ARM966E-S™

Revision: r2p1

Technical Reference Manual

ARM®

ARM DDI 0213E

ARM966E-S
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
July 2000	A	First release
January 2002	B	Second release
February 2002	C	Third release
February 2002	D	Fourth release
February 2004	E	Fifth release r2p1

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
ARM966E-S Technical Reference Manual

List of Tables
ARM966E-S Technical Reference Manual

List of Figures
ARM966E-S Technical Reference Manual

Preface

This preface introduces the *ARM966E-S Revision r2p1 Technical Reference Manual*. It contains the following sections:

- *About this manual* on page xii
- *Feedback* on page xvii.

About this manual

This document is the Technical Reference Manual for the ARM966E-S r2p1 processor.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described 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 manual is written for experienced hardware and software engineers implementing the ARM966E-S processor system designs.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM966E-S processor.

Chapter 2 *Programmer's Model*

Read this chapter for a description of the ARM966E-S coprocessor registers and programming details.

Chapter 3 *Memory Map*

Read this chapter for a description of the ARM966E-S fixed memory map implementation.

Chapter 4 *Tightly-Coupled Memory Interface*

Read this chapter for a description of the requirements and operation of the tightly-coupled memory.

Chapter 5 *Bus Interface Unit*

Read this chapter for a description of the operation of the *Bus Interface Unit* (BIU) and write buffer.

Chapter 6 *Coprocessor Interface*

Read this chapter for a description of the coprocessor interface and the operation of common coprocessor instructions.

Chapter 7 ***Debug Support***

Read this chapter for a description of the debug support for the ARM966E-S processor and the EmbeddedICE-RT logic.

Chapter 8 ***Embedded Trace Macrocell Interface***

Read this chapter for a description of the ETM interface, including details of how to enable the interface.

Chapter 9 ***Test Support***

Read this chapter for a description of the test methodology used for the ARM966E-S synthesized logic and tightly-coupled memory.

Appendix A ***Signal Descriptions***

Read this appendix for a description of the ARM966E-S signals.

Appendix B ***AC Parameters***

Read this appendix for a description of the timing parameters applicable to the ARM966E-S processor.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams* on page xiv
- *Signals* on page xiv
- *Numbering* on page xv.

Typographical

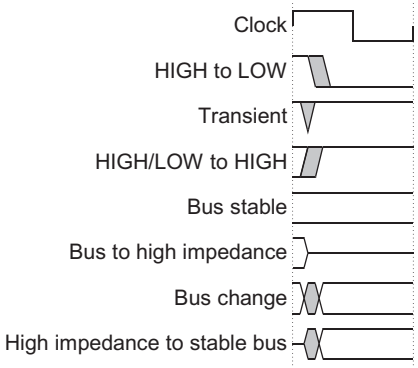
The typographical conventions are:

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.
`mono`space	Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

monospace italic	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
monospace bold	Denotes language keywords when used outside example code.
< and >	Angle brackets enclose replaceable terms for assembler syntax where they appear in code or code fragments. They 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.

Timing diagrams

The figure named *Key to timing diagram conventions* explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.



Key to timing diagram conventions

Signals

The signal conventions are:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals.
Prefix H	Denotes *Advanced High-performance Bus* (AHB) signals.
Prefix n	Denotes active-LOW signals except in the case of AHB or *Advanced Peripheral Bus* (APB) reset signals.

Prefix P Denotes APB signals.

Suffix n AHB **HRESETn** and APB **PRESETn** reset signals.

Numbering

The numbering convention is:

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

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

- 'h7B4 is an unsized hexadecimal value.
- 'o7654 is an unsized octal value.
- 8'd9 is an eight-bit wide decimal value of 9.
- 8'h3F is an eight-bit wide hexadecimal value of `0x3F`. This is equivalent to b00111111.
- 8'b1111 is an eight-bit wide binary value of b00001111.

Further reading

This section lists 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 ARM Limited Frequently Asked Questions list.

ARM publications

This manual contains information that is specific to the ARM966E-S processor. Refer to the following manuals for related information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM9E-S Technical Reference Manual* (ARM DDI 0240)
- *AMBA® Specification* (ARM IHI 0011)
- *ARM966E-S Implementation Guide* (ARM DII 0025)
- *ARM966E-S Test Chip Implementation Guide* (ARM DXI 0137)
- *AHB Example AMBA System Technical Reference Manual* (ARM DDI 0170)
- *ETM9 Technical Reference Manual* (ARM DDI 0157)
- *Application Note 99 - Core Type and Revision Identification* (ARM DAI 0099).

Other publications

This section lists relevant documents published by third parties:

- IEEE Std. 1149.1- 1990, *Standard Test Access Port and Boundary-Scan Architecture*.

Feedback

ARM Limited welcomes feedback both on the ARM966E-S processor and its documentation.

Feedback on the product

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 manual

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

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

ARM Limited also welcome general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the ARM966E-S r2p1 processor. It contains the following sections:

1.1 About the ARM966E-S processor

The ARM966E-S processor is a synthesizable macrocell with *Tightly-Coupled Memory* (TCM) interfaces. It is a member of the ARM9E™ family of high-performance, 32-bit *System-on-Chip* (SoC) processor solutions. The ARM966E-S processor is targeted at a wide range of embedded applications where high performance, low system cost, small die size, and low power are all important.

The ARM966E-S processor provides a high-performance processor subsystem that includes:

- An ARM9E-S RISC integer CPU core featuring:
  - — ARMv5TE 32-bit instruction set with improved ARM/Thumb code interworking and enhanced multiplier designed for improved DSP performance
  - — ARM debug architecture with additional support for real-time debug. This enables critical exception handlers to execute while debugging the system.
- Support for external TCM. A TCM interface is provided for each of the external instruction and data memory blocks. The TCM interfaces of the ARM966E-S processor enable high-speed operation without incurring the performance and power penalties of accessing the system bus, while having a lower area overhead than a cached memory system. The size of both the Instruction and Data TCM blocks are implementor-specific to enable tailoring of the hardware to the embedded application.
- A simple fixed memory map for the local TCM, ideal for real-time embedded control applications.
- An AMBA AHB bus interface.
- Support for external coprocessors enabling floating point or other application-specific hardware acceleration to be added.
- Support for the use of a scan test methodology for the standard cell logic and *Built-In-Self-Test* (BIST) for the TCM.

Providing this complete high-frequency subsystem frees the SoC designer to concentrate on design issues unique to their system. The synthesizable nature of the device eases integration into ASIC technologies.

Figure 1-1 on page 1-3 shows the block diagram of the ARM966E-S processor.

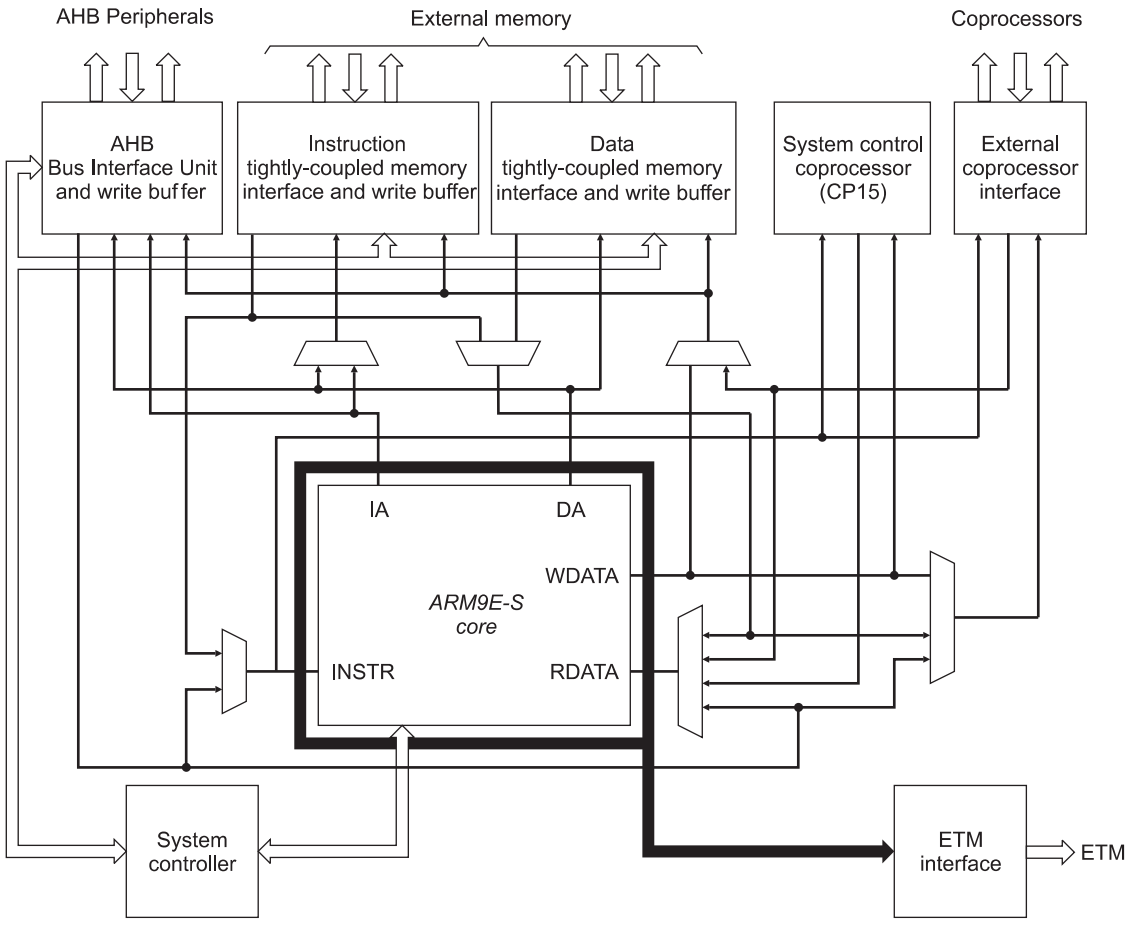


Figure 1-1 ARM966E-S processor block diagram

Table 1-1 Location of block descriptions

Block	Location of description
ARM9E-S	*ARM9E-S Technical Reference Manual*
AHB Bus Interface Unit and write buffer	Chapter 5 *Bus Interface Unit*
TCM interface	Chapter 4 *Tightly-Coupled Memory Interface*
System control coprocessor (CP15)	Chapter 2 *Programmer's Model*

Table 1-1 Location of block descriptions (continued)

Block	Location of description
External coprocessor interface	Chapter 6 *Coprocessor Interface*
System controller	Chapter 2 *Programmer's Model*
ETM interface	Chapter 8 *Embedded Trace Macrocell Interface*

1.2 Silicon revision information

This manual is for revision r2p1 of the ARM966E-S processor. Differences from revision r2p0 are:

- The ID code (CP15 register c0) value has changed, the new value is `0x41259661`.
- The polarity of the clock on the test wrapper lock-up latch has been corrected so that **WSO** changes on the negative edge of the system clock when **WEDGE** is 0.
- Data tracing for an STC instruction has been corrected.
- The criteria that make a TCM access sequential have changed therefore, the behavior of **ITCMSEQ** and **DTCMSEQ** have changed. For more information, see Chapter 4 *Tightly-Coupled Memory Interface*.
- The behavior of the **ITCMADDR** and **DTCMADDR** outputs has been modified to reduce unnecessary changes of state during ITCM and DTCM idle cycles.
- The AHB address no longer changes while the bus is idle during a locked AHB transfer. ARM966E-S r2p0 and r2p1 are AMBA AHB compliant.
- **DTCMCANCEL** and **ITCMCANCEL** are now asserted when all of the data required for a read is provided by the respective TCM write buffers.
- **HLOCK** is no longer unnecessarily asserted for write accesses preceeding a SWP access.

