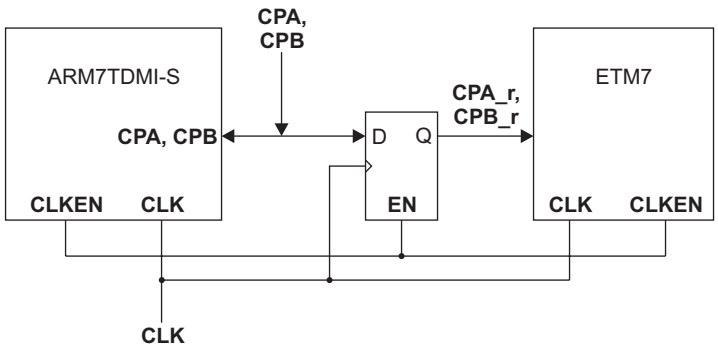

Figure 3-4 ARM7TDMI-S CPA and CPB connections

ARM720T

No retiming is necessary. The **ETMCPA** and **ETMCPB** outputs from the ARM720T must be connected directly to the **CPA** and **CPB** inputs of the ETM7.

3.3 Clocks and resets

The ETM7 has two sets of clock, clock enable, and reset inputs:

Main system controls:

- **CLK**
- **CLKEN**
- **nRESET**.

Scan chain controls:

- **TCK**
- **TCKEN**
- **nTRST**.

These are described under the following headings:

- *CLK and CLKEN*
  ETM reset
  TCK and TCKEN
  TAP reset

3.3.1 CLK and CLKEN

CLK **CLK,**
CLK **MCLK** **CLK**

CLKEN **CLK**
CLK **CLKEN**
CLK

Note

CLKEN
CLK

Connecting to an ARM7TDMI-S

CLK **CLKEN,**
CLK **CLKEN**

Connecting to an ARM7TDMI

CLK

MCLK CLKEN nWAIT

Connecting to an ARM720T

CLK ETMCLK

CLKEN ETMCLKEN

3.3.2 ETM reset

nRESET

nRESET

nTRST DBGnTRST

CLK TCK nTRST

CLK

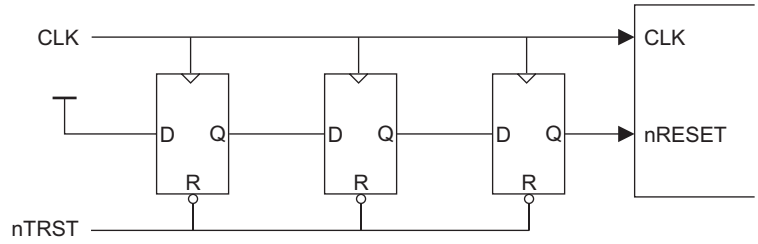

Figure 3-5 Synchronizing reset

nTRST CLK nTRST CLK

3.3.3 TCK and TCKEN

TCK TCKEN

TCK

CLK TCKEN

CLK

TCK **TCKEN**

CLK **TCK**

3.3.4 TAP reset

nTRST

Note

TCK **nTRST**

TMS

3.4 TAP interface wiring

SDOUTBS
ARMTDO

TDO

TDO

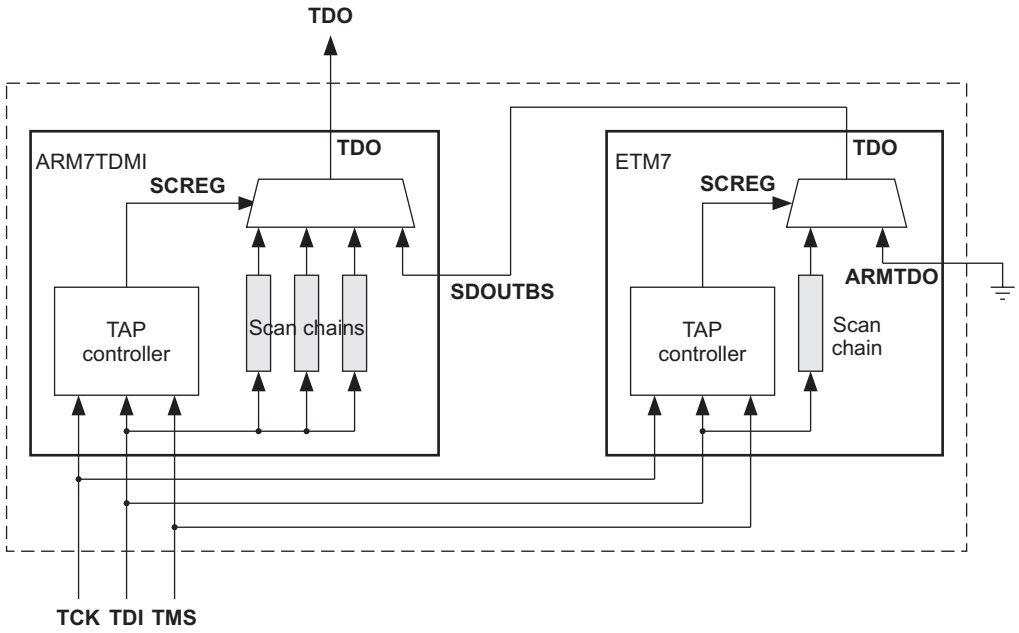

Figure 3-6 ARM7TDMI and ETM7 TAP interface structure

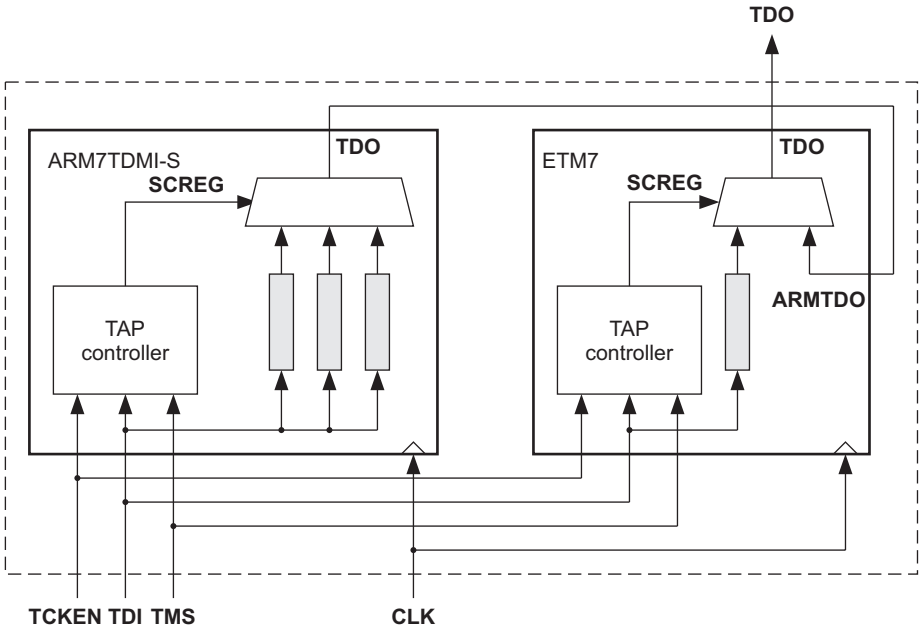

Figure 3-7 ARM7TDMI-S and ETM7 TAP interface structure

Note

nTRST

nTRST *Multi-ICE User Guide*

TDO

ARMTDO

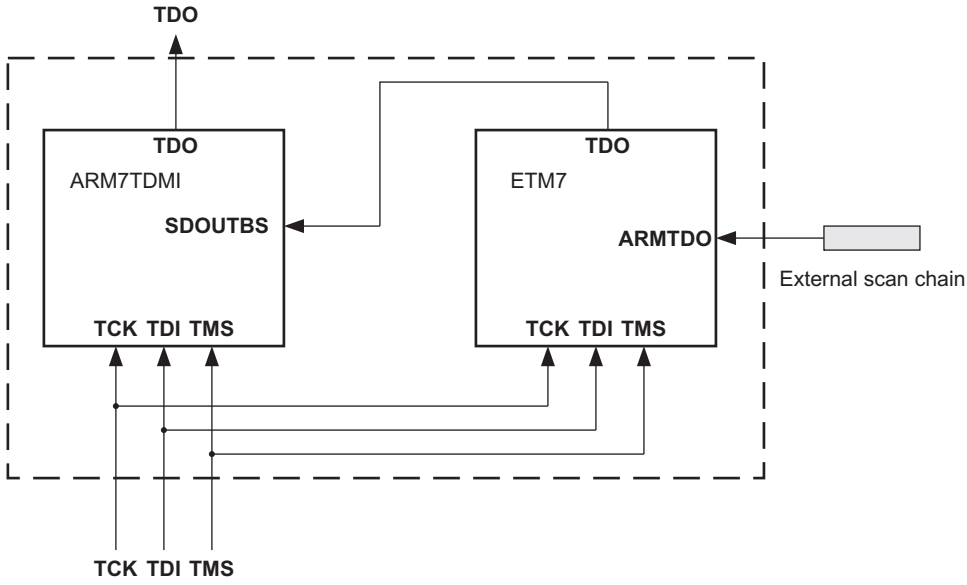

Figure 3-8 Using ETM7 and ARM7TDMI with an external scan chain

ARMTDO

3.4.1 IEEE 1149.1 compatibility

TDO **TCK** **TCKEN**

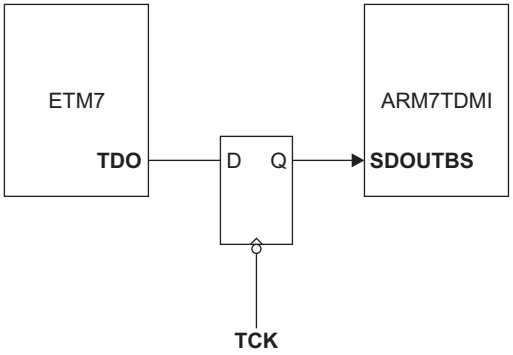

Figure 3-9 TDO output retiming circuit for IEEE 1149.1 compatibility

Note

ARMTDO
TCK

3.4.2 Multiprocessor TAP structure

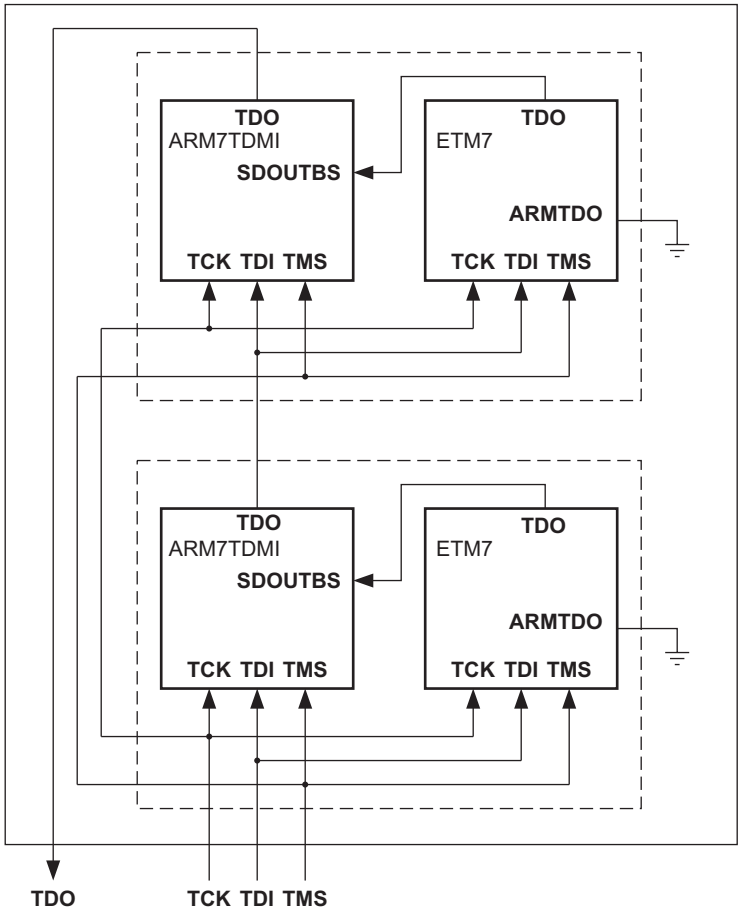

Figure 3-10 Multiprocessor TAP structure

Note

nTRST

nTRST

Multi-ICE User Guide

3.5 System control signals

3.5.1 Debug request output wiring

DBGRQ

DBGACK

Note

DBGRQ

DBGRQ **DBGRQ**
DBGRQ

DBGRQ

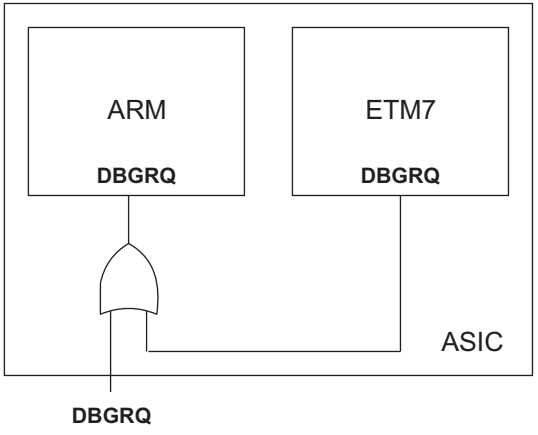

Figure 3-11 Combining DBGRQ inputs

3.5.2 Using the PWRDOWN output

PWRDOWN
CLK

nTRST **PWRDOWN**

PWRDOWN **CLK**

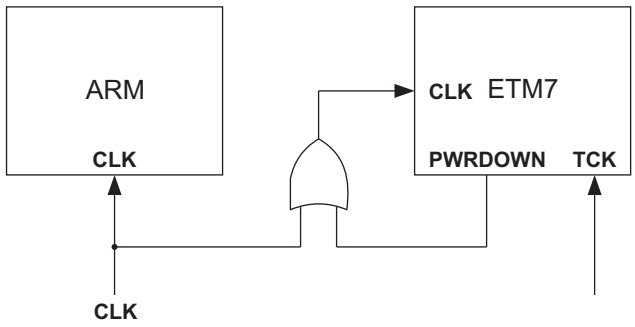

Figure 3-12 Clock gating the ETM7

PWRDOWN

ETMPWRDOWN

PWRDOWN **TCK** **PWRDOWN**

PWRDOWN **PWRDOWN**

PWRDOWN **CLK**

3.5.3 FIFOFULL

CLK

FIFOFULL

CLKEN

FIFOFULL

CLKEN

Embedded

Trace Macrocell Specification

FIFOFULL

FIFOFULL

FIFOFULL

Note

FIFOFULL **nIRQ** **nFIQ**

3.5.4 Using the process ID signals

PROCID[31:0] **PROCIDWR**

ETMPROCID[31:0] **ETMPROCIDWR**
PROCID[31:0] **PROCIDWR**

peripheral or another coprocessor, subject to software toolkit compatibility.

Note

If you are using a processor that does not provide these signals, you must tie the unused ETM7 inputs LOW.

Future versions of the ARM trace debug tools will support the process ID extensions, to allow tracing of dynamically loaded memory and overlay systems. See Chapter 6 *Software Considerations for Trace* for more details.

3.5.5 Using the system options bus

The system options bus is an 8-bit input bus called **SYSOPT[7:0]**. This is provided on ETM7 Rev 1. It allows you to specify whether certain trace features, such as half-rate clocking, are implemented on the ASIC. You must tie each of the bits of the bus to either GND or V_{DD}, depending on the features supported. The trace debug tools should read the state of this input bus using the JTAG interface, and adapt the user options offered accordingly. The bit meanings of the **SYSOPT** bus are shown in Table 3-2.

Table 3-2 SYSOPT bus settings

Bit number	Description
7	If HIGH, demultiplexed trace data format is supported.
6	If HIGH, multiplexed trace data format is supported.
5	If HIGH, normal trace data format is supported.
4	If HIGH, full-rate clocking is supported.
3	If HIGH, half-rate clocking is supported.
2:0	Maximum port width supported:000 = 4-bit only001 = 4/8-bit only010 = 4/8/16-bit.

Note

If the correct input is not supplied to the **SYSOPT** bus, the operation of the trace tools might be unreliable.

3.6 Trace port interfacing

Trace port interfacing is described under the following headings:

- *Trace port logic*
- *Single-processor tracing*
- *Dual-processor tracing* on page 3-23
- *Trace signal output timing* on page 3-25
- *PCB design guidelines* on page 3-27.

See *Modes of operation of the trace port* on page 3-28 for details of trace port operation.

3.6.1 Trace port logic

The trace information from the ETM7 is broadcast on the following signals:

- **PIPESTAT**
- **TRACESYNC**
- **TRACEPKT**.

In addition, three configuration signals are also provided:

- **ETMEN**
- **PORTSIZE**
- **PORTMODE**.

You can use these to configure the external logic connected to the trace port, under the control of the debugger.

3.6.2 Single-processor tracing

Some chips might not dedicate 16 pins to the **TRACEPKT** bus. Under some circumstances you might be able to reuse miscellaneous output signals from the chip as trace port pins. To allow this the ETM7 has the following outputs:

- **ETMEN**
- **PORTSIZE[2:0]**.

Figure 3-13 on page 3-23 shows one way in which the **TRACEPKT** pins can be shared with the ASIC pins.

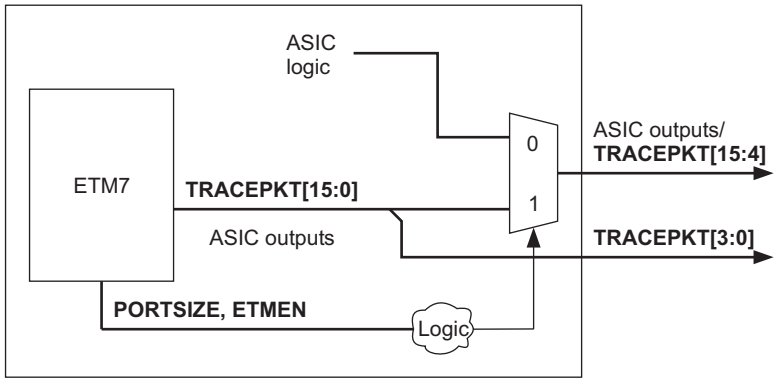

Figure 3-13 Reusing TRACEPKT pins

You can use the **PORTSIZE** and **ETMEN** signals to control on-chip logic to select between the normal ASIC output signals and the ETM7 trace port signals. This allows you to control the port width of the trace, and the number of pins used, from the debugger.

At reset the ETM7 is disabled (**ETMEN** LOW) and a 4-bit port is selected (**PORTSIZE** = 000). This ensures that normal operation of the ASIC is undisturbed.

Once the debug session starts, the debug tools can control **ETMEN** and **PORTSIZE** by programming the ETM control register.

3.6.3 Dual-processor tracing

Where there are multiple ARM processors on a single chip, it is recommended that each ARM processor has its own dedicated ETM.

The principle of controlling the port width, described in *Single-processor tracing* on page 3-22, can be extended to support dual-processor systems without dedicating a large number of pins to the trace signals.

The recommended dual trace configuration uses 21 pins on the ASIC, because this matches the 20 data pins and 1 clock pin defined in the trace connector specification. These pins are configured as 20 data pins and a single clock pin (assuming that both processors are clocked off a single clock).

This allows a number of configurations. Possible configurations for a single processor are shown in Table 3-3.

Table 3-3 Single-processor configurations

TRACEPKT	PIPESTAT	TRACESYNC	Total
16 trace packet	3 status	1 sync	20 data pins
8 trace packet	3 status	1 sync	12 data pins
4 trace packet	3 status	1 sync	8 data pins

You can, therefore, set a single trace port to allow the configurations shown in Table 3-4.

Table 3-4 Dual-processor trace port configurations

Processor A	Processor B
20 data	No trace
12 data	8 data
8 data	12 data
No trace	20 data

Pseudo-HDL to implement this is as follows:

```
if (PORTSIZE_B = 21)
    TRACE_DATA <= {PIPESTAT_B, TRACESYNC_B, TRACEPKT_B[15:0]}
else if (PORTSIZE_A = 13) and (PORTSIZE_B = 9)
    TRACE_DATA <= {PIPESTAT_B, TRACESYNC_B, TRACEPKT_B[3:0],
                   PIPESTAT_A, TRACESYNC_A, TRACEPKT_A[7:0]}
else if (PORTSIZE_A = 9) and (PORTSIZE_B = 13)
    TRACE_DATA <= {PIPESTAT_A, TRACESYNC_A, TRACEPKT_A[3:0],
                   PIPESTAT_B, TRACESYNC_B, TRACEPKT_B[7:0]}
else
    -- select A as the "master" for all other combinations.
    TRACE_DATA <= {PIPESTAT_A, TRACESYNC_A, TRACEPKT_A[15:0]}
end if
```

The *Embedded Trace Macrocell Specification* documents the target system connector pin allocations for single and dual-processor configurations. Support for the dual-processor pinouts is dependent on the debug tools and the TPA.

It is not recommended that you connect a single ETM7 to multiple ARM7 processors, because there is no general mechanism available to control the logic that selects which processor is connected to the single ETM.

3.6.4 Trace signal output timing

The trace connection to the TPA requires a clock, **TRACECLK,** to be exported from the ASIC. This is not generated by the ETM7, but must be generated by the system implementer. It is essential that you balance the clock to provide sufficient hold time on the trace data signals. The required hold times are defined in the *Embedded Trace Macrocell Specification*. It is essential that you maintain these hold times to guarantee reliable trace functionality.

It is recommended that the trace data signals are shifted by a clock phase from **TRACECLK**. This ensures that, on **TRACECLK** transitions, the trace data signals are stable, with a sufficient setup and hold time around the clock edge. Most TPAs require approximately 3ns of data valid time, and a hold time in the range 1 to 2ns, or greater, for reliable acquisition.

The ETM also supports a half-rate clocking mode, controlled by the **CLKDIVTWOEN** ETM7 output. When asserted, you should drive **TRACECLK** from the ETM clock (**CLK**) divided by two. When the debugger selects this mode, it also tells the TPA that it must sample the trace data signals on both edges of the clock, instead of only the rising edge.

Note

You do not have to implement half-rate clocking, and for low-speed systems (for example, less than 50MHz) the normal clocking mode is adequate. The primary purpose of half-rate clocking is to reduce the signal transition rate on the **TRACECLK** pin of the ASIC. This might be necessary to reduce electrical interference, to maintain **TRACECLK** signal integrity when using low drive strength pads, or for systems with very high clock speeds.

Figure 3-14 on page 3-26 shows an example circuit that implements both half-rate clocking and shifting of the trace data with respect to the clock.

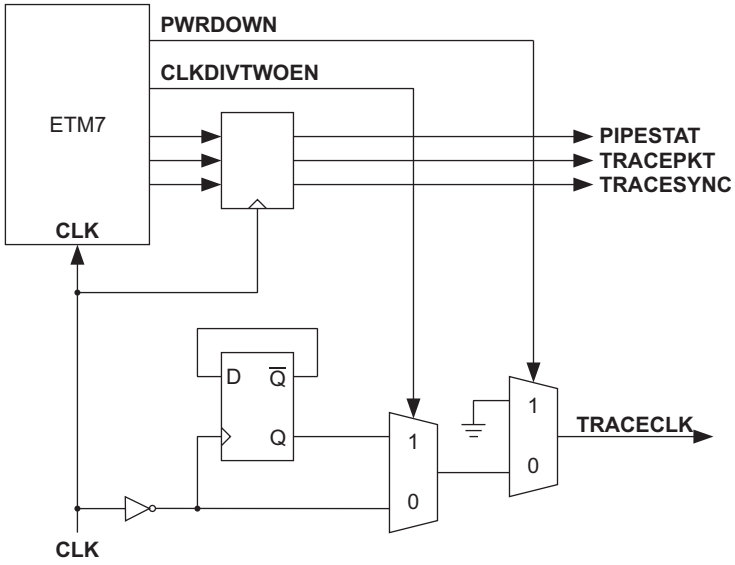

Figure 3-14 Trace output circuit with inverted clock

If your design flow does not allow you to invert the clock, you can also use falling edge D-types to retime the trace data signals, as shown in Figure 3-15 on page 3-27.