Chapter 2
Programmer's Model

This chapter describes the ARM966E-S registers and provides information for programming the microprocessor. It contains the following sections:

2.1 About the programmer's model

The programmer's model for the ARM966E-S processor primarily consists of the ARM9E-S core programmer's model (see *About the ARM9E-S programmer's model* on page 2-3). Additions to this model are required to control the operation of the ARM966E-S internal coprocessors, and any coprocessor connected to the external coprocessor interface.

There are two internal coprocessors within the ARM966E-S processor:

- CP14 within the ARM9E-S core enables software access to the debug communications channel
- CP15 enables configuration of the *Tightly-Coupled Memory* (TCM) and write buffer and other ARM966E-S system options such as big-endian or little-endian operation.

The registers defined in CP14 are accessible with MCR and MRC instructions. These are described in *The debug communications channel* on page 7-19.

The registers defined in CP15 are accessible with MCR and MRC instructions. These are described in *CP15 registers* on page 2-5.

Any coprocessors registers and operations, attached to the external coprocessor interface, are accessible with appropriate coprocessor instructions.

2.2 About the ARM9E-S programmer's model

The ARM9E-S core implements the ARMv5TE architecture, that includes the 32-bit ARM instruction set and the 16-bit Thumb instruction set. For a description of both instruction sets, see the *ARM Architecture Reference Manual*.

2.2.1 Data Abort model

The ARM9E-S core implements the *base restored data abort model*, that differs from the *base updated data abort model* implemented by ARM7TDMI.

The difference in the Data Abort model affects only a very small section of operating system code, the Data Abort handler. It does not affect user code. With the *base restored data abort model*, when a Data Abort exception occurs during the execution of a memory access instruction, the base register is always restored by the processor hardware to the value the register contained before the instruction was executed. This removes the requirement for the Data Abort handler to unwind any base register update that might have been specified by the aborted instruction.

The *base restored data abort model* significantly simplifies the Data Abort handler software.

2.2.2 ARM966E-S processor abort sources

Data Aborts can be generated from the following sources:

- data transactions to the AHB memory space that return an AHB ERROR response (except for buffered writes)
- data TCM reads for which the **DTCMERROR** input is asserted
- instruction TCM data reads for which the **ITCMERROR** input is asserted
- unaligned data accesses whenever data alignment checking is enabled.

Prefetch aborts can be generated from the following sources (if the instruction fetched is executed):

- instruction fetches from the AHB memory space that return an AHB ERROR response
- instruction fetches from the instruction TCM for which the **ITCMERROR** input is asserted.

Executing a BKPT instruction causes the Prefetch Abort exception to be entered. For more information, see the *ARM Architecture Reference Manual*.

2.2.3 PLD instruction execution

The *Preload* (PLD) instruction is treated as a NOP by the ARM966E-S processor. For more information about this instruction, see the *ARM Architecture Reference Manual*.

2.3 CP15 registers

CP15 enables configuration of the TCM and write buffer. It also enables other ARM966E-S system options such as big-endian or little-endian operation.

The ARM966E-S coprocessor 15 registers are described in the following sections:

- *CP15 register map summary*
- *CP15 c0, ID Code Register* on page 2-6
- *CP15 c0, TCM Size Register* on page 2-6
- *CP15 c1, Control Register* on page 2-7
- *CP15 c7, Core Control Register* on page 2-9
- *CP15 c13, Trace Process Identifier Register* on page 2-10
- *CP15 c15, Test and Configuration Register* on page 2-11.

2.3.1 CP15 register map summary

The ARM966E-S processor uses CP15 for system control functions. Table 2-1 shows the register map for CP15.

Table 2-1 CP15 register map

Register	Read	Write
CP15 c0	ID code	Unpredictable
CP15 c0	Tightly-coupled memory size	Unpredictable
CP15 c1	Control	Control
CP15 c2-c6	Unpredictable	Unpredictable
CP15 c7	Unpredictable	Core control
CP15 c8-c12	Unpredictable	Unpredictable
CP15 c13	Trace process identifier	Trace process identifier
CP15 c14	Unpredictable	Unpredictable
CP15 c15	Test	Test

Note

Register c0 and register c15 provide access to more than one register. The register access depends on the value of the Opcode_2 field. See the register descriptions in this section for more information.

2.3.2 CP15 c0, ID Code Register

This is a read-only register that returns a 32-bit device ID code. You can access the ID Code Register by reading CP15 c0 with the Opcode_2 field set to any value other than 2. For example:

```
MRC p15, 0, <Rd>, c0, c0, {0, 1, 3-7}; returns ID Code Register
```

Figure 2-1 shows the format of the ID Code Register.

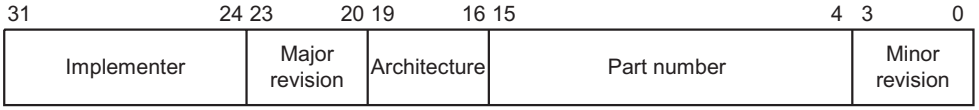


Figure 2-1 ID Code Register

Table 2-2 shows the bit fields of the ID Code Register.

Table 2-2 ID Code Register bit functions

Bit	Function	Value
[31:24]	Implementer	0x41
[23:20}	Major specification revision	0x2
[19:16]	ARM architecture v5TE	0x5
[15:4]	Part number	0x966
[3:0]	Minor specification revision	0x1

2.3.3 CP15 c0, TCM Size Register

This is a read-only register that returns the size of the Instruction and Data TCM attached to the ARM966E-S processor.

You can access the TCM Size Register by reading CP15 c0 with the Opcode_2 field set to 2. For example:

```
MRC p15, 0, <Rd>, c0, c0, 2; returns Tightly-Coupled Memory Size Register
```

Figure 2-2 on page 2-7 shows the format of the TCM Size Register.

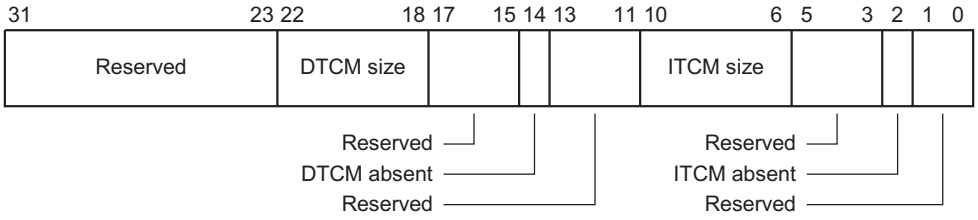


Figure 2-2 TCM Size Register

Table 2-3 shows the bit fields of the TCM Size Register.

Table 2-3 TCM Size Register bit functions

Bit	Function	Value
[31:23]	Reserved	b000000000
[22:18}	Data TCM size	Implementation specific
[17:15]	Reserved	b000
[14]	Data TCM absent	Implementation specific
[13:11]	Reserved	b000
[10:6]	Instruction TCM size	Implementation specific
[5:3]	Reserved	b000
[2]	Instruction TCM absent	Implementation specific
[1:0]	Reserved	b00

The memory size parameters take the values shown in *TCM interface signals* on page A-6. If TCM size is set to zero, the TCM absent bit is set to 1.

2.3.4 CP15 c1, Control Register

This register contains the global control bits of the ARM966E-S processor. All reserved bits must either be written with zero or one, as indicated, or written using read-modify-write. The reserved bits have an Unpredictable value when read. To read and write this register:

```
MRC p15, 0, <Rd>, c1, c0, 0; read Control Register
MCR p15, 0, <Rd>, c1, c0, 0; write Control Register
```

Figure 2-3 shows the format of the Control Register.

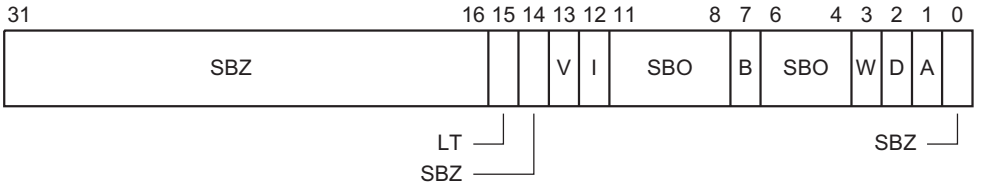


Figure 2-3 Control Register

Table 2-4 shows the bit fields of the Control Register.

Table 2-4 Control Register bit functions

Bit	Function
[31:16]	Reserved. When read, returns an Unpredictable value. When written, Should Be Zero.
[15]	Load PC Thumb disable bit: 0 = Loading PC sets the T bit 1 = Loading PC does not set the T bit. Reset clears this bit.
[14]	Reserved. When read, returns an Unpredictable value. When written, Should Be Zero.
[13]	Location of exception vectors: 0 = Normal exception vectors selected, address range = `0x00000000-0x0000001C` 1 = High exception vectors selected, address range = `0xFFFF0000-0xFFFF001C`. At Reset, the VINITHI pins determine the value of this bit.
[12]	Instruction TCM enable: 0 = All accesses to the instruction memory space access the AMBA AHB 1 = All accesses to the fixed instruction memory space access the instruction TCM interface. At Reset, the INITRAM pins determine the value of this bit.
[11:8]	Reserved. Should Be One.
[7]	Endianness. This bit configures the ARM966E-S processor to rename the low four-byte addresses within a 32-bit word: 0 = Little-endian operation 1 = Big-endian operation.Reset clears this bit.
[6:4]	Reserved. Should Be One.
[3]	BIU write buffer enable: 0 = All stores to the AMBA AHB are treated as nonbufferable 1 = All stores to the fixed bufferable space of the AMBA AHB are treated as buffered writes. Reset clears this bit.
[2]	Data TCM enable. At Reset, the INITRAM pins determine the value of this bit.
[1]	Address alignment fault checking enable: 0 = Fault checking of address alignment disabled 1 = Fault checking of address alignment enabled. Reset clears this bit.
[0]	Reserved. When read, returns an Unpredictable value. When written, Should Be Zero.

2.3.5 CP15 c7, Core Control Register

You can use a write to this register, to perform *wait for interrupt* and *drain write buffer* operations.

Wait for interrupt

This operation enables the ARM966E-S processor to enter a low-power standby mode. When the operation is invoked, the clock enable to the processor core is negated until either an interrupt or a debug request occurs. This function is invoked by a write to Register 7. The following ARM instruction causes this to occur:

```
MCR p15, 0, <Rd>, c7, c0, 4; wait for interrupt
```

Note

This is the preferred encoding that must be used by new software. For compatibility with existing software, ARM966E-S processor also supports the following ARM instruction that has the same affect:

```
MCR p15, 0, <Rd>, c15, c8, 2; wait for interrupt
```

This stalls the processor from the time that the instruction is executed until **nFIQ**, **nIRQ**, or **EDBGRQ** are asserted. Also, if the debugger sets the debug request bit in the EmbeddedICE-RT control register then this causes the wait-for-interrupt condition to terminate.

In the case of **nFIQ** and **nIRQ**, the processor core is woken up regardless of whether the interrupts are enabled or disabled (that is, independent of the I and F bits in the processor CPSR). The debug-related waking only occurs if **DBGEN** is HIGH, that is, only when debug is enabled.

If interrupts are enabled, the ARM9E-S core is guaranteed to take the interrupt before executing the instruction after the wait for interrupt. If debug request is used to wake up the system, the processor enters debug-state before executing any more instructions.

Wait for interrupt does not prevent the write buffer from emptying.

Drain write buffers

This CP15 operation causes instruction execution to be stalled until the AHB and TCM write buffers are emptied. This operation is useful in real-time applications where the processor must be sure that a write to a peripheral has completed before program execution continues. An example is where a peripheral in a bufferable region is the

source of an interrupt. When the interrupt has been serviced, the request must be removed before interrupts can be re-enabled. This can be ensured if a drain write buffer operation separates the store to the peripheral and the enable interrupt functions.

The drain write buffer operation is invoked by a write to Register 7 using the following ARM instruction:

```
MCR p15, 0, <Rd>, c7, c10, 4; drain write buffer
```

Note

This stalls the processor core until any outstanding accesses in the write buffers have been completed, that is, until all data has been written to memory.

2.3.6 CP15 c13, Trace Process Identifier Register

This register enables the real-time trace tools to identify the currently executing process in multitasking environments.

The **ETMPROCID[31:0]** pins reflect the contents of the Trace Process Identifier Register.

Note

Writing to the Trace Process Identifier Register sets the ETMPROCIDWR signal for one clock cycle.

To read and write this register:

```
MRC p15, 0, <Rd>, c13, {c0 - c15}; read Trace Process Identifier Register
MCR p15, 0, <Rd>, c13, {c0 - c15}; write Trace Process Identifier Register
```

Figure 2-4 shows the format of the Trace Process Identifier Register.

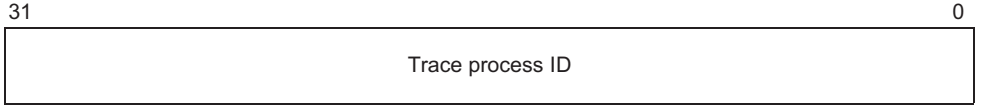


Figure 2-4 Trace Process Identifier Register

2.3.7 CP15 c15, Test and Configuration Register

This register provides access to:

- the Instruction and Data TCM test features
- the configuration control features.

Table 2-5 shows the register map for CP15 c15.

Table 2-5 Register 15, Test and Configuration Register

Register	Read	Write
Configuration Control Register	`MRC p15, 1, <Rd>, c15, c1, 0`	`MCR p15, 1, <Rd>, c15, c1, 0`
BIST Control Register	`MRC p15, 1, <Rd>, c15, c0, 1`	`MCR p15, 1, <Rd>, c15, c0, 1`
Instruction BIST Address Register	`MRC p15, 1, <Rd>, c15, c0, 2`	`MCR p15, 1, <Rd>, c15, c0, 2`
Instruction BIST General Register	`MRC p15, 1, <Rd>, c15, c0, 3`	`MCR p15, 1, <Rd>, c15, c0, 3`
Data BIST Address Register	`MRC p15, 1, <Rd>, c15, c0, 6`	`MCR p15, 1, <Rd>, c15, c0, 6`
Data BIST General Register	`MRC p15, 1, <Rd>, c15, c0, 7`	`MCR p15, 1, <Rd>, c15, c0, 7`

Note

The Opcode_1 field is set HIGH when accessing Register 15. The Opcode_2 field is used to index registers within the Register 15 register map.

2.3.8 Configuration Control Register

The Configuration Control Register enables modification of the default behavior of the ARM966E-S processor. This might be necessary in situations where the behavior of previous revisions of the ARM966E-S processor is required, or where particular features are not compatible with a system design.

Figure 2-5 on page 2-12 shows the format of the Configuration Control Register.

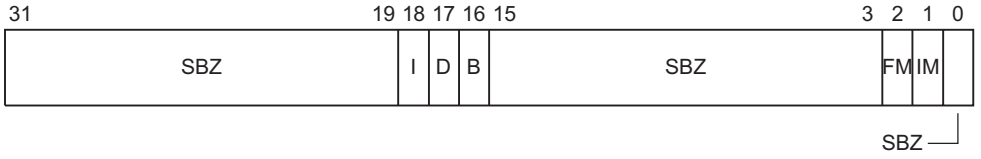


Figure 2-5 Configuration Control Register

Table 2-6 shows the bit fields of the Configuration Control Register.

Table 2-6 Configuration Control Register bit functions

Bit	Function
[31:19]	Reserved. Should Be Zero.
[18]	Instruction TCM order bit: 0 = ITCM read accesses not stalled by data in ITCM write buffer 1 = ITCM read accesses stalled if ITCM write buffer contains data. Asserting this bit ensures that TCM accesses are performed in the order generated by the ARM9E-S core and that writes are committed to memory before subsequent reads are done. Asserting this bit when data is still in the TCM write buffer stalls any subsequent TCM access until the buffer is empty. Reset clears this bit.
[17]	Data TCM order bit: 0 = DTCM read accesses not stalled by data in DTCM write buffer 1 = DTCM read accesses stalled if DTCM write buffer contains data. Asserting this bit ensures that TCM accesses are performed in the order generated by the ARM9E-S core and that writes are committed to memory before subsequent reads are done. Asserting this bit when data is still in the TCM write buffer stalls any subsequent TCM access until the buffer is empty. Reset clears this bit.
[16]	AHB instruction prefetch buffer disable bit: 0 = Instruction prefetch buffer enabled 1 = Instruction prefetch buffer disabled. When this bit is set, all instruction accesses are performed as nonsequential transfers. This results in a number of idle cycles between each access. Reset clears this bit. See *AHB instruction prefetch buffer* on page 5-3.
[15:3]	Reserved. Should Be Zero.
[2]	FIQ interrupt mask when ETM FIFO is full: 0 = **nFIQ** enables core clocks until interrupt is serviced, but clocks are disabled on exit from FIQ mode 1 = **nFIQ** cannot enable core clocks when **FIFOFULL** is HIGH. Reset sets this bit.
[1]	IRQ interrupt mask when ETM FIFO is full: 0 = **nIRQ** enables core clocks until interrupt is serviced, but clocks are disabled on exit from IRQ mode 1 = **nIRQ** cannot enable core clocks when **FIFOFULL** is HIGH. Reset clears this bit.
[0]	Reserved. Should Be Zero.

2.3.9 BIST Control Register

Table 2-7 shows the bit assignments within the BIST Control Register.

Note

If the ARM966E-S BIST hardware is not present, the relevant BIST size field is always read back as all zeros.

Table 2-7 BIST Control Register

Bit	Meaning when written	Meaning when read
[31:21]	Instruction TCM BIST size.	Instruction TCM BIST size
[20]	Reserved. Should Be Zero.	Instruction TCM BIST complete flag
[19]	Reserved. Should Be Zero.	Instruction TCM BIST fail flag
[18]	Instruction TCM BIST enable.	Instruction TCM BIST enable
[17]	Instruction TCM BIST pause.	Instruction TCM BIST pause
[16]	Instruction TCM BIST start strobe.	Instruction TCM BIST running flag
[15:5]	Data TCM BIST size.	Data TCM BIST size
[4]	Reserved. Should Be Zero.	Data TCM BIST complete flag
[3]	Reserved. Should Be Zero.	Data TCM BIST fail flag
[2]	Data TCM BIST enable.	Data TCM BIST enable
[1]	Data TCM BIST pause.	Data TCM BIST pause
[0]	Data TCM BIST start strobe.	Data TCM BIST running flag

At reset, all bits are cleared LOW except for the BIST size fields. Before a BIST operation starts, BIST must be enabled. When BIST is enabled to test one or both tightly-coupled memories, the TCM being tested is automatically disabled by clearing its enable bit in CP15 Register c1. This is to prevent the programmer inadvertently using the TCM following a BIST operation, because the BIST algorithm corrupts the TCM contents.

The BIST size field determines the size of the BIST operation. The value written to this field N, is decoded as follows:

BIST size in bytes = 2^{N+2}

Table 2-8 shows some examples.

Table 2-8 BIST size encoding examples

Instruction RAM BIST size [31:21]	N	Size of test
b000000 00001 (minimum)	1	8 bytes
b000000 00100	4	64 bytes
b000000 00111	7	512 bytes
b000000 01000	8	1KB
b000000 01010	10	4KB
b000000 01111	15	128KB
b000000 11000 (maximum)	24	64MB

Note

BIST size bits [31:26] Should Be Zero.

Writing to the BIST Control Register with bit[0] set initiates a Data TCM BIST operation.

Writing to the BIST Control Register with bit[16] set initiates an Instruction TCM BIST operation.

Running BIST operations

You can run Instruction and Data BIST operations individually or concurrently. You must set up the Size, Pause and Enable bits within the BIST Control Register prior to initiating a BIST operation.

Reading the BIST Control Register Returns the status of the BIST operations. See *BIST of tightly-coupled memory* on page 9-4 for a detailed description of the BIST support and the additional BIST registers.

Chapter 3
Memory Map

This chapter describes the ARM966E-S processor fixed memory map implementation. It contains the following sections:

3.1 About the ARM966E-S memory map

The ARM966E-S processor couples Instruction and Data TCM memories of configurable size to the ARM9E-S core. This enables high-speed operation without incurring the performance and power penalties of accessing the system bus. Write buffers decouple the ARM9E-S core from wait states incurred when accessing the AHB bus and the TCMs.

The fixed memory map provides simple control over the AHB write buffers and TCM. Figure 3-1 shows the ARM966E-S memory map.

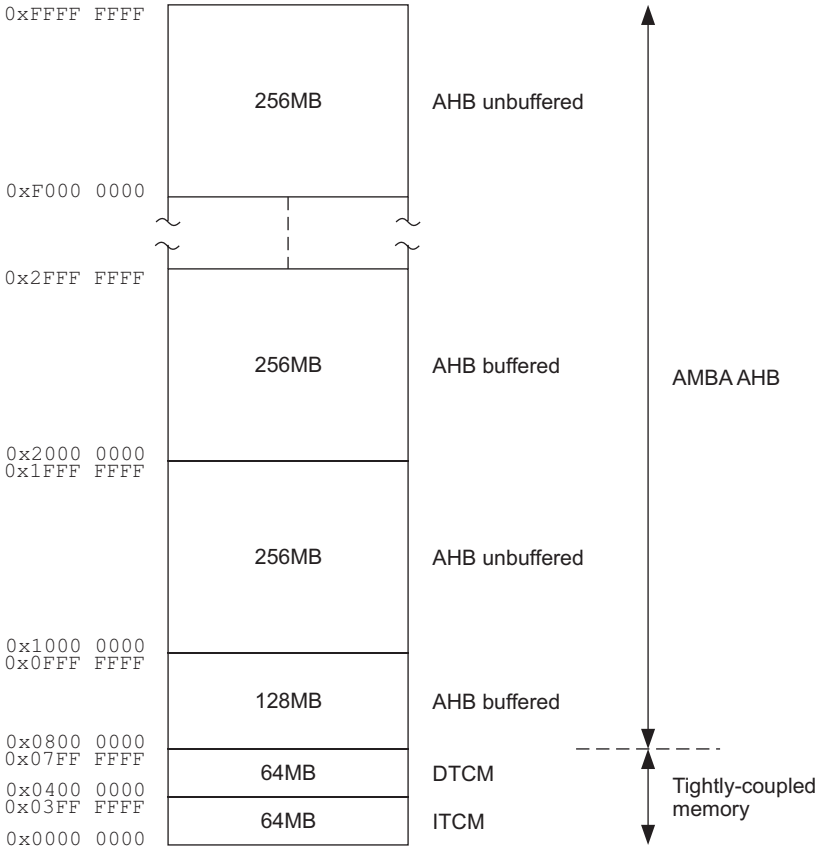


Figure 3-1 ARM966E-S memory map

3.2 Tightly-coupled memory address space

The *Tightly-coupled Memory* (TCM) is at the bottom of the memory map. The memory map allocates the bottom 64MB space for the *Instruction TCM* (ITCM) and the next 64MB to *Data TCM* (DTCM).