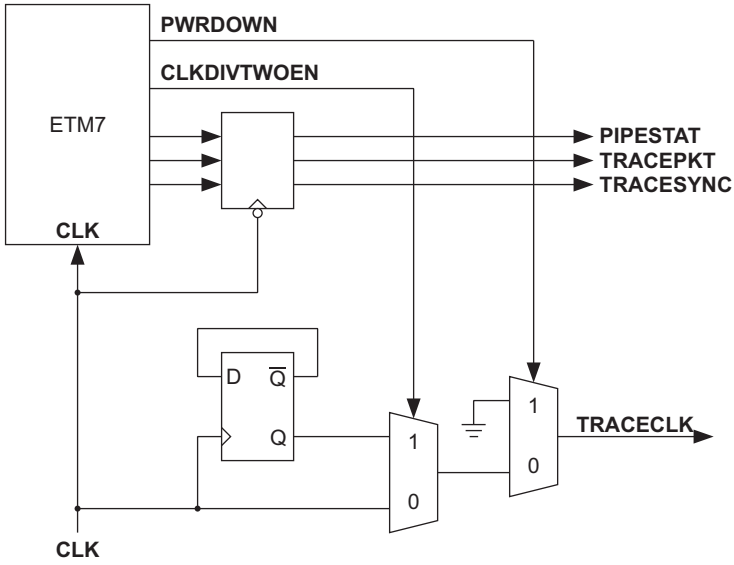

Figure 3-15 Trace output circuit using falling-edge D-types

It is recommended that you analyze carefully the timing of the trace data and clock signals to ensure the optimum setup and hold timing on the pins of the ASIC. It is also advisable to do detailed simulations of the output pads, package (for example, bond wires), PCB tracking, and logic analyzer loads to ensure the setup and hold times and signal integrity are met for the analyzer.

3.6.5 PCB design guidelines

See Chapter 2 *Accessing ETM7 Registers* for information about output pad selection and PCB design, including:

- trace signal termination
- PCB track lengths
- pad drive strength.

3.7 Modes of operation of the trace port

The **PORTMODE** output bus, which is available in ETM7 Rev 1, provides a copy of the contents of bits 17:16 of the ETM control register. This bus allows the trace debug tools to configure how the trace port signals from the ETM (**PIPESTAT**, **TRACEPKT**, and **TRACESYNC**) are mapped onto the trace port pins of the ASIC. The three modes of operation are described in the following sections:

- *Normal trace port signals*
- *Multiplexed trace port signals*
- *Demultiplexed trace port signals* on page 3-30.

3.7.1 Normal trace port signals

Normal mode tracing is the only mode of operation directly supported by Rev 0 of the ETM. Both normal and half-rate clocking can be supported in this mode, and for very high speed designs (greater than 100MHz) half-rate clocking is recommended to maintain the signal integrity of the clock.

3.7.2 Multiplexed trace port signals

This mode of operation multiplexes two trace data signals onto a single trace output pin. This means that the TPA must capture the data on both edges of **TRACECLK**. This scheme is only recommended for low-frequency designs (for example, less than 50MHz). This is because it is difficult to maintain the required setup and hold between **TRACECLK** and the trace data signals to the TPA.Half-rate clocking is not supported in this mode, because it already relies on the TPA capturing the state of the trace data pins twice per trace clock cycle.

The *ETM Specification* provides the trace connector pinout for this mode of operation.

You must pair up the trace signals as shown in Table 3-5 on page 3-29. Each row contains a separate pair of signals, one signal occurs on the rising edge of **TRACECLK** and the other on the falling edge. The **PIPESTAT** and **TRACEPKT[0]** signals are sampled first by the TPA to determine the trigger and trace storage qualification information.

Table 3-5 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 an 4-pin trace port connector	These pins are paired for a 6-pin trace port connector	**PIPESTAT[0]**	**TRACESYNC**
		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]**

Figure 3-16 shows the logic to implement multiplexed data trace signals.

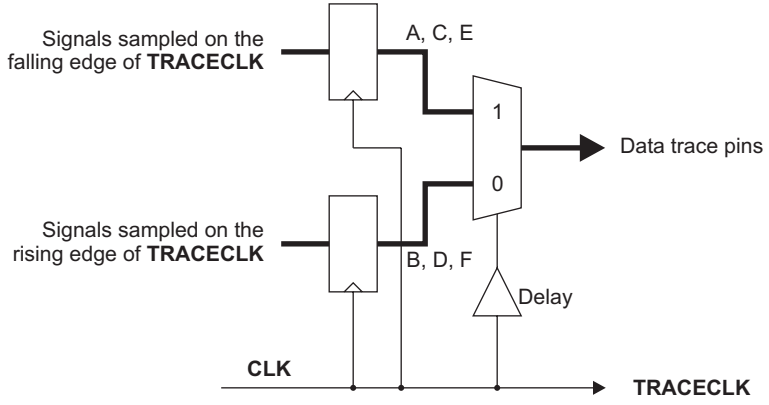

Figure 3-16 Multiplexing data trace signals

Figure 3-17 on page 3-30 shows the timing of the multiplexed signals.

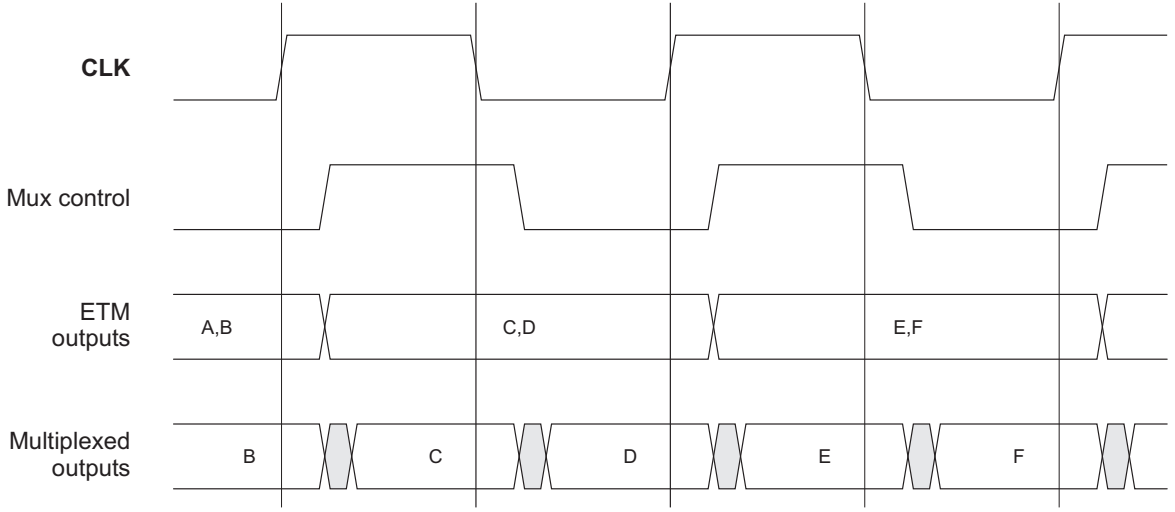

Figure 3-17 Multiplexed signal timing

Sufficient delays must be present in the switching of the trace data pins with respect to both edges of **TRACECLK**. You can achieve this by ensuring that **TRACECLK** is taken from the root of the ASIC clock tree. It is recommended that you carry out careful analysis to verify the timing on the pins of the ASIC.

3.7.3 Demultiplexed trace port signals

This scheme is recommended in systems that reuse general ASIC pins for trace, or for high-speed systems where the switching frequency of the off-chip trace signals is unacceptable. Figure 3-18 on page 3-31 shows logic to implement a demultiplexed trace port.

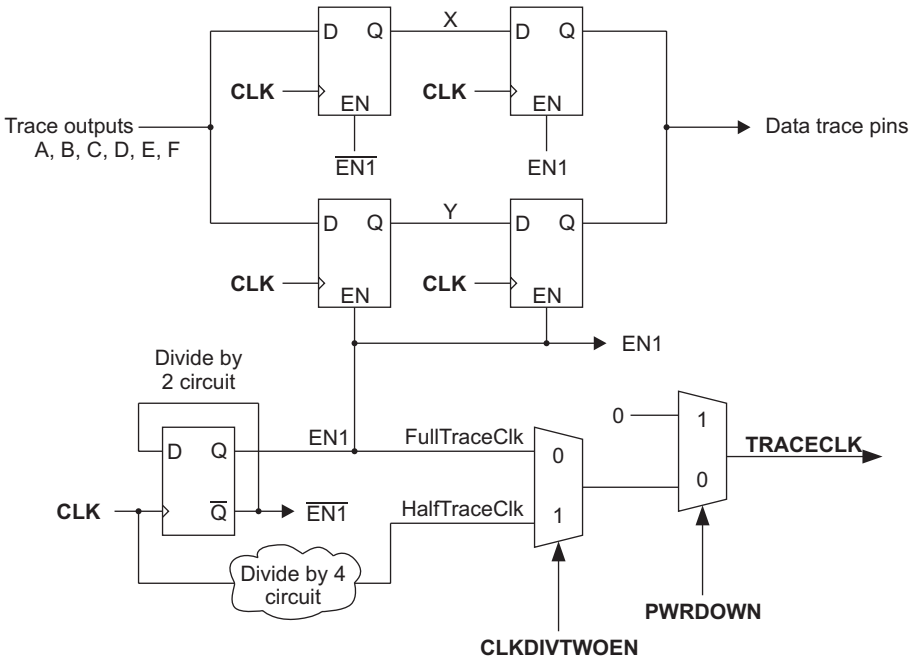

Figure 3-18 Demultiplexing trace data signals

Figure 3-19 on page 3-32 shows the timings for demultiplexed trace data signals.