In practice, each TCM is likely to be much smaller than the 64MB allowable. The address decode is implemented so that each memory is aliased throughout its 64MB range. Figure 3-2 shows an example of a 16KB ITCM aliased through the 64MB address space.

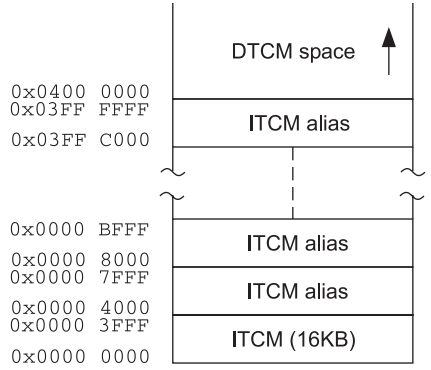


Figure 3-2 ITCM aliasing example

All accesses to addresses above the 128MB combined TCM address space result in AMBA AHB transfers controlled by the *Bus Interface Unit* (BIU).

An instruction fetch from the ARM9E-S core to the DTCM address space goes to the AHB, regardless of whether the DTCM is enabled. A data interface access from the ARM9E-S core can access both the DTCM and the ITCM. The ability to additionally access the ITCM is required for fetching inline literals within code, for programming of the ITCM, and for debugging purposes.

When a TCM is disabled, all accesses to its address space go to the AHB. When enabled, the TCM must be programmed before use. The value of the input pin **INITRAM** during Reset enables or disables the TCMs. Several boot options are available using **INITRAM** and the exception vectors location pin **VINITHI**. These are discussed in *Using INITRAM input pin* on page 4-4.

3.3 Bufferable write address space

The use of the AHB write buffer is controlled by both the CP15 c1 Control Register and the fixed address map.

When the ARM966E-S processor comes out of Reset, the AHB write buffer is disabled by default. All data writes to the AHB are performed as unbuffered. The ARM9E-S core is stalled until the BIU completes the write on the AHB interface.

When the AHB write buffer is enabled by writing to CP15 control register bit 3 (see *CP15 registers* on page 2-5), the data address (**DA[31:0]**) from the ARM9E-S core controls whether the AHB write buffer is used. If **DA[28]** is set, the write is unbuffered. If DA[28] is clear, the write is buffered and the BIU write buffer FIFO is used. Buffered writes enable the core to continue program execution while the write is performed on the AHB. If the AHB write buffer is full, the core stalls until space in the buffer becomes available. See *AHB write buffer* on page 5-6 for details of the BIU and AHB write buffer behavior.

Note

Writes to TCM address space do not write through to the AHB if the TCM being accessed is enabled. Writes to the address space of a disabled TCM are buffered AHB writes when the AHB write buffer is enabled or unbuffered AHB writes when the AHB write buffer is not enabled.

Chapter 4
Tightly-Coupled Memory Interface

This chapter describes the ARM966E-S *Tightly-Coupled Memory* (TCM) interface. It contains the following sections:

4.1 About the TCM interface

The ARM966E-S processor supports both instruction and data TCMs. You can use the DTCM and ITCM to store real-time and performance-critical code. The TCMs are instantiated externally to the ARM966E-S macrocell providing for flexibility in the design of the memory subsystem. The system designer can select memory type and optimize the memory subsystem for power or speed.

The features of the TCM interface include:

- independent ITCM and DTCM sizes of 0KB or 1KB-64MB in power-of-two increments
- software visibility and programmability of TCM size and enable
- boot control for ITCM
- data access to the ITCM for literal pool accesses in code
- simple SRAM-style interface supporting both reads and writes
- variable TCM wait state control for ITCM and DTCM
- ability to indicate sequential and nonsequential accesses.

The ARM966E-S processor contains a TCM controller that:

- schedules requests to the TCM interface
- handshakes with the ARM966E-S memory system controller to acknowledge when requests have been serviced
- returns TCM read data back to the ARM9E-S core.

The TCMs are located in the TCM address space. See Chapter 3 *Memory Map*.

4.2 TCM size

The TCM supports a programmable memory size with a fixed offset defined in the ARM966E-S memory map. The TCM size can be determined by reading CP15 register c0 with the Opcode_2 field set to 2. See *CP15 c0, TCM Size Register* on page 2-6.

Table 4-1 shows how the **ITCMSIZE[4:0]** and **DTCMSIZE[4:0]** inputs control TCM RAM sizes. The supported sizes are 0 and 2^n KB for n = 0 to 16.

Table 4-1 Supported TCM RAM sizes

Value	TCM size
b00000	0KB
b00001	1KB
b00010	2KB
b00011	4KB
b00100	8KB
b00101	16KB
b00110	32KB
b00111	64KB
b01000	128KB
b01001	256KB
b01010	512KB
b01011	1MB
b01100	2MB
b01101	4MB
b01110	8MB
b01111	16MB
b10000	32MB
b10001	64MB

For more information, see *TCM interface signals* on page A-6.

4.3 Enabling TCM

This section describes how to use the two mechanisms for controlling the enable of the TCM:

- *Using INITRAM input pin*
- *Using CP15 c1 Control Register* on page 4-5.

4.3.1 Using INITRAM input pin

The **INITRAM** pin enables the ARM966E-S processor to boot with both external instruction and data memory blocks either enabled or disabled. Two resets are described in the following sections:

- *Reset with INITRAM LOW*
- *Reset with INITRAM HIGH*.

Reset with INITRAM LOW

If **INITRAM** is held LOW during Reset, the ARM966E-S processor comes out of Reset with both external instruction and data memory disabled. All accesses to external instruction and data memory space go to the AHB. The TCMs can then be individually or jointly enabled by writing to the CP15 c1 Control Register.

Reset with INITRAM HIGH

If **INITRAM** is held HIGH during Reset, the ARM966E-S processor comes out of Reset with both external instruction and data memory enabled. This is normally used for a warm Reset where the TCM has already been programmed before the application of **HRESETn** to the ARM966E-S processor. In this case, the TCM contents are preserved and the ARM966E-S processor can run directly from the TCM following Reset. Either one or both TCMs can be further disabled or enabled by writing to the CP15 c1 Control Register.

Note

If **INITRAM** is held HIGH during a cold Reset (the TCM has not previously been initialized), the **VINITHI** pin must be set HIGH to ensure that the ARM966E-S processor boots from `0xFFFF 0000`, that is in AHB address space and is substantially outside the TCM address space. This is necessary because if **VINITHI** is LOW, the ARM966E-S processor attempts to boot from `0x0000 0000`, and this selects the uninitialized ITCM.

4.3.2 Using CP15 c1 Control Register

When out of Reset, the state of CP15 c1 Control Register determines the behavior of the TCM.

Enabling the ITCM

You can enable the ITCM interface by setting bit [12] of the CP15 c1 Control Register. You must access this register in a read-modify-write fashion to preserve the contents of the bits not being modified. See *CP15 registers* on page 2-5 for details of how to read and write the CP15 c1 Control Register. After you enable the ITCM interface, all future ARM9E-S core instruction fetches and data accesses to the ITCM address space cause the ITCM interface to be accessed as shown in Figure 3-1 on page 3-2.

Enabling the ITCM interface greatly increases the performance of the ARM966E-S processor because the majority of accesses to it can be performed with no stall cycles, whereas accessing the AHB might cause several stall cycles for each access. Care must be taken to ensure that the ITCM interface is appropriately initialized before it is enabled and used to supply instructions to the ARM9E-S core. If the core executes instructions from uninitialized ITCM interface, the behavior is Unpredictable.

Disabling the ITCM

You can disable the ITCM interface by clearing bit [12] of the CP15 c1 Control Register. After you disable the ITCM interface, all further ARM9E-S core instruction fetches access the AHB. If the core performs a data access to the ITCM address space as shown in Figure 3-1 on page 3-2, an AHB access is performed.

The contents of the memory are preserved when it is disabled. If it is re-enabled, accesses to previously initialized memory locations return the preserved data.

Note

The TCM write buffers must be drained before disabling the ITCM interface.

Enabling the DTCM

You can enable the DTCM interface by setting bit [2] of the CP15 c1 Control Register. See *CP15 registers* on page 2-5 for details of how to read and write this register. After you enable the DTCM interface, all future read and write accesses to the DTCM address space, as shown in Figure 3-1 on page 3-2, cause the DTCM interface to be accessed.

Disabling the DTCM

You can disable the DTCM by clearing bit [2] of the CP15 c1 Control Register. After you disable the DTCM, all further reads and writes to the DTCM address space, as shown in Figure 3-1 on page 3-2, access the AHB.

Note

The TCM write buffers must be drained before disabling the DTCM interface.

4.4 TCM write buffers

To minimize the occurrence of stall cycles and to decouple the processor from memory wait states, there are write buffers in the DTCM and ITCM interfaces.

Each TCM write buffer is two entries deep. Each entry is an address and data pair. In normal operation, the data for a write access to the TCM address space is held in the TCM write buffer until it is forced out by another write to the TCM address space or by natural drain when there are no read requests to the TCM address space.

Write accesses from the core always go into the TCM write buffer. If there is space in the TCM write buffer, writes are always single-cycle operations regardless of external TCM wait states. If there is no space in the TCM write buffer, any write access stalls the ARM9E-S core until a TCM write buffer entry becomes free.

4.4.1 TCM order bit

In normal operation, the TCM write buffer drains naturally into the TCM whenever there are no read accesses to the TCM address space. One effect of this drain mechanism is that read and write accesses to the TCM can be in a different order from that issued by the ARM9ES core. If the TCM write buffer contains the data required by a read access, data is returned from the buffer. Otherwise, a read can bypass a write that is pending in the TCM write buffer when a read is to a different address.

Read and write accesses to DTCM and ITCM can be maintained in the order that the ARM9E-S core generated them by using the TCM order bits in the CP15 c15 Configuration Control Register. See *Configuration Control Register* on page 2-11.When the TCM order bit is set, the TCM write buffer is still used but any subsequent read accesses to the TCM are stalled until the buffer is emptied.

To ensure correct operation, perform a drain-write-buffer operation immediately prior to setting the TCM order bit. To drain the TCM and AHB write buffers, use a CP15 c7 core control operation. See *Drain write buffers* on page 2-9.

To ensure correct ordering of data reads and writes to the DTCM and ITCM, set the DTCM and ITCM order bits. There is no enforced ordering between instruction reads and data accesses to the ITCM.

Note

The **ITCMSIZE[4:0]** and **DTCMSIZE[4:0]** inputs are used to ensure that write accesses to aliased addresses return the correct data when read. The **ITCMSIZE[4:0]** and DTCMSIZE[4:0] values must match instantiated memory size for this to operate correctly. See *TCM interface signals* on page A-6 for details of the **ITCMSIZE** and **DTCMSIZE** values.