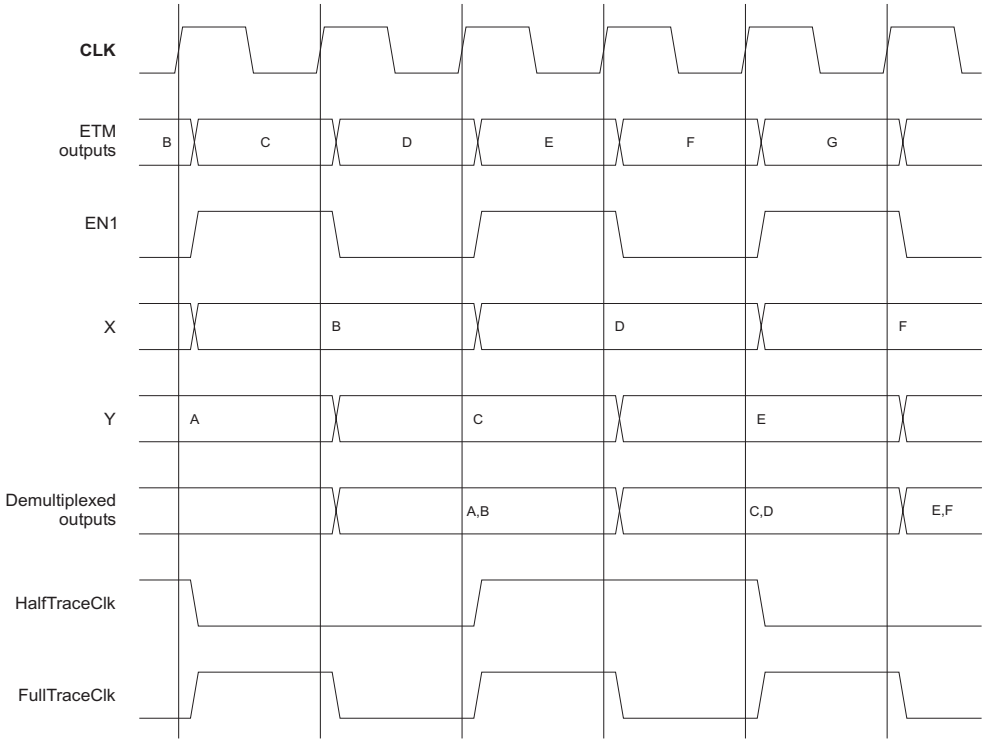

Figure 3-19 Demultiplexed signal timing

Half-rate clocking can be supported with demultiplexed trace data signals. However, you must take care to produce a clean trace clock. The following Verilog is an example of how a half-rate clock (in effect one quarter of the rate of the ETM clock) might be produced:

```
always @(posedge CLK or negedge nRESET)  if (nRESET == 1'b0)    begin      State
<= 2'b00;      HalfTraceClk <= 1'b0;    end  else    case (State[1:0])
2'b00  : begin State <= 2'b01; HalfTraceClk <= 1'b0; end      2'b01  : begin
State <= 2'b10; HalfTraceClk <= 1'b0; end      2'b10  : begin State <= 2'b11;
HalfTraceClk <= 1'b1; end      2'b11  : begin State <= 2'b00; HalfTraceClk <=
1'b1; end      default : begin State <= 2'bXX; HalfTraceClk <= 1'bX; end
```

This scheme ensures that the delay from the system clock to **TRACECLK** is minimized and ensures that there are no registers clocked from the data output of other registers. This helps static timing analysis.

In demultiplexed mode, the TPA must examine the two cycles of trace data in parallel to determine whether a trigger has occurred. It must also check for the trace disabled pipeline status in both cycles of data.

3.7.4 Operation with asynchronous TCK

You can use the ETM7 in systems that have a fully asynchronous **TCK** and **CLK**. All synchronization issues are taken care of in the ETM7. All groups of signals are synchronous to the relevant clock:

- ARM7 interface **CLK**
- Trace port **CLK**
- JTAG port **TCK**.

Slow system clock speeds

The ETM7 contains synchronizing D-types to synchronize between the **TCK** timing domain and the **CLK** timing domain. When the system clock speed is very slow, this synchronization time causes a delay of several cycles before you can disable tracing.

Chapter 4
Memory Map Decode Interface

This chapter discusses the Memory Map Decode Interface used in the ETM7. It contains the following sections:

4.1 About the memory map decode interface

When you implement an ASIC or ASSP, there are usually a number of memory mapped peripherals and areas of external and internal RAM, ROM, and flash, for example.

The memory map decode outputs allow simple, low-cost decoding of this address map using ASIC-specific logic. This logic drives the **MMDIN** inputs to the ETM, making them available to you as ETM resources, in a similar way to the address comparator and address range comparator resources.

The structure of the (MMD) logic is shown in Figure 4-1

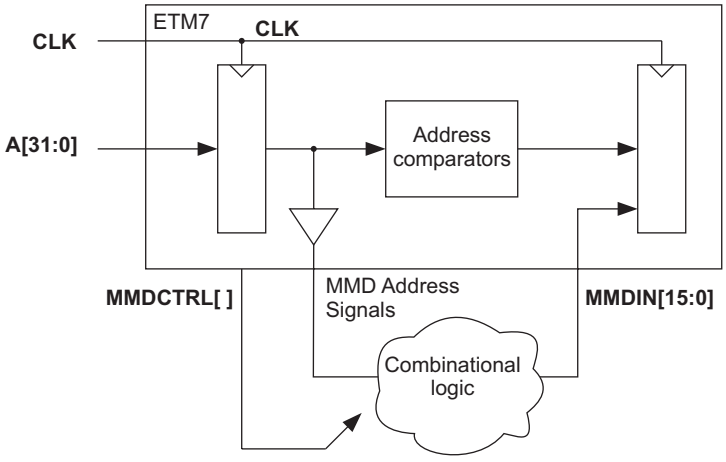

Figure 4-1 Memory map decode logic structure

In Figure 4-1, MMD Address signals are:

- **MMDA[31:0]**
- **MMDnOPC**
- **MMDnMREQ**
- **MMDnRW**.

All unused **MMDIN** inputs must be tied to logic zero.The **MMDCTRL** bus comes from an ETM register, programmed by the Trace debug tools. These allow you to specify the value to be programmed into this 8-bit register.

4.1.1 Signal descriptions

The MMD signals are as follows:

MMDnMREQ Pipelined version of **nMREQ**

MMDnRW Pipelined version of **nRW**

MMDA[31:0] Pipelined version of **A[31:0]**

MMDnOPC Pipelined version of **nOPC**

MMDMAS[1:0] Pipelined version of **MAS[1:0]**

MMDCTRL[7:0] Memory map decode control signals.

4.2 Memory map decode example

Figure 4-2 shows a memory map decode example.

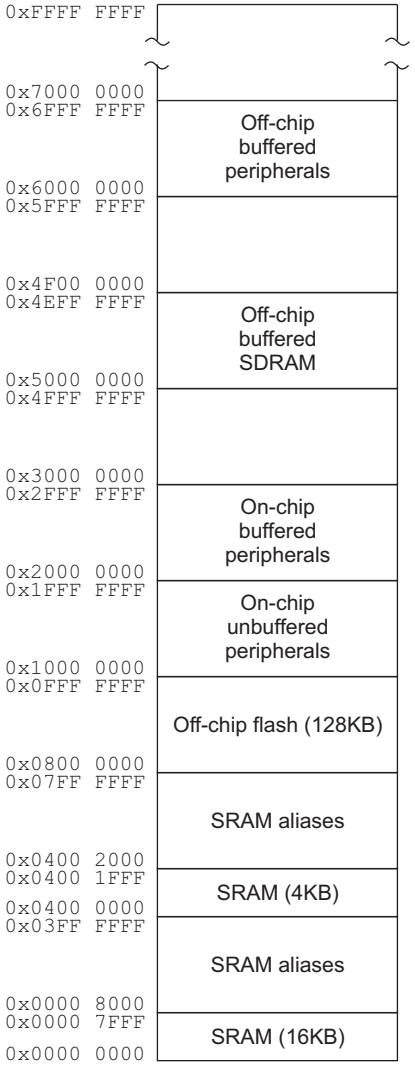

Figure 4-2 Memory map decode example

The combinational logic used to decode memory map addresses in Figure 4-2 on page 4-4 is shown in pseudo-HDL format in Table 4-1.

Table 4-1 Memory map decode example pseudo-HDL

Logic expression	Comment
MMDIN[0] = (MMDA[31:24] = 0x00) AND NOT(MMDnMREQ)	SRAM
MMDIN[1] = (MMDA[31:24] = 0x04) AND NOT(MMDnMREQ)	SRAM
MMDIN[2] = (MMDA[31:24] = 0x08) AND NOT(MMDnMREQ)	Flash memory
MMDIN[4] = (MMDA[31:20] = 0x100) AND NOT(MMDnMREQ)	Three unbuffered peripherals
MMDIN[5] = (MMDA[31:20] = 0x101) AND NOT(MMDnMREQ)	
MMDIN[6] = (MMDA[31:20] = 0x102) AND NOT(MMDnMREQ)	
MMDIN[7] = (MMDA[31:20] = 0x200) AND NOT(MMDnMREQ)	Two buffered peripherals
MMDIN[8] = (MMDA[31:20] = 0x201) AND NOT(MMDnMREQ)	
MMDIN[9] = (MMDA[31:28] = 0x4) AND NOT(MMDnMREQ)	Off-chip SDRAM
MMDIN[10] = (MMDA[31:28] = 0x600) AND NOT(MMDnMREQ)	Off-chip buffered peripherals
MMDIN[11] = (MMDA[31:28] = 0x601) AND NOT(MMDnMREQ)	
MMDIN[12] = (MMDA[31:28] = 0x602) AND NOT(MMDnMREQ)	
MMDIN[13] = (MMDA[31:28] = 0x603) AND NOT(MMDnMREQ)	
MMDIN[14] = (MMDA[31:28] = 0x604) AND NOT(MMDnMREQ)	

Chapter 5
ASIC Trace Validation

This chapter describes how to validate that an ETM has been correctly integrated into an ASIC. It contains the following sections:

5.1 About ASIC trace validation

You can use the tests described in this chapter to generate chip-level vectors for production test. ARM recommends that you carry out the following sequence of steps to ensure that your ASIC is fully validated:

1. Run the example test on the example test bench. This illustrates the components and processes required to validate a real system (see on page 5-6, on page 5-7, and on page 5-8).
2. Modify your ASIC test bench to add your system components (see on page 5-9).
3. Modify the example test program for your ASIC environment (see on page 5-10).
4. Run your modified test program to validate the trace.

There are two parts to the process of validating the trace for an ETM integrated into an ASIC design:

- validating that the ETM is correctly wired to the ARM processor it is tracking
- validating the trace port wiring to the pins of the ASIC.

You can achieve both of these by carrying out the following steps:

1. Enable the ETM and run a program on the ARM processor.
2. Capture the resulting trace on the trace port pins of the ASIC.
3. Decompress the trace (see on page 5-17).
4. Compare the decompressed output against the ARM instructions that are executed (see on page 5-17).

A valid trace output indicates that the ETM and the trace port are correctly wired.

Note

ARM has found that some versions of Perl have bugs that result in incorrect operation of the supplied scripts. ARM uses Perl version 5.005_004 and recommends that if any problems arise in the use of these scripts, then you must investigate possible Perl version incompatibilities as a first step.

5.2 Release package structure

The ASIC trace validation package comprises:

Validation package

This is assigned the ARM part numbers TM020-SW-01001 and TM030-SW-01001. The contents of the packages are identical. Each package contains:

- ARM assembler source of the example test program, including an example Verilog test bench to run it on
- decompression and comparison scripts.

Boundary Scan Trickbox (BST)

This is a ModelGen (DSM) that drives the JTAG interface of the ASIC. It is controlled by commands written in a file called `JTAGbsi`. The format of these commands is hard to write manually, and therefore a script called `parse_bsi.pl` is also provided in the BST release and in the `scripts` directory. This script allows higher level commands, which can include ARM assembler instructions, to be written and turned into the low-level commands understood by the BST.

ARM7TDMI model

This is a ModelGen DSM that is used in the example Verilog test bench. It contains example components of a typical ARM7TDMI-based ASIC design.

Note

There is no ETM7-specific validation package. All tools and examples are common to ETM7 and ETM9.

Details of the BST and scripts are provided in the simulator-specific BST release deliverable, including how the BST is integrated into the simulator being used.

Note

The `parse_bsi.pl` script requires that the ARM toolkit is available. This is used to assemble the ARM instructions that are scanned into the ARM processor when it is in debug state.

5.2.1 Directory structure

The ASIC trace validation package has the directory structure shown in Figure 5-1.

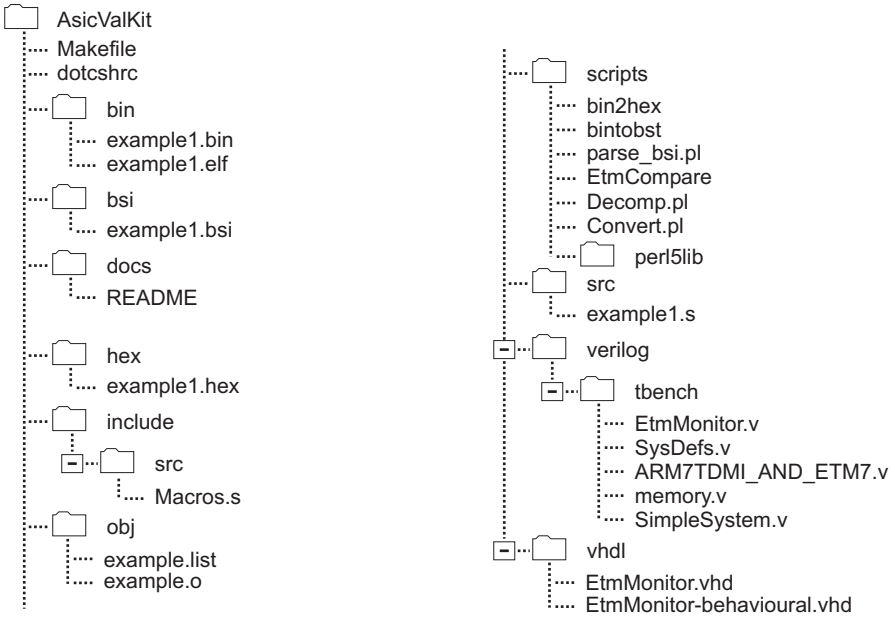

Figure 5-1 Directory structure

A make file is provided in the `AsicValKit` directory that allows you to compile the example test program.

A file called `dotcshrc` is provided as an example of how to set up the required environment variables, and adjust the user path, for example.

Note

You must edit this file to refer to your own environment and simulator. You must also edit the top of the Perl scripts to set the correct path to your local installation of Perl.

Figure 5-2 on page 5-5 shows the equivalence of package components to the hardware components of a real system.

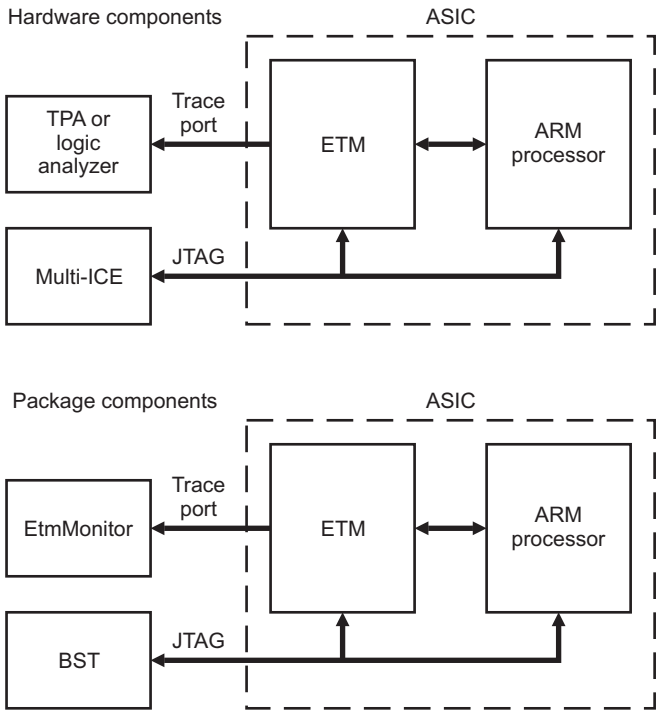

Figure 5-2 Equivalence of package components to hardware

5.3 Using the example test bench

The verilog file, SimpleSystem.v, is in the verilog/tbench directory. This instances an ARM7TDMI, ETM7, and the BST, and provides a simple platform for you to try out the example test program.

You are advised to run the test bench as supplied to gain an insight into the operation of a typical system. You can then modify the test bench to match your own system configuration (see on page 5-9), before using the modified test bench for testing.

To compile and run the test bench for the ModelSim verilog simulator, use the following command in the AsicValKit directory:

```
ln -s hex/example1.hex rom.hex
ln -s bsi/example1.bsi JTAGbsi
```

This links in the files that the test bench looks for. They are provided as precompiled object files. Before you compile the verilog, you must edit the paths to the DSMs in the dotcshrc file and the verilog/tbench/verilog.vc file. For example, in dotcshrc:

```
setenv DIR_ARM7TDMIr3 .../arm7tdmi_vsystemv_SunOS5_3A.00/ARM7TDMIr3
```

and, in verilog.vc:

```
-v .../arm7tdmi_vsystemv_SunOS5_3A.00/ARM7TDMIr3/ARM7TDMIr3.v
```

The next step is to compile and run the test bench verilog. For example, if you are using ModelSim:

```
vlib work
vlog -f verilog.vc verilog/tbench/SimpleSystem.v
vsim -i SimpleSystem
```

For Verilog-XL, use the following:

```
verilog -f verilog.vc verilog/tbench/SimpleSystem.v
```


5.4 Using the BST

The commands that you scan into the JTAG port must be included in the source code of the test. The TOBST assembler macro indicates the start of the BST instructions in the test.

A script called bintobst is provided. This script first extracts the BST instructions from the assembled binary image of the test. It then uses a program called parse_bsi.pl to process the high level commands into a form that the BST can accept. This process is explained fully in the following files (supplied with the BST DSM):

- DOCS/BoundaryScanCommands.txt
- DOCS/armBST.txt.

If the JTAGbsi file read by the BST is incorrect it can cause the following types of problems:

- the ARM does not enter debug state
- the ETM is not programmed correctly
- the ARM does not exit debug state, or restarts at the wrong address.

Many of these problems can be caused by the BST not knowing the type of ARM processor (ARM7 family or ARM9 family) that it is talking to. This is controlled by the PROC instruction. The example test shows the two forms of PROC instruction necessary to drive an ARM7 or ARM9 processor.

Another problem that can arise is that the parse_bsi.pl script can have difficulty assembling the ARM instructions to be executed in Debug state. The ARMINST command specifies an instruction to execute. The armasm program (not provided in the ASIC validation release) is called to assemble the ARM instruction. The resulting instruction bit pattern is then turned into a BST scan command.This instruction is scanned into the ARM and executed at debug speed. For details of exactly how this works, refer to the appropriate ARM core Technical Reference Manual.Other problems can arise if armasm is not available, or if the version available is too old. You must use the armasm program that is provided with the unix version of the (ADS), or SDT 2.50 (SDT 2.11a is not supported).

5.5 The test program

A simple test program is supplied in the src directory, in a form that can easily be assembled and linked by the ARM tools. The ARM Developer Suite is required to assemble and link the test program, and to create the command file read by the BST.

The example test program attempts to exercise the majority of the signals between the ARM processor, the ETM, and the trace port signals.

5.5.1 Building the example test program

A makefile is provided to assemble and link the test program (src/example1.s).

The following final files are generated by the build process:

hex/example1.hex
: A hex image of the test.

bsi/example1.bsi
: The JTAG instructions for the BST.

Note

The BST reads a file called JTAGbsi, in the directory that the simulation is run in. It is recommended that you link this to the bsi/example1.bsi file.

5.5.2 Test program operation

The test program is written to be loaded at 0x0. If your ASIC design does not allow you to load the test program at address 0x0, you must modify the source code so that the code is loaded at another address and then a jump is made to the start address from the reset vector at 0x0 (or 0xFFFF0000, if in an ARM720 system with **HIVECS** HIGH).

When the program starts running, it initially executes an infinite loop. This can be at the ARM reset vector, or at the address that is initially jumped to by the instruction at the reset vector. The purpose of this infinite loop is to allow the BST to program the ETM registers to enable tracing to start. The BST carries out the following operations:

1. Stops the ARM processor by setting the DBGRQ bit in the EmbeddedICE debug control register.
2. Programs and enables the ETM.
3. Restarts the ARM processor at the beginning of the main body of the test.

The test program then runs to completion.

5.6 Modifying your ASIC test bench

To test the ETM in your system you must add the BST DSM and the `EtmMonitor` HDL into your ASIC test bench. You must add these components at the highest possible level to ensure that all of the trace port and JTAG wiring is fully tested. For example you must not add these components inside your ASIC.The BST and the `EtmMonitor` script provide the same functionality as the run control and TPA hardware that you will use to develop your software.The example test bench shows how the two components should be integrated into a Verilog test bench. Also, the `README` file provided with the BST tells you how to integrate the BST DSM into your chosen simulator.A VHDL version of the `EtmMonitor` script is provided in the release.

`EtmMonitor.vhd`

Contains the entity model.

`EtmMonitor-behavioural.vhd`

Contains the architectural model.

The test program supports character output and automatic termination of the simulation using a memory-mapped . The example test bench implements this at address `0x03000000`. Writes to this address are sent to the simulator, and a write of `0x04` terminates the simulation once the ETM FIFO has drained.

5.7 Modifying the test program

Before you use the test program, you must:

- configure the Makefile
- adapt the test program to your environment and test requirements.

5.7.1 Configuring the Makefile

Before using the test program, you must configure the provided Makefile to reflect your choice of ARM processor. For example, if you are using an ARM7TDMI, you must make sure that the following line appears at the start of the Makefile:

```
ASDEF1 = -PD 'PROC SETS "ARM7TDMI"' -PD 'ARCH SETS "ARMv4T"'
```

You can do this by uncommenting the appropriate `ASDEF1` definition from the four that appear as comments at the start of the Makefile.

You must also modify the Makefile if your simulation environment requires a different form of object file.

If you are using SDT 2.50, then you must adjust the rule in the Makefile for `bin/example1.bin`. You can do this by uncommenting the appropriate rule from the two that appear towards the end of the Makefile. You must also comment out the default ADS rule.

5.7.2 Adapting the test program

The program provided allows basic testing of most of the major signals between the ARM processor and the ETM. It is not possible for you to test all signals in a general way however, because many ASICs implement partial memory maps, that often contain memory that is read or write sensitive. For this reason ARM recommends that the test program is extended and adapted to suit your specific ASIC memory map.

The areas of the test program that you can extend are:

- Memory address range. It is important to test that all the address bits are correct, requiring that loads/stores and instructions can be accessed from all of the memory regions.
- Specific testing of static configuration signals, such as **BIGEND**.
- Miscellaneous ETM signals, such as **PWRDOWN** and **ETMEN**.

The remainder of this section identifies signals, or groups of signals, that you might need to pay special attention to during testing. Each signal or group is considered individually.

5.7.3 ARM instruction and data interfaces

The ETM tracks the main instruction and data interfaces. At appropriate points in the sample test, the program contains comments that describe how the test must be extended to ensure that all of the address interface between the ARM processor and the ETM are correctly connected.

5.7.4 BIGEND

If your system uses **BIGEND**, you must test that loads and stores, and thumb code executed in big-endian mode, are traced correctly. You can do this by running the test twice:

- once in little-endian mode
- once in big-endian mode.

You can use the code shown in Example 5-1 to switch the BIGEND configuration bit in the system control coprocessor (CP15):

Example 5-1 Switching the BIGEND configuration bit

```
[   {ENDIAN} = "big"
; If the -bigend armasm flag has been used, set the BIGEND bit
; in CP15.
MRC p15, 0, r0, c1, c0, 0
ORR r0, r0, #(1<<7)
MCR p15, 0, r0, c1, c0, 0
]
```

The code in Example 5-1 uses the `ENDIAN` assembler variable that is automatically defined when the arm assembler is called with the `-bigend` command line flag. If your system drives the **BIGEND** input to an ARM7TDMI or ARM9TDMI core using a memory mapped register, you must modify the code appropriately.

Note

`EtmCompare` has a `-BigEnd` flag. This is required because `EtmCompare` can deal with tracing that is controlled by comparisons on the load and store data. The internal ETM data comparisons are affected by:

- the way the memory system returns the data to the ARM
- how the data masks in the ETM are programmed.

However, the code in Example 5-1 does not use this functionality, and it is unlikely to be necessary to add it to the test.

The example Makefile has an ENDIAN variable to correctly set the command line options on the arm assembler and the bin2hex scripts.