4.5 TCM error detection signals

Large SRAM arrays are susceptible to errors caused by alpha particle radiation. These errors can result in incorrect data being returned. You can use parity checking or some form of error detection outside the ARM966E-S macrocell to detect these errors.

To enable the ARM966E-S processor to support external error detection on the TCMs, there is one error signal for each of the TCMs:

- **DTCMERROR**
- **ITCMERROR**.

The error signals inform the processor of error conditions during TCM read accesses and are ignored during write accesses. These signals are valid in the same clock cycle as the data returned from the TCM, and the processor ignores them at all other times.

Error detection is performed externally to the ARM966E-S macrocell. If error support is not required, **DTCMERROR** and **ITCMERROR** must be tied LOW. Because the ARM966E-S processor is capable of performing byte accesses, parity information must be generated for each byte. The parity bit must be generated at the same time as the data is written to memory. Data is always read from the TCMs in 32-bit words and a parity error in any one byte must be returned to the core as an error.

For data reads from either ITCM or DTCM, any error returned causes a Data Abort exception. The exception handler determines what corrective action, if any, to take.

For instruction fetches from the ITCM, any error returned causes a Prefetch Abort exception if the ARM966E-S processor tries to execute the returned instruction.

Note

If all the data is returned from the TCM write buffer, **ITCMERROR** or **DTCMERROR** is ignored. If only part of the data is returned from the TCM write buffer, **ITCMERROR** or DTCMERROR is sampled. To prevent errors from uninitialized locations, memory must be initialized so that spurious read errors are not generated.

4.6 TCMSEQ signals

When the **DTCMSEQ** and **ITCMSEQ** signals are asserted, the current TCM memory address is sequential to the previous address, that is, the current **ITCMADDR** or **DTCMADDR** is equal to the previous **ITCMADDR** or DTCMADDR plus one, and the current memory access is of the same type (read or write) as the previous access.

Note

To optimize signal timing, **ITCMSEQ** and **DTCMSEQ** are not derived from direct comparisons of current and previous states of the **ITCMADDR** or **DTCMADDR** and the **ITCMnRW** or **DTCMnRW** signals, but from other internal signals. This can result in sequential TCM accesses that are not marked as sequential, that is, **ITCMSEQ** or DTCMSEQ is not asserted. The **ITCMSEQ** and DTCMSEQ signals are valid when the relevant **ITCMCS** or DTCMCS signal is asserted.

4.7 Interface operation

This section describes the operation of the TCM interfaces.

4.7.1 Single-cycle and multicycle accesses

This section describes:

- *Single-cycle TCM interface*
- *Multicycle TCM interface* on page 4-11.

Single-cycle TCM interface

Figure 4-1 shows a mixture of read and write operations and that the TCM must be able to support for back to back operations.

When **DTCMWAIT** is not asserted, the TCM controller expects all read and write operations to be single-cycle. **DTCMWAIT** can be tied LOW if a memory always performs single-cycle accesses. The ITCM operation is identical. **DTCMCANCEL** is HIGH in the second cycle and so the data in A does not get used.

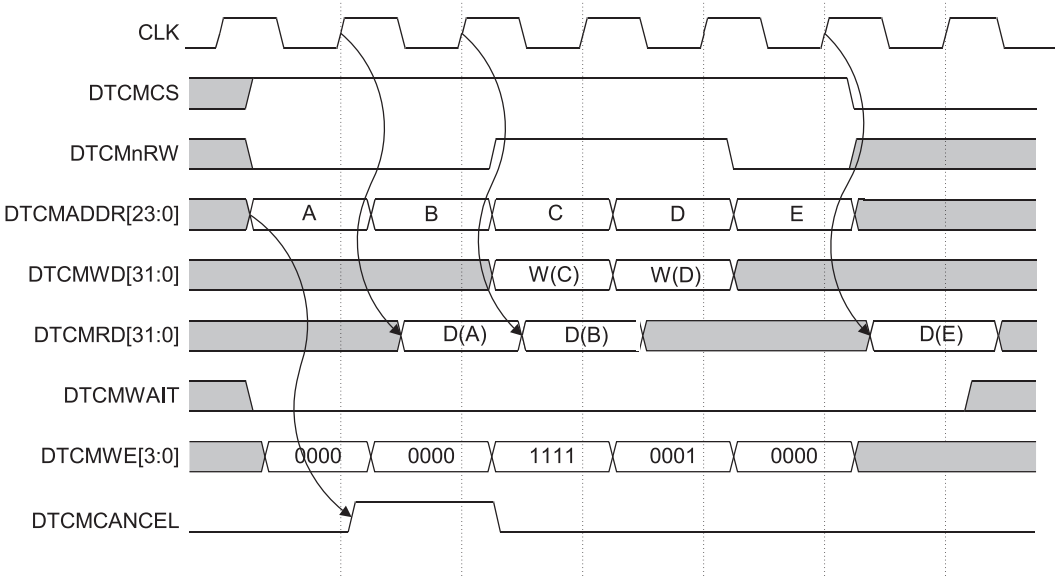


Figure 4-1 Single-cycle TCM read and write

Note

The **ITCMERROR** and **DTCMERROR** signals are valid in the same clock cycle as **ITCMRD[31:0]** or **DTCMRD[31:0]**. For more information, see *TCM error detection signals* on page 4-8.

Multicycle TCM interface

If a TCM is not able to service a memory request in a single-cycle, it must assert its wait signal to stall the ARM966E-S processor. For example, if a TCM is not able to return read data in the cycle following the request, its wait signal must be asserted in the request cycle before the next rising **CLK** edge. The wait signal must be deasserted in the cycle prior to the read data being returned.

Figure 4-2 on page 4-12 shows TCM read/write operations to a DTCM that requires two cycles for all accesses. On each new memory request, the DTCM asserts **DTCMWAIT** to inform the ARM966E-S processor that it is not able to service the memory request in that cycle. The ARM966E-S processor stalls until **DTCMWAIT** is deasserted. To aid in efficient use of multicycle memories, if the current access is sequential to the previous access, the ARM966E-S processor asserts **DTCMSEQ**.

The **TCMWAIT** signal is ignored until the ARM966E-S processor has started its memory access and **TCMWAIT** is used to stall the core until the memory access is complete. This makes it easy to generate wait state controllers for memory accesses because the state of **TCMWAIT** when there is no memory access does not matter.

When the core is stalled because of **TCMWAIT**, the TCM interface signals are no longer valid.

Design of single-port DMA controllers is more difficult because the ARM966E-S processor can start a new memory access even when **TCMWAIT** is held active. Logic must be implemented to register all TCM interface signals for memory accesses that start when **TCMWAIT** is active.

The following pseudo-code gives additional information to the block diagram in Figure 4-10 on page 4-21:

```
while (DMA access)
    TCMWAIT = 1
    if (TCMS = 1)
        CS_reg = TCMCS
        ADDR_reg = TCMADDR
        WE_reg = TCMWE
        WD_reg = TCMWD
        RnW_reg = TCMRnW
        CANCEL_reg = TCMCANCEL
```

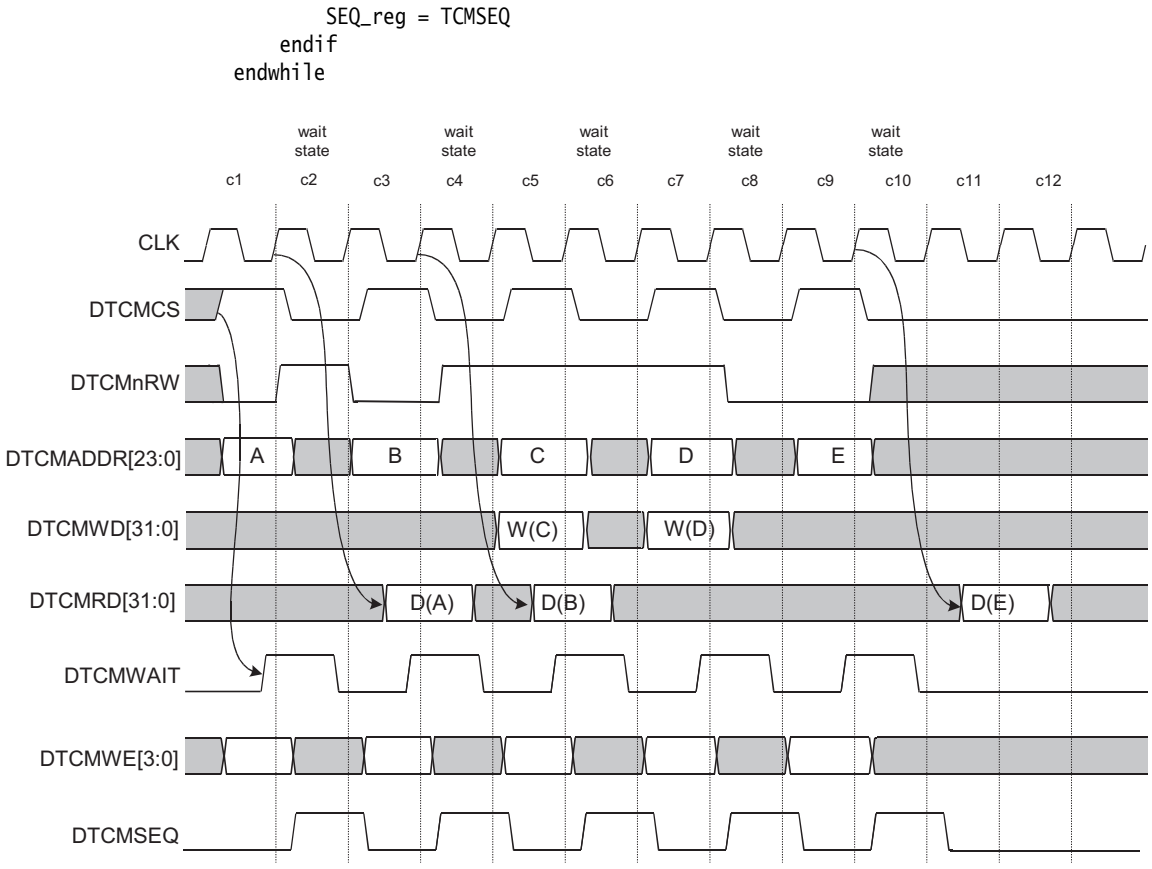



Figure 4-2 Two cycle TCM read and write

Note

The **ITCMERROR** and **DTCMERROR** signals are valid in the same clock cycle as **ITCMRD[31:0]** or **DTCMRD[31:0]**. For more information, see *TCM error detection signals* on page 4-8.

4.7.2 Speculative TCM read access

Some data reads from data memory and instruction reads from instruction memory are speculative. After a memory location is read, it is likely that the next read comes from the same area of memory. This second read is speculative because it is likely that it is correct but it is not certain. Misreads from speculative accesses cannot be aborted within one clock cycle.

Other situations where read accesses can be made to memory but the requested data is not used are:

- the data is available internally from the TCM write buffers
- the ARM9E-S core cancels the read.

In all the above situations, **DTCMCANCEL** or **ITCMCANCEL** is used to signal that the requested memory data is not required. If no waited cycles are added, **DTCMCANCEL** or **ITCMCANCEL** is asserted in the clock cycle following **DTCMCS** or **ITCMCS**. However, if waited cycles are added, then **DTCMCANCEL** or **ITCMCANCEL** can be asserted in any or all clock cycles from the clock cycle following **DTCMCS** or **ITCMCS** to the clock cycle in which **DTCMRD[31:0]** or ITCMRD[31:0] is valid. When asserted, the data requested from the memory is not used by the ARM9E-S core or attached coprocessors.

Canceled accesses

Typical situations where the **ITCMCANCEL** and **DTCMCANCEL** signals occur are:

- a change from one memory region to another, typically leaving TCM space
- the ARM9E-S core aborts the first data access in a series of back-to-back data accesses
- the data requested is available internally from the TCM write buffers
- the ARM9E-S core cancels a speculative instruction fetch.

Figure 4-3 on page 4-14 shows examples of canceled accesses to the DTCM. In cycle c5, **DTCMCANCEL** goes HIGH when the memory access is canceled as a result of the error generated for the TCM access, TCM1. TCM2 is canceled. In cycles c7 to c9, **DTCMCANCEL** goes HIGH because there has been a change of memory space and the next data address is an AHB address. **DTCMCANCEL** goes HIGH and the data is not used by the ARM9E-S core.

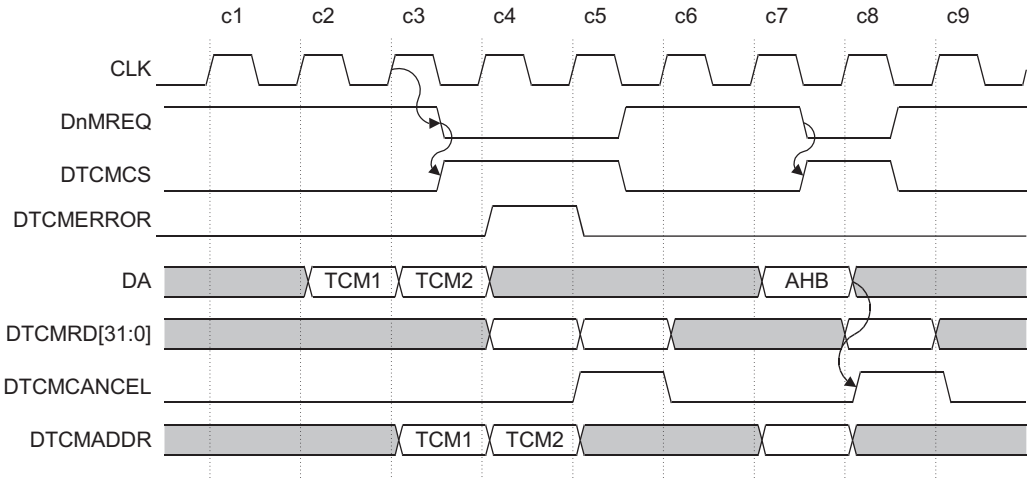


Figure 4-3 DTCM reads with cancels

4.8 TCM implementation examples

This section contains the following examples:

- *Simplest zero-wait-state RAM example*
- *Byte-banks of RAM example* on page 4-16
- *Multiple banks of RAM example* on page 4-17
- *Sequential RAM example* on page 4-18
- *Single or Multiple wait-state RAM example* on page 4-19
- *Dual port DMA-capable RAM example* on page 4-21.

Note

The examples in this section are for the DTCM. These are also applicable to the ITCM.

The additional logic required for implementing the examples in this section is the responsibility of the implementer.

4.8.1 Simplest zero-wait-state RAM example

Figure 4-4 shows a single RAM device with a 32-bit data width connected directly to the TCM interface. The **DTCMWAIT** signal must be tied off to zero.

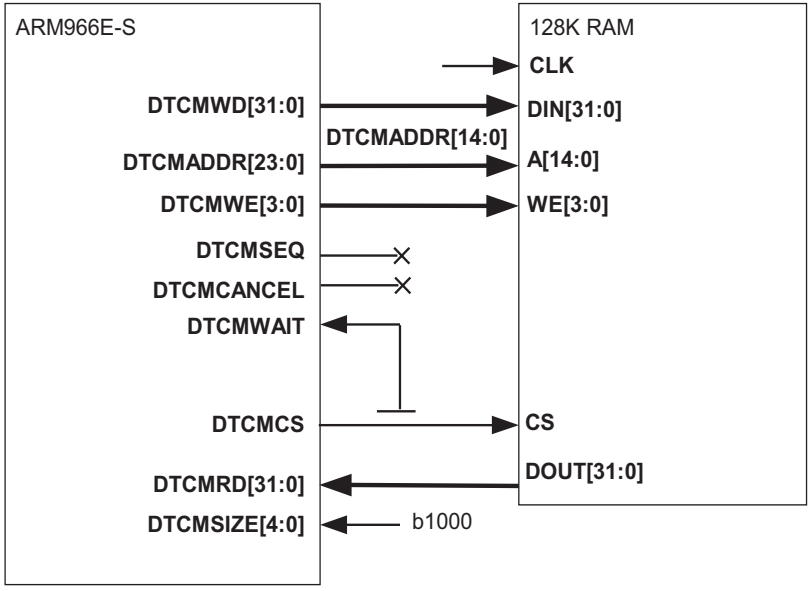


Figure 4-4 Simplest zero-wait-state RAM example

Inverters must be used if there are any polarity differences between the RAM input signals and those of the TCM interface. When the RAM chip select is active-LOW, an inverter must be placed between **DTCMCS** and the RAM chip select. This integration places a limit on the size of the TCMs. Multiple banks of RAM can be used to overcome this limitation, see *Multiple banks of RAM example* on page 4-17.

4.8.2 Byte-banks of RAM example

If byte-write RAM is not available, you can use four banks of 8-bit wide RAM by routing each of the four bits of **DTCMWE** to one of the four RAM write enable inputs, as shown in Figure 4-5.

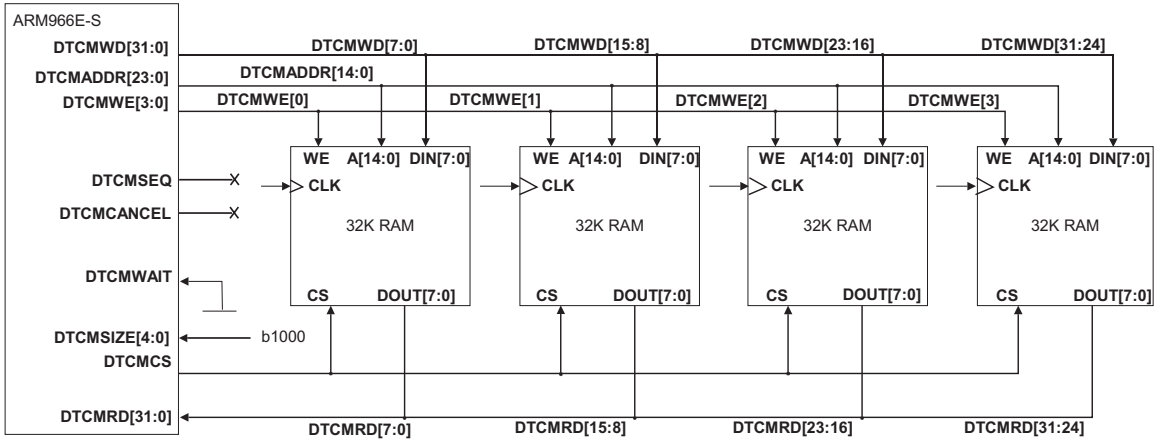


Figure 4-5 Byte-banks of RAM example

You can save a small amount of power by ANDing the chip select for each RAM device with (**DTCMWE** OR NOT **DTCMnRW**), so that byte and halfword writes generate requests only to the required byte RAM as shown in Figure 4-6 on page 4-17.

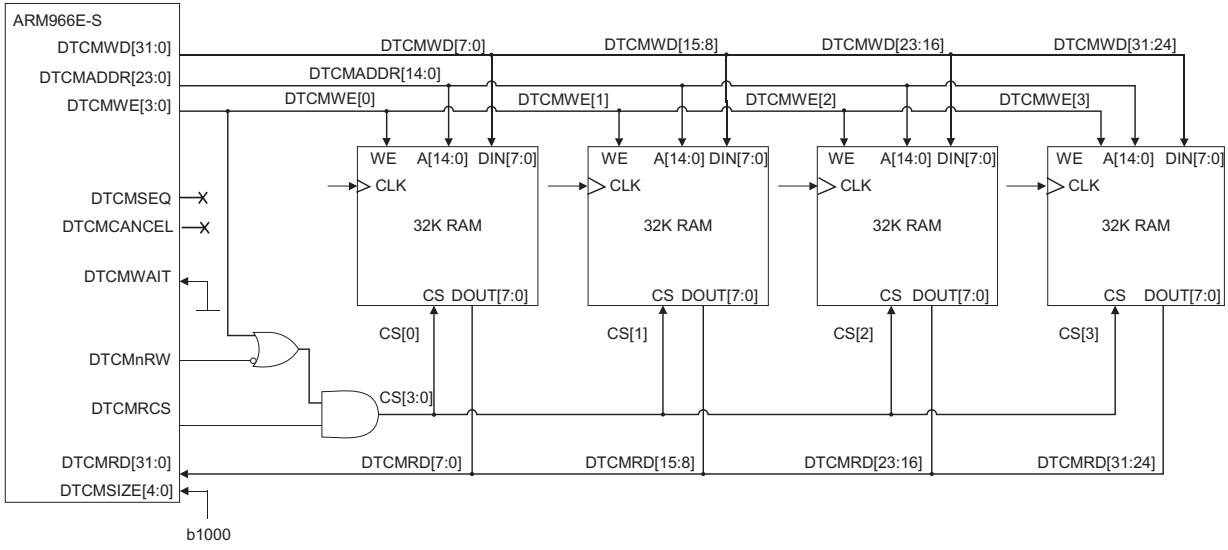


Figure 4-6 Byte-banks of RAM alternative example

4.8.3 Multiple banks of RAM example

If RAMs of sufficient size are not available, you can use multiple RAM devices. The read data can come from one of two or more devices that are selected using a multiplexor or multiplexors. To ensure that write data is not written to all devices, additional logic is required on either the chip select or the write enable.

Figure 4-7 on page 4-18 shows an example of multiple banked RAM with the chip select signal **DTCMCS** ANDed with the top address signal **DTCMADDR[14]**.

Figure 4-6 shows an example of multiple banked RAM with the write enable signals **DTCMWE[3:0]** ANDed with the top address signal **DTCMADDR[14]**.

Note

For the banked RAM example shown in Figure 4-7 on page 4-18, the reads and writes only occur on the RAM device required, giving a reduction in the speed and power used when compared to the example shown in Figure 4-6.