5.7.5 Aborts

The abort signal(s) might also need specific additions, depending on whether there are memory regions that can abort. Even if aborts are not expected, it is strongly recommended that one example of an Instruction Abort and one example of a Data Abort are tested.

Note

The memory model in the example test bench generates a warning when an uninitialized address is accessed. The warning occurs during the abort test, for example:

```
# Read from uninitialised memory location at 0x01000008. (2)
```

Data Aborts

You must place the following instruction at the Data Abort vector:

```
; Return to the instruction AFTER the one
; that caused the abort.
SUBS PC,r14,#4
```

You must also add loads and stores, that access the aborting memory regions, to the test. When the Data Abort occurs, the instruction at the vector automatically returns to the instruction following the one that caused the exception.

The sample test includes the above code, and accesses to a data aborting address in the test bench.

Instruction (Prefetch) Aborts

You must place the following instruction at the Prefetch Abort vector:

```
    ; Return to the instruction at the address in R0.
    MOVS PC,R0
```

You must also add jumps, to and from the aborting memory regions, to the test body. You can do this as follows:

```
    LDR R1,=AbortingAddress
    ADRL R0,ReturnAddress
    MOV PC,R1
ReturnAddress
```


The sample test includes the above code, and has a branch to an aborting memory region in the test bench.

5.7.6 Interrupts

No attempt is made to cause an interrupt in the test program. This is because the ETM does not have any connection to the interrupt wires.

The `EtmCompare` program can compare sequences containing interrupts, if you choose to test them.

5.7.7 ETM outputs

You must test any ETM output signals that you use, such as:

- **FIFOFULL**
- **PORTSIZE[2:0]**
- **PWRDOWN**
- **ETMEN**
- **CLKDIVTWOEN**
- **EXTOUT[3:0]**.

5.7.8 FIFOFULL

You must test this in two ways:

- You must run code that causes frequent overflows to occur. The best way to do this is to use a 4-bit packet size with address and data tracing enabled. You can do this by setting the assembler directives at the top of the sample test as follows:

  ```
  PORT_SIZE SETA 4TRACE_DATA SETA 1TRACE_DATA_ADDRESSES SETA 1
  ```

  The assembler macros provided with the test program, support the programming of the **FIFOFULL** level register, and the setting of the **FIFOFULL** enable bit in the ETM control register. When you assemble the test, you must set the `ETM_STALL_COUNT` assembler variable to the value to be programmed into the level register. For example, `ETM_STALL_COUNT SETA 10`. The assembler macros check if this has been set, and write the value required to the register. The test also programs the **FIFOFULL** region register and sets the **FIFOFULL** enable bit in the ETM control register, if required.

- Connect a random signal generator (for example an LFSR) to drive the **FIFOFULL** input to the ARM subsystem. The random signal generator is allowed to run for all of the tests in the ASIC validation process. The random setting of the **FIFOFULL** signal that results is used to show that, no matter when

the stall input is asserted, the system continues to operate successfully. The simple example ETM test supplied is not adequate to completely test the stalling function for the ASIC under test.

5.7.9 PORTSIZE

If you use the **PORTSIZE** outputs to control on-chip logic, such as is shown in Figure 3-13 on page 3-23, this logic must be tested. To do this you must run the trace test program up to three times, once with each of the supported port widths programmed into the ETM control register (bits 6:4), to test each of the configurations. If the trace can be correctly decompressed, the ASIC logic is correct.

To change the **PORTSIZE** value that is programmed into the ETM, use the assembler define `PORT_SIZE`, as shown at the top of the sample test.

5.7.10 PORTMODE

You must run the test for each of the different port modes that your system supports. For a description of operating modes see on page 3-28.

5.7.11 PWRDOWN

If you use the **PWRDOWN** ETM output to turn off the ETM and/or to gate the **TRACECLK** ASIC pin, you must visually check that this is occurring. You can test the logic that enables the ETM and/or that allows the gating of the **TRACECLK** pin by running the trace test program, and then confirming system operation in response to the **PWRDOWN** signal.

5.7.12 ETMEN

If the **ETMEN** ETM output is used to control on-chip logic, such as for multiplexing the ASIC outputs with the trace port signals, this logic must be tested. To do this you must test the ASIC outputs as part of the ASIC validation. In addition, when you run the trace test program, the ASIC outputs must be multiplexed with the trace signals onto these pins using all the supported port widths. If the trace can be correctly decompressed, the ASIC logic is correct.

The example test program sets the **ETMEN** output HIGH at the start of the test, allowing the testing of any logic that depends on the **ETMEN** signal.

5.7.13 CLKDIVTWOEN

The recommended configuration allows you to use this signal to divide the trace clock, see on page 3-25. To control this output, you can use the SET_CLKDIVTWOEN and UNSET_CLKDIVTWOEN macros in the BST sequences. You must do this before starting tracing.

The EtmMonitor block has a **CLKDIVTWOEN** input which, when set, causes it to sample the trace signals on both edges of the trace clock. You can tie this HIGH when testing half-rate clocking or, if using Verilog, use a direct reference to the ETM output to directly observe the state of the CLKDIVTWOEN signal.

5.7.14 EXTOUT

If these ETM outputs are used in the ASIC, you must test both the ETM output, and the logic that it drives. The external outputs are controlled by events, and you can use the following BST commands in the BST section of the test to set the external outputs LOW:

```
BST "ETMINST WRITE 0x68 0x0000406F"BST "ETMINST WRITE 0x69 0x0000406F"BST "ETMINST
WRITE 0x6A 0x0000406F"BST "ETMINST WRITE 0x6B 0x0000406F"
```

To set an external output HIGH, write 0x0000006F to the event register. For example:

```
; Set external output 2 highBST "ETMINST WRITE 0x6A 0x0000006F"
```

The external output event registers, like most other ETM registers, are not reset. The external outputs are held LOW when the ETM programming bit is HIGH. This occurs automatically at reset. It is important therefore, to program the external output event registers before clearing the programming bit.

5.7.15 EXTIN

This provides a number of inputs to the ETM from the ASIC or even off-chip inputs. You must test all the external inputs that you use, to verify the logic and wiring that drives the inputs.The simplest way to test the external inputs is to program the ETM to trigger when the external inputs are asserted. For example, to cause a trigger when external input 1 goes HIGH:

```
BST  "ETMINST WRITE 0x02 0x00000060"
```

It is also possible to route the external inputs straight through to the external outputs, for example:

```
BST  "ETMINST WRITE 0x68 0x00000060"BST  "ETMINST WRITE 0x69 0x00000061"BST
"ETMINST WRITE 0x6A 0x00000062"BST "ETMINST WRITE 0x6B 0x00000063"
```

5.7.16 Trace port signals

The trace port signals are:

- **TRACESYNC**
- **PIPESTAT**
- **TRACEPKT**
- **TRACECLK**.

You can test these adequately by carrying out the basic tracing tests. There are no specific additional requirements needed to guarantee full testing of them. However, you are advised to ensure that your test program covers:

- the three possible port widths
- instruction-only tracing
- full data tracing.

5.7.17 Trace filter testing

You do not have to test the trace filtering functionality of the ETM, because this is tested as part of the ETM validation process.

5.8 Trace script usage

Three scripts are provided with the release, in the `scripts` directory:

`Decomp.pl` The trace decompression script.

`Convert.pl` Converts EIS output to an intermediate format.

`EtmCompare` Allows a decompressed trace to be compared with an EIS.

Note

`Decomp.pl` and `Convert.pl` do not need to be run manually because they are run automatically by `EtmCompare`. See on page 5-19 for instructions on running `EtmCompare`.

You must edit the first line of the Perl scripts to point to your local Perl installation.

5.8.1 Decompressor

This script takes the output from the `EtmMonitor.v` Verilog block, plus an image of the code being executed, and produces a decompressed trace. It uses a number of Perl modules (`Image4b.pm`, `ImageAxf.pm`, `CF.pm`, `Coproc.pm`, `Output.pm`, `Trigger.pm`). The command for invoking the decompression script is:

```
Decomp.pl [options] compressedFile imageFile
```

The command-line options can appear in any order. The options are:

`-arch <Name>`

When ARMV5T is specified, allows decompression of ARMv5T instructions.

`-base <HexValue>`

Allows the base of a binary image to be specified, if it is not 0x0.

`-a` Indicates that data address tracing is enabled (can be used with `-d`).

`-d` Indicates that data data tracing is enabled (can be used with `-a`).

`-PortSize <size>`

Specifies the port width setting of the trace. `<size>` must be either 4, 8, or 16.

`-c <FileName>`

Specifies an optional coprocessor map file.

`-m` Specifies whether coprocessor CPRT tracing is enabled.

The hex format of the image file is the same as the format used to be read directly into verilog memories (using `$readmemh`). Example 5-2 shows a typical decompressor script command line.

Example 5-2 Typical decompressor script usage

```
Decomp.pl -PortSize 8 -arch ARMV4T -a -d -m log.etm rom.hex
```

The image file format supported is hex (see the `hex/example1_hex` fileprovided). These files can be generated using the `fromelf` program provided with the ARM tools, followed by the `bin2hex` Perl script provided. For example:

```
fromelf -bin -output test.bin test.elf
bin2hex test.bin test.hex
```

5.8.2 EIS converter

A script called `Convert.pl` is providedto take an (EIS) output file from an ARM model (ARM7TDMI, ARM9TDMI, or ARM9E-S) and convert it into a format similar to that of the decompressed trace. It uses two Perl modules (`EISBase.pm`, plus one of either `ARM7MG.pm`, `ARM9MG.pm`, or `ARM9vhd.pm`). The command for invoking the EIS converter script is:

```
Convert.pl [options]
```

The options can appear in any order. The options are:

`-eis <FileName>`

The name of the EIS file (normally `log.eis`).

`-unc <FileName>`

The name of the file produced (normally `log.eis_unc`).

`-eistype <EisFormat>`

The EIS format must be `ARM7TDMI`, `ARM9TDMI`, or `ARM9x`. `ARM9x` is the default and must be used for all ARM9 cores other than ARM9TDMI and ARM920T.

Example 5-3 on page 5-19 shows a typical EIS converter command line.

Example 5-3 Typical EIS converter script usage

```
Convert.pl -eis log.eis -unc log.eis_unc -eistype ARM9x
```

5.8.3 Trace comparison script

The `EtmCompare` Perl script compares the decompressed trace with the converted EIS output, using the ETM programming information in the `JTAGbsi` and `log.dsm_bst` files, to verify that the trace is correct. It uses no modules. The command for invoking the trace comparison script is:

```
EtmCompare [options]
```

The `EtmCompare` script can take a large number of options. The options used in the validation process are:

`-TRACE_DATA_ADDRESSES 1` Specifies data address tracing is enabled.

`-TRACE_DATA 1`

Specifies coprocessor register transfer tracing is enabled.

`-TRACE_CPRT 1`

Specifies coprocessor register transfer tracing is enabled.

`-PORT_SIZE <size>`

Specifies the selected port width. This option is always required. `<size>` must be either 4, 8, or 16.

`-trace <trace_file>`

The name of the compressed trace file (default is `log.etm`).

`-image <image_file>`

The name of the memory image file (default is `rom.hex`).

`-bstlog <log_file>`

The name of the command log file produced by the BST (default is `log.dsm_bst`).

`-eis <FileName>`

The name of the EIS file (default is `log.eis`).

`-c <coproc_map>`

The name of the coprocessor map to pass to `Decomp.pl`. This file is necessary for the decompressor to successfully decompress traces involving coprocessor memory transfers (ARM instructions `LDC`, `LDCL`, `STC`, and `STCL`). It is not necessary if no external coprocessors are installed in the system (CP15 does not use these instructions).

`-base <address>`

Base address for image in `rom.hex` (default is 0). A common alternative is `0xFFFF0000`, if **HIVECS** is in use.

`-eistype <EisFormat>`

The EIS format, which must be `ARM7TDMI`, `ARM9TDMI`, or `ARM9x`. `ARM9x` is the default and must be used for all ARM9 cores other than ARM9TDMI.

`-arch <arch>`

The architecture version (`ARMV4T` or `ARMV5T`).

`-BigEnd`

This option is required if the test was run with the processor in big-endian mode and tracing is controlled using data comparators.

`-tbtube <address>`

Alternate tube address (default `0x03000000`). `EtmCompare` looks for a write of control-D (`0x04`) to this location to signify the end of the test and terminate the comparison. This option is not necessary if you turn tracing off before the end of the test.

The following options are also available, but are not normally used:

`-etmNoComp`

Only run `Decomp.pl` and `Convert.pl`. Do not compare their outputs.

`-noDecomp`

Do not run `Decomp.pl`.

`-noConvert`

Do not run `Convert.pl`.

The `EtmCompare` program invokes the `Decomp` and `Convert` programs automatically. This can be suppressed using `-noDecomp` and/or `-noConvert`. Example 5-4 on page 5-21 shows a typical `EtmCompare` command line.

Example 5-4 Typical EtmCompare script usage

```
EtmCompare -TRACE_DATA_ADDRESS 1 -TRACE_DATA 1 -TRACE_CPRT 1 \
    -PORT_SIZE 16 -trace log.etm -image rom.hex -eis log.eis \
    -eistype ARM7TDMI -arch ARMV4T
```

Example 5-5 shows a typical output, obtained after running the command in Example 5-4.

Example 5-5 Typical EtmCompare output

```
EtmCompare: Revision $Revision: 2.15 $EtmCompare: Running Decomp.pl -PortSize 16
-arch ARMV4T -a -d -m log.etm rom.hexEtmCompare: Decomp return code =
0EtmCompare: Running Convert.pl -eis log.eis -unc log.eis_unc -eistype
ARM7TDMIEtmCompare: Convert return code = 0EtmCompare: Comparing log.trc_unc and
log.eis_unc ...EtmCompare: 0 error(s) found
```

As the number of options to be passed to `EtmCompare` can be quite large, `EtmCompare` looks for a plain text file named `EtmCompare.cfg` on startup. You can specify one command-line option per line. They are added to the beginning of the options passed on the command line. If you specify an option twice, in `EtmCompare.cfg` and on the command line, the command-line value is used. This allows you to save your default command line in `EtmCompare.cfg` and to override any selected nonstandard options on the command line.

Problems with EtmCompare

When you access areas of uninitialized memory, unknown values can be traced by the ETM. These unknown values are propagated through the ETM, but are converted to known values in the `EtmMonitor` module. This is because the Perl libraries used by the scripts cannot understand `X` values.The side effect of this is that `EtmCompare` can generate errors when comparing the decompressed trace with the converted EIS file. You can avoid these errors by ensuring that memory addresses are initialized before they are read.

Chapter 6
Software Considerations for Trace

This chapter describes software issues relating to the ETM7. It contains the following sections:

6.1 Tracing dynamically loaded images

Support for dynamically-loaded images with the ETM7 Rev 0 is only possible with instrumented code, collusion of the OS, and additional support in the debugger.

ETM7 Rev 1 adds specific hardware support for tracing process IDs and overlay numbers. However, software support is still required.

6.1.1 Why dynamically-loaded code requires special hardware and software support

When a debugger is debugging a system it talks to the system largely in terms of addresses in the system's (possibly virtual) memory space. To be useful to its user it must translate between these addresses and locations in the images loaded on the system. This allows it to present a symbolic or source level view of the code running on the system to the user.

In a simple statically linked and loaded system the system runs a single image. The user tells the debugger the name of this image, and the image describes the mapping between target addresses as image locations. The debugger needs no further information to debug the image.

Many systems, including s (OS) such as WinCE, Linux, or Epoc32, load part or all of their software dynamically. This can have a number of 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 even know what images are candidates to be loaded until they are loaded.

To debug such a system the debugger must be able to ask the target system what images are loaded and where. At present ARMs debugger cannot ask such questions.

The problem is more difficult when using trace, because trace contains historical information. When analyzing a trace the debugger needs to know what images were loaded when the trace data was collected, rather than what images are loaded now. Additionally, even without a valid image, you can do some very basic, but sometimes useful interactive debugging, such as single-stepping instructions. The compression algorithm used for trace data means that the debugger cannot start to decode trace data unless the images that were loaded when it was generated are available.

In particular, you must remember that, for ALL embedded trace solutions to work, an image of the code being executed must be available to the trace decompression software of the debugger. This is because the instructions being executed are not broadcast, due to the data bandwidth that would be required, and so only the minimum of address

information is traced. 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 allows the target address of direct branches (`B` and `BL` instructions in the case of ARM) to be implied. Virtual memory and software paging, (effectively self-modifying code) for example, make this hard because the debugger probably does not know where the code ends up being executed from.

For a device with a full MMU, and an operating system that loads application and OS code into arbitrary locations in RAM, a way of telling the debugger what the code image for a particular trace sequence is required. The side-effect of this is that the closer to a physical address that can be supplied the better. For ARM720T the chosen solution is to broadcast the modified virtual address on the trace interface.

6.1.2 ETM7 Rev 1 hardware support

ETM7 Rev 1 provides the ability to trace a variable length process identifier field, whenever tracing is enabled, and as part of the periodic address broadcast. This enhancement allows simpler cooperation between the target operating system and the trace debug tools.

6.2 Simple overlay support

A system for supporting simple overlays is possible that does not require specific support within the trace debug tools. 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. That is, some of the unused address bits are *don't care* when determining the memory to be accessed.

For example, if you have 16KB of SRAM, bits 13:2 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 00, 01, 10, or 11.

If this 16KB block is to hold overlays, you can use bits 15:14 to indicate which of four possible overlays are loaded into this memory block. The value of bits 15:14 of the program counter will be traced by the ETM. In the trace tools a static image of the code being executed, with the four possible overlays statically linked (using scatter loading in the ARM tools) into the appropriate 16KB blocks of memory space, allows the trace decompression tools to successfully decompress the trace.

When the overlay manager loads and calls a new overlay, it copies the code into RAM and then jumps to the overlay. Bits 15:14 of the address, loaded into the PC, are set appropriately.

Chapter 7
Physical Trace Port Signal Guidelines

This chapter contains some signal guidelines that can ensure correct operation of the ETM and trace tools. It contains the following sections:

7.1 About trace port signal quality

Guaranteed operation of the TPA or Logic Analyzer depends on correct design of the ASIC and of the target PCB.

When integrating an ETM into an ASIC, the quality and timing of the trace port signals to the TPA are critical for reliable operation. Some of the issues to consider are:

- output pad selection
- PCB track lengths
- PCB track termination
- setup and hold times for the trace data signals with respect to .

The importance of these issues is directly proportional to the operating frequency. At frequencies greater than 100MHz, careful SPICE analysis of the system including the characteristics of the package and the chosen Trace Port Analyzer is recommended.

7.2 ASIC pad selection, placement and package type

The position and type of ASIC pad that you select depends on the following factors:

- the ability to minimize the noise and coupling between trace and other signals
- the ability to drive the external load.

The quality of the signal, as observed by the TPA, has the greatest effect on the reliability of the system. It is vital that transitions move cleanly through the threshold region of the input circuitry of the TPA, without glitches or ringing.

With certain types of package and pin placement (for example, pads on the corner or the edge of a package), the signal coupling between the trace data signals and the trace clock can be significant. If this problem is encountered during simulations, place or static I/O signals on both sides of the signal.

The quality of the package, and specifically the presence or absence of a ground plane in the package, can significantly affect the quality of the output signal. In general, ASIC pads are specified in terms of current drive and signal slew rate. For calculating the PCB signal quality you are likely to also have to determine:

- the signal rise and fall times
- the pad output impedance.

Matched impedance output pads give a significantly improved performance.

You must also consider the pad placement, to ensure that the PCB tracking to the trace port connector is possible. You are recommended to place the pads so that they are:

- on the outside of the package
- grouped together
- in the same order as the connector.

7.3 PCB design guidelines

Two implementations are possible:

The TPA or Logic Analyzer is the only load on the trace port signals. See *Dedicated trace port*.

A shared trace port

The trace pins are shared with other functions, and therefore there are stubs on the PCB tracks of the development board and an increased load on the output driver. See *Shared trace port* on page 7-6.

7.3.1 Dedicated trace port

This is the preferred implementation for connecting a TPA to a trace port. The TPA is the only load on the nodes connected to target ASIC pins, so the only factor affecting operation is signal integrity at the TPA connector.

If you know the characteristics of your PCB tracks, use the actual trace impedance and propagation delay. If you do not have access to this information, use the following guidelines for microstrip (track on outer layer over a ground plane) on FR4 PCB:

- Propagation speed is typically 160ps/inch (approximately 63ps/cm).
- The impedance of a 0.005-inch wide track as a microstrip is between 70Ω to 75Ω on a typical six-layer foil construction board. The impedance of a track reduces as the width of the track increases.

To design the target system effectively, you must know:

- the characteristic impedance and signal edge rates of the ETM output drivers
- the actual setup and hold provided by the ASIC ETM outputs with reference to the ETM **TRACECLK**.

If you do not know the characteristics of the signals from your ASIC, consult your ASIC vendor. It is difficult to provide any general rule because ETM output drivers and timings vary between ASIC vendors.

PCB track length

You must match all **TRACECLK**, **PIPESTAT[2:0]**, **TRACESYNC**, and **TRACEPKT[15:0]** track lengths between the ASIC and the trace port connector within 100ps. Overall differences in track lengths directly impact setup and hold requirements as follows:

- if the clock is delayed compared to the data, you must increase the setup specification by the additional clock delay
- if any data is delayed compared to the clock, you must add the delay to the setup requirement
- if data paths are such that data has both greater than and less than delays compared with the clock, you must add the difference to both the setup and hold specification.

Signal quality

The primary variable that characterizes signal quality is the rise time of a signal compared to its propagation time. It is this relationship that affects the track length, and this is where the minimum signal rise and fall time becomes important.

To ensure accurate data acquisition, you must minimize all reflections, overshoot, and undershoot. Aim to keep the one-way propagation time for all tracks at less than one third of the signal rise time.

As the fabrication process for your ASIC improves, your output driver is likely to improve and your rise and fall times are likely to decrease. If you cannot keep the propagation time for all tracks below one third of the signal rise time, some form of signal termination is required. This can be either of the following:

Series termination (Recommended method.) The series resistor must be placed as close as possible to the ASIC pin (half an inch or closer). The value of this series resistor plus the output impedance of the signal driver must closely match the impedance of the PCB track.

Parallel or matched AC termination

If you cannot use series termination, add parallel or matched AC termination on each signal track at the TPA target header. This requires significantly more power from the ASIC, and the AC termination must closely match the frequency and rise time of the terminated signal. In practice therefore, parallel termination is rarely possible.

If the total track length is one rise time propagation delay or greater in length, follow standard high-speed design practices to minimize cross talk between the clock and the data signals. (The total track length is the target PCB track length plus any PCB track on the TPA buffer board.)

Note

ASIC output pads with an output impedance that is matched to the PCB track might be available from your ASIC vendor. If these are used, the signal quality of the trace port signals is significantly improved.

7.3.2 Shared trace port

Some applications might not have enough pins available for trace, so you might have to multiplex trace signals with other functions. This has the effect of increasing the load on the trace signals, unless a specific trace-only development board is built.

When an ETM output pin is multiplexed with other functions, the addition of the TPA target header can add a stub to the PCB track on the target system. When this happens, the following additional constraints apply (The goal is to minimize the effect of the TPA target header on non TPA-based signal usage and maintain the integrity of the trace measurements.):

Signal does not require termination in normal operation or is parallel-terminated

This means that a full voltage swing signal travels down the track. Ensure that the propagation delay of the stub added for the TPA target header is 20 per cent or less of the overall rise and fall time of the signal.