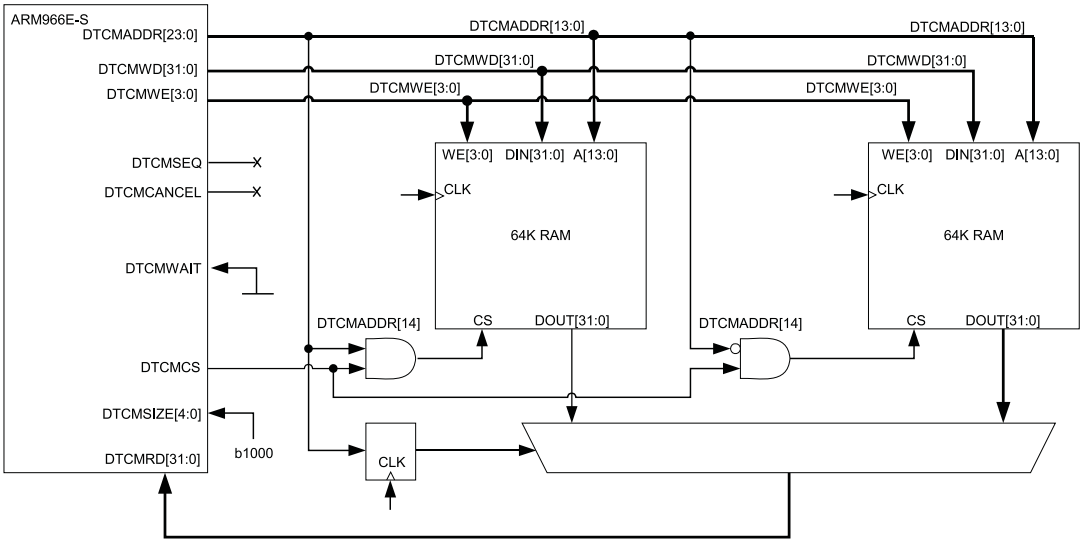


Figure 4-7 Multiple banks of RAM example

4.8.4 Sequential RAM example

If the RAM devices require a single wait-state except for sequential reads, the device can be connected as shown in Figure 4-8 on page 4-19. The **DTCMWAIT** signal is derived by using an inverter. The behavior of the **DTCMSEQ** signal is designed so that the correct wait signal behavior is produced.

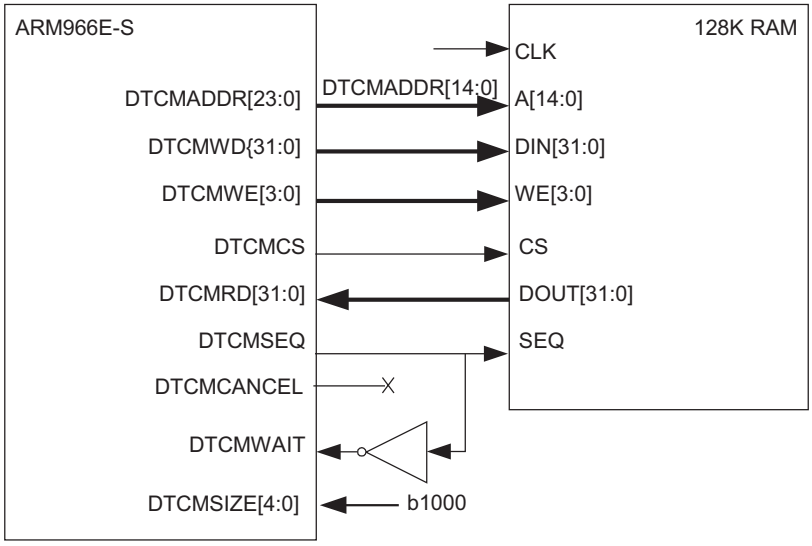


Figure 4-8 Sequential RAM example

4.8.5 Single or Multiple wait-state RAM example

If the RAM devices require more than one cycle for all accesses, logic is required to assert the wait signal for the required number of cycles. Figure 4-9 on page 4-20 shows a Wait State Controller asserting the wait signal as required. The power control block removes power to the TCM when it is not required.

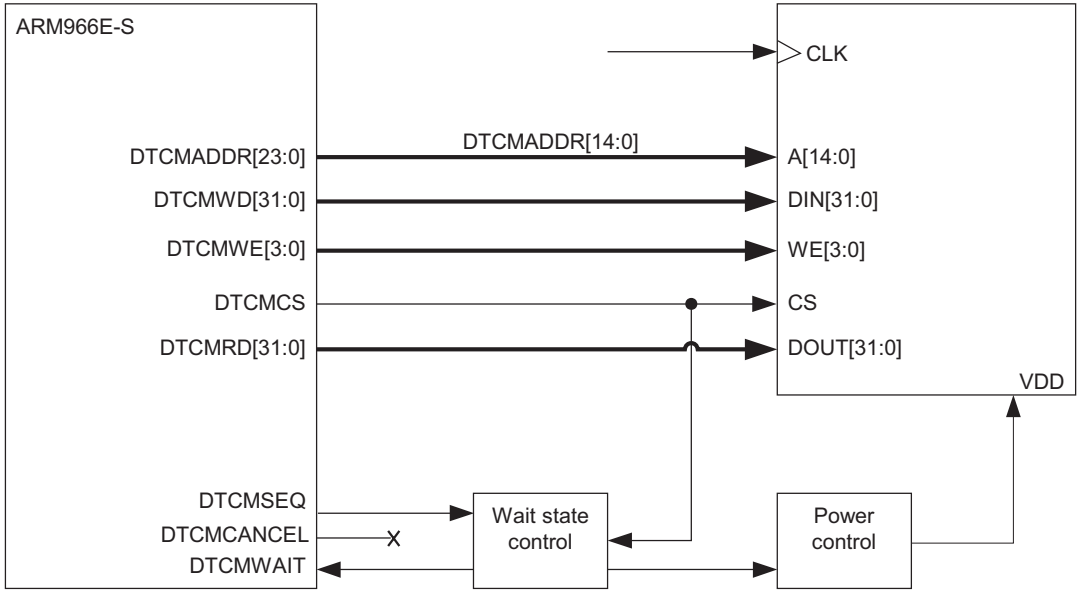


Figure 4-9 Single or Multiple wait-state RAM example

4.8.6 Single port RAM example

When using single port RAM, multiplexors are required to switch between DMA controller and ARM966E-S memory interface signals. Figure 4-10 on page 4-21 shows a single port RAM example.

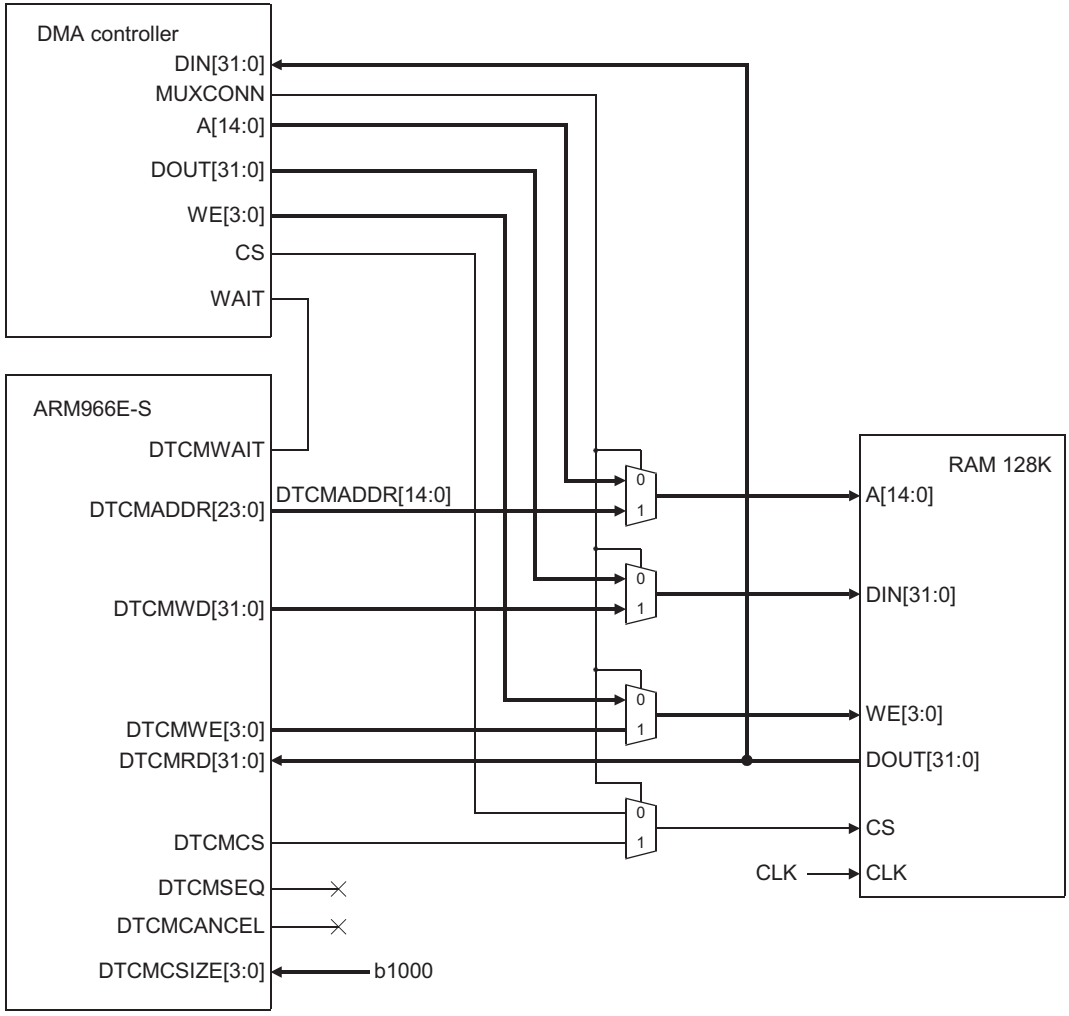


Figure 4-10 Single port DMA-capable RAM example

4.8.7 Dual port DMA-capable RAM example

For dual port RAM, the TCM interface must be attached to one port and the DMA controller to the other port as shown in Figure 4-11 on page 4-22. This requires minimal extra logic (more logic might be required to detect address conflicts, depending on the devices and the application).

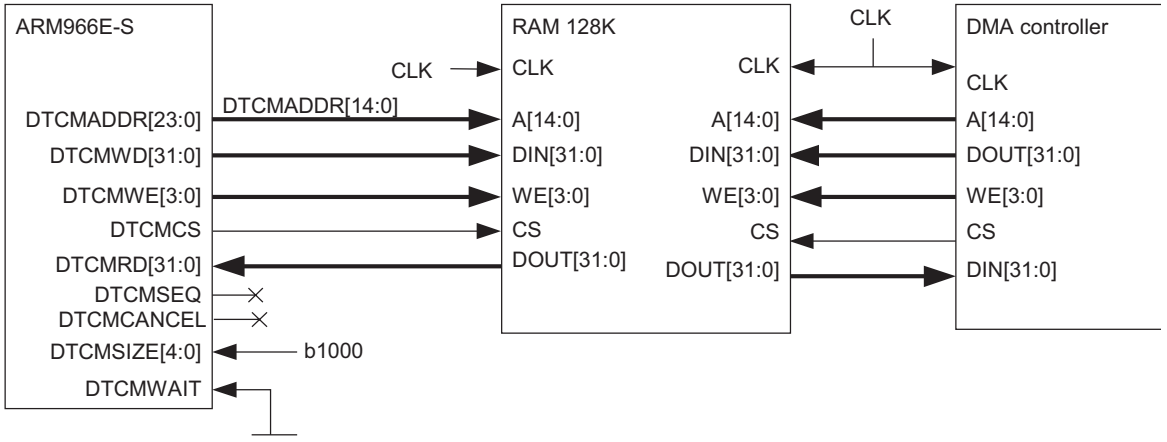


Figure 4-11 Dual port RAM example

Chapter 5
Bus Interface Unit

This chapter describes the ARM966E-S *Bus Interface Unit* (BIU) and AHB write buffer. It contains the following sections:

5.1 About the BIU

The ARM966E-S processor uses an *Advanced Microprocessor Bus Architecture* (AMBA) *Advanced High-performance Bus* (AHB) interface. The AHB is a new generation of AMBA interface that addresses the requirements of synthesizable high-performance designs, including:

- single rising-clock-edge operation
- unidirectional buses
- mapped burst transfers
- split transactions
- single-cycle bus master handover.

See the *AMBA Specification (Rev 2.0)* for full details of this bus architecture.

The ARM966E-S BIU implements a fully-compliant AHB bus master interface and incorporates an instruction prefetch buffer and a write buffer to increase system performance. The BIU is the link between the ARM9E-S core with its TCMs and the external AHB memory. The AHB memory must be used to initialize the TCMs and to access code and data that are not assigned to the TCM address space.

5.2 AHB instruction prefetch buffer

The BIU *Instruction Prefetch Buffer* (IPB) is four 32-bit entries deep. All nonsequential instruction fetches to AHB space cause the IPB to be flushed and an initial burst of four words to be performed on the AHB. After this initial burst, the IPB performs AHB accesses to keep the buffer full. If the ARM9E-S core takes an instruction out of the buffer on each clock cycle, the fetches on the AHB interface are performed as incrementing bursts of unspecified length (**HBURST[2:0]** = 001).

Only valid instruction requests initiates prefetching. The prefetch buffer marks each entry with the error response returned from the AHB. The prefetch buffer also marks each entry with the external breakpoint request returned from the external memory system. Instruction prefetching does not cross 1KB boundaries.

5.2.1 Optimized Thumb instruction prefetch

In Thumb state, the prefetch buffer depth is reduced to two words (four Thumb instructions). The ARM966E-S processor performs a two-word incrementing burst for nonsequential fetches. When space becomes available the IPB performs transfers to fill any vacant entries up to the buffer depth limit of two word entries available in Thumb state.

5.2.2 IPB disable bit

Setting bit [16] of the CP15 c15 Configuration Control Register disables the IPB. Actual prefetch behavior does not change until the next nonsequential instruction fetch occurs. See *CP15 c15, Test and Configuration Register* on page 2-11. Reset clears bit [16] and enables prefetching.

5.2.3 AHB error response with IPB

If an error response is returned from the AHB, it is stored in the IPB along with the instruction. If the instruction reaches the Execute stage of the ARM9E pipeline, a Prefetch Abort exception occurs.

5.2.4 IPB timing examples

This section gives two examples of IPB operation:

- *Nonsequential instruction fetch* on page 5-4
- *Nonsequential instruction fetch after a data access* on page 5-5.

Nonsequential instruction fetch

Figure 5-1 shows AHB prefetching in operation. In this case, the ARM9E-S core is executing code sequentially from the AHB. As soon as each instruction is returned from the AHB, it is returned to the ARM9E-S core. None of the returned data is placed into the prefetch buffer because the processor is continuously requesting instructions from the AHB. The AHB runs ahead of the core to minimize the number of stall cycles.

The ARM9E-S core generates a nonsequential instruction fetch. This can be a result of a branch or an operation changing the value of the PC, for example. The BIU terminates the current burst and starts a new prefetch operation with a burst of length four to fill the prefetch buffer. The first instruction from the burst is returned to the ARM9E-S core. The BIU keeps the prefetch buffer full by performing an undefined length burst. This is because the ARM9E-S core is running sequentially and requesting instructions each cycle. Because this is a new burst, AHB indicates NSEQ.

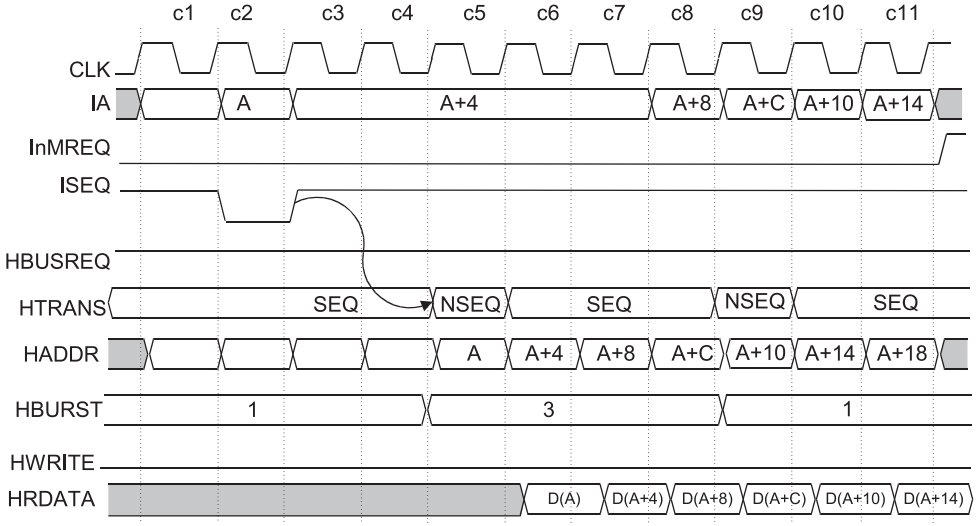


Figure 5-1 Nonsequential instruction fetch

Note

All timing examples in this chapter are based on one-to-one clocking in which the ARM966E-S processor and AHB share the same clock. See *AHB clocking* on page 5-15 for details of AHB clocking modes. All timing examples assume that bus mastership has been granted.

Nonsequential instruction fetch after a data access

Figure 5-2 shows an AHB data access between instruction fetches. Because data accesses take precedence over instruction fetches, the instruction fetch starts after the data access. After the first instruction address is issued on the AHB, sequential instruction prefetching starts. The core does not advance until both of the simultaneous memory requests are satisfied.

As shown in the previous example in Figure 5-1 on page 5-4, the prefetch buffer is not used immediately because each instruction returned from the BIU is immediately used by the processor. The second data memory request causes the processor to stall until the data request is completed. This causes the two outstanding instruction prefetches to be stored in the prefetch buffer. Prefetching stops as a result of data request.

The instruction request issued with the data access can be acknowledged as soon as the AHB transfer is complete. After the data access, prefetching can continue because the address is sequential to the previous instruction address.

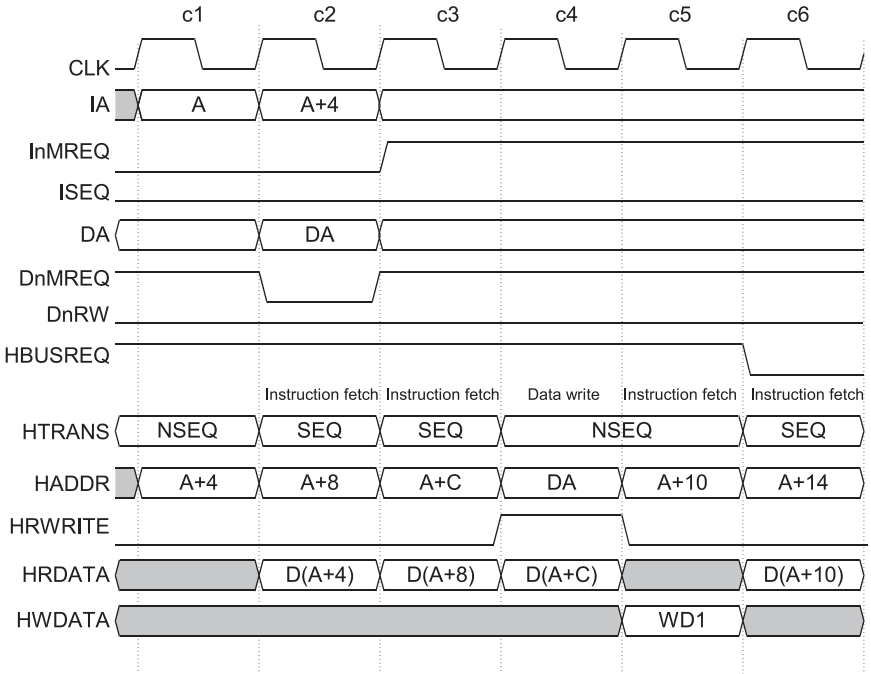


Figure 5-2 Nonsequential instruction fetch after a data access

5.3 AHB write buffer

The AHB write buffer can hold a four-entry address and eight data word entry. The write buffer decouples the core from the wait cycles caused by accessing the AHB. If a write is sent to the write buffer, the core is able to continue program execution without having to wait for the write to complete on the AHB. More writes can be committed to the write buffer without stalling if spare entries are available. If the ARM9E-S core tries to write to a buffered location when the write buffer is full, the core stalls until there is space in the write buffer.

If the ARM9E-S core performs a read from AHB address space or an unbuffered write to AHB address space, the core stalls until all write buffer entries are written. Draining the write buffer ensures data coherency.

5.3.1 Committing write data to the AHB write buffer

The AHB write buffer is used when the following conditions are met:

- the write buffer is enabled
- the write address is in a bufferable region
- the write address is in AHB address space
- the write address selects a TCM that is disabled.

For details on AHB write buffer enable and the ARM966E-S fixed address map, see

- *CP15 c1, Control Register* on page 2-7
- *About the ARM966E-S memory map* on page 3-2.

When the core performs a write that conforms to these conditions, the address for the write is put into the first available address entry of the write buffer FIFO. The next available entry data is used for the write data. If the write is a store multiple (STM), subsequent data entries are used for each word of the STM. It is therefore possible for the FIFO to contain eight words of an STM.

Alternatively, if several shorter bufferable STM or single writes (STR) instructions are performed, one address entry is used for each write instruction. The worst case is that only four data words fill the FIFO caused by four STR writes. In this case, the FIFO holds four address entries and four data entries.

5.3.2 Draining write data from the AHB write buffer

The AHB write buffer can drain naturally when AHB writes occur each time data is committed to the FIFO. The core stalls only if the write buffer overflows. However, there are times when a complete drain of the write buffer is enforced.

Natural AHB write buffer drain

When a write is being committed to the AHB write buffer, a signal to the BIU initiates an AHB write. The BIU then pops the address for the write from the write buffer followed by the data and starts an AHB transfer. This process might take several cycles because the write access is to an AHB slave in a bufferable region that has a multicycle response. Additionally, if the AHB is running at a lower rate than the core, there can be extra delay in the buffered write process. This can cause the core to fill the write buffer by committing data faster than the write buffer can drain. The ARM9E-S core stalls until an entry becomes available.

When an address is placed in the AHB write buffer, a marker is also stored to indicate if the size of the write is byte, halfword, or word. If an STM is performed, a sequentiality marker is stored with the data, to indicate to the BIU that the address incrementer must be used to produce the AHB address for the second and following writes of the STM. This mechanism allows only one FIFO entry to be used for the address, leaving more room for data.

Enforced AHB write buffer drain

There are two situations in which the core stalls and the AHB write buffer is forced to drain completely before program execution can continue:

- the core requests an instruction fetch, data load, or unbuffered AHB write
- the core performs a drain-write-buffer operation.

AHB read access requested

To ensure data coherency, the core must be prevented from reading data from a location when new data for that location is still in the AHB write buffer. If the read occurs before the write buffer is drained, the core reads the old data, causing a data coherency failure.

For this reason, whenever an AHB read, an instruction fetch, a data load, or a load multiple is requested, the core must be stalled until the write buffer is drained. There is no dedicated logic to initiate a write buffer drain because this process is occurring whenever