Signal is series terminated for normal operation

This means that a one-half voltage swing signal begins each transition on the track and propagates down the track until it is terminated at the target node. This case is potentially very problematic. The one-half voltage swing signal can maintain the TPA input at its threshold voltage for longer than the required rise and fall time. To prevent this, you must move the TPA target header to within one fifth of the rise time of the target end of the track. If this is not possible, you must slow down the rise and fall times until this requirement can be met.

Sometimes, board layout constraints make it impossible to keep the stubs to the trace target header to less than 20 per cent of the rise and fall time. If length and speed requirements do not allow the rise and fall times to be increased to meet the design

requirements in this case, you can adjust the thresholds used by the TPA or LA, if supported. The target system must be able to provide:

- sufficient noise margin around an altered threshold
- sufficient setup and hold times because these will now be reduced.

Note

It is unlikely that any TPA supplier will guarantee support for this mode of operation.

7.4 EMI compliance

If you follow the guidelines in *Trace signal output timing* on page 3-25, the trace port pins, including the **TRACECLK**, are inactive. This means that the trace port does not affect your EMI compliance testing. The trace port pins are active only when the Trace Debug Tools are connected to the target.

It might be useful to carry out some testing with the trace port enabled, to determine the effect of the trace port switching on overall system noise.

7.5 Further references

Many TPA vendors provide models for download from the internet. These models enable you to use SPICE-like tools to analyze the signal integrity at the point that it is sampled by the TPA or Logic Analyzer.

Agilent The Agilent web site enables you to download data on their TPA and LA products. You can use the search engine on the web site to look for pages and documents that refer to ETM. For example, the document *Trace Port Analysis for ARM ETM* (Agilent document number E5903-97002) contains equivalent models for Agilent TPA and Logic Analyzer products.

Tektronix The Textronix web site has a number of documents relating to the use of their Logic Analyzers for acquiring trace. For example, the document *P6434 Mass Termination Probe* (Tektronix document number 070-9793-02) provides models for the equivalent load of the Logic Analyzer probe.

Other vendors

Details about TPA vendors are added to this document as they become known to ARM. You can also contact your chosen vendor directly for the latest information.

Appendix A
Signal Descriptions

Signal descriptions

A.1 Signal descriptions

Table A-1 ETM7 signals

Type	Signal name	Description
	A[31:0]	The processor address bus driven by the ARM.
Input	**ABORT**	The Abort signal driven into the ARM. The **ABORT** signal is used to tell the processor that the requested memory access is not allowed.
Input	**ARMTDO**	The **TDO** output signal from the ARM, or from an external scan chain.
Input	**BIGEND**	The signal driving the ARM **BIGEND/CFGBIGEND** input. When HIGH the processor treats bytes in memory as big-endian format. When LOW memory is treated as little-endian. This is a static configuration signal.
Input	**CLK**	This clock times all operations in the ETM7. All outputs change from the rising edge and all inputs are sampled on the rising edge. The clock can be stretched in either phase. Synchronous wait states can be added using the **CLKEN** signal.
Output	**CLKDIVTWOEN**	If HIGH, indicates that the ETM is in half-rate clocking mode.
Input	**CLKEN**	The ETM can be stalled by driving **CLKEN** LOW. This signal should be held HIGH at all other times. This must be the same as the signal that drives the ARM **nWAIT/CLKEN** input.
Input	**CPA**	The coprocessor absent signal driven into the ARM.
Input	**CPB**	The coprocessor busy signal driven into the ARM.
Input	**DBGACK**	The debug acknowledge signal driven by the ARM. When HIGH this signal indicates that the ARM is in debug state.
Output	**DBGRQ**	Debug request. A signal that can be used to stop the ARM processor.
Output	**ETMEN**	This output will be HIGH when the debugger has enabled the ETM.

Table A-1 ETM7 signals (continued)

Type	Signal name	Description
Input	**EXTIN[3:0]**	External inputs to the ETM. These inputs are available as resources within the ETM.
Output	**EXTOUT[3:0]**	External outputs from the ETM. Can be used to trigger hardware inside the ASIC, or external equipment such as a logic analyzer.
Output	**FIFOFULL**	When enabled, this indicates that there is less than a user-programmed number of bytes in the ETM FIFO.
Input	**INSTRVALID**	The **INSTRVALID** pipeline status signal driven by the ARM macrocell. The instruction valid signal indicates that the instruction in the Execute stage is valid, and has not been flushed.
Input	**MAS[1:0]**	The memory access size bus driven by the ARM. These encode the size of a data memory access in the following cycle.
Output	**MMDCTRL[7:0]**	A control bus, used to reconfigure the memory map decode logic.
Output	**MMDA[31:0]**	The ARM **A[31:0]** signal, pipelined for the memory map decode interface.
Output	**MMDMAS[1:0]**	The ARM **MAS[1:0]** signal, pipelined for the memory map decode interface.
Output	**MMDIN[15:0]**	Memory map decode inputs to the ETM trigger logic.
Output	**MMDnMREQ**	The ARM **nMREQ** signal, pipelined for the memory map decode interface.
Output	**MMDnOPC**	The ARM **nOPC** signal, pipelined for the memory map decode interface.
Output	**MMDnRW**	The ARM **nRW** signal, pipelined for the memory map decode interface.
Output	**MMDTBIT**	The ARM **TBIT** signal, pipelined for the memory map decode interface.
Input	**nCPI**	The not coprocessor instruction signal driven into the ARM.

Table A-1 ETM7 signals (continued)

Type	Signal name	Description
Input	**nEXEC**	The **nEXEC** pipeline status signal driven by the ARM macrocell. The instruction executed signal indicates that the instruction in the Execute stage of the pipeline follower of the ETM7 has been executed.
Input	**nMREQ**	The memory request signal driven by the ARM. If LOW at the end of a cycle then the processor requires a memory access in the following cycle.
Input	**nOPC**	The not coprocessor opcode signal driven into the ARM.
Input	**nRESET**	Active LOW ETM reset.
Input	**nRW**	The read write signal driven by the ARM. If LOW at the end of a cycle then any memory access in the following cycle is a read. If HIGH then it is a write.
Input	**nTRST**	Active LOW JTAG test reset.
Output	**PIPESTAT[2:0]**	Indicates the pipeline status of the ARM.
Output	**PORTMODE[1:0]**	This output bus allows the on-chip trace port output logic to be configured for normal, multiplexed, or demultiplexed modes of operation.
Output	**PORTSIZE[2:0]**	Indicates the currently selected port size in use on the **TRACEPKT[15:0]** bus.
Input	**PROCID[31:0]**	This bus provides a copy of the current process ID or overlay number from the ARM system control coprocessor or peripheral.
Input	**PROCIDWR**	This signal must be asserted whenever the **PROCID** bus changes. This causes the ETM to output the new process ID at the next available opportunity.
Output	**PWRDOWN**	When HIGH, indicates that the ETM may be powered down.
Input Input	**RANGEOUT0,** **RANGEOUT1**	The **RANGEOUT0,1/DBGRNG[1:0]** EmbeddedICE signals driven by the ARM. The EmbeddedICE **RANGEOUT** signals indicate that the corresponding watchpoint unit has matched the conditions currently present on the address, control and data buses. These signals are independent of the state of the enable control bit of the watchpoint unit.

Table A-1 ETM7 signals (continued)

Type	Signal name	Description
Input	**RDATA[31:0]**	The **DIN/RDATA** bus driven into the ARM.
Input	**SEQ**	The data sequential address signal driven by the ARM. If HIGH at the end of the cycle then any data memory access in the following cycle is sequential from the last memory access.
Input	**SYSOPT[7:0]**	Indicates to the debug tools the system options that have been implemented. Bits are tied HIGH or LOW as appropriate, as part of the integration process.
Input	**TBIT**	The **TBIT** signal driven by the ARM. When HIGH denotes that the ARM processor is in Thumb state. When LOW the processor is in ARM state. This signal is valid with the address.
Input	**TCK**	Test clock.
Input	**TCKEN**	Synchronous enable for test clock.
Input	**TDI**	Test data input.
Output	**TDO**	Test data output.
Input	**TMS**	Test mode select.
Output	**TRACEPKT[15:0]**	The trace packet port.
Output	**TRACESYNC**	A synchronization signal, indicating the start of a branch sequence on the trace packet port.
Input	**WDATA[31:0]**	The **DOUT/WDATA** bus driven by the ARM.

Appendix B
Differences between ETM7 versions

This appendix describes the changes that have been made to ETM7 for Rev 1. It contains the following sections:

- *Pin differences* on page B-2
- *Changes to the programmer's model in Rev 1* on page B-3.

B.1 Pin differences

The following pins have been added to ETM7 Rev 1 that do not occur in Rev 0:

- **PORTMODE[1:0]**
- **PROCID[31:0]**
- **PROCIDWR**
- **SYSOPT[7:0]**
- **INSTRVALID**.

B.2 Changes to the programmer's model in Rev 1

Rev 1 adds a number of features to the programmer's model. These are described in this section and fully documented in the fourth release of the *Embedded Trace Macrocell Specification*. The version of the ETM that has been implemented is identified by the protocol version field of the ETM configuration code register.

B.2.1 Process ID

You can use the **PROCID** bus and the **PROCIDWR** signal to allow tracing of different process IDs or overlay numbers.

B.2.2 System options

Rev 1 of the ETM7 allows you to input system configuration options using the **SYSOPT** bus. You can use these hard-wired inputs to change the operation of your trace debug tools according to the ETM7 system configuration.

B.2.3 Trace port mode

You can configure the ETM7 outputs for three modes of operation:

- normal mode
- multiplexed mode
- demultiplexed mode.

These allow you to optimize the use of trace output pins in your particular ASIC design.

B.2.4 Instruction tracing on/off

Using the EnOnOff bit of TraceEnable control register 1, in conjunction with preset instruction addresses, allows you to turn tracing on and off at preset instruction addresses. You can use this to inhibit or enable tracing for individual sections of code, such as for sub-functions, ensuring that you only trace the instructions that you need to trace.

Glossary

This glossary describes some of the terms used in this manual. Where terms can have several meanings, the meaning presented here is intended.

Application Specific Integrated Circuit An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.

Application Specific Standard Part/Product Another name for an Application Specific Integrated Circuit. The name implies that the device performs complete functions, and can be used as a building block in a range of products.

ASIC See *Application Specific Integrated Circuit*.

ASSP See *Application Specific Standard Part/Product*.

Clock gating Gating a clock signal for a macrocell with a control signal (such as **PWRDOWN**) and using the modified clock that results to control the operating state of the macrocell.

Debugger A debugging system which includes a program, used to detect, locate, and correct software faults, together with custom hardware that supports software debugging.

Embedded Trace Macrocell A hardware macrocell which, when connected to a processor core, outputs instruction and data trace information on a trace port.

ETM See *Embedded Trace Macrocell*.

Half-rate clocking 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.

Macrocell A complex logic block with a defined interface and behavior. A typical VLSI system will comprise several macrocells (such as an ARM9E-S, an ETM9, and a memory block) plus application-specific logic.

Joint Test Action Group 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*.

SCREG The currently selected scan chain number in an ARM TAP controller.

SPICE An accurate transistor-level simulation tool.

TAP See *Test access port*.

Test Access Port 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**.

Trace driver An RDI target that controls a piece of trace hardware. That is, the trigger macrocell, trace macrocell and trace capture tool.

Trace hardware A term for a device that contains an Embedded Trace Macrocell.

Trace port A port on a device, such as a processor or ASIC, which is used to output trace information.

TPA See *Trace Port Analyzer*.

Trace Port Analyzer 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.

Index

The items in this index are listed in alphabetical order. The references given are to page numbers.

A

B

C

D

E

F

I

M

N

O

P

R

S

T

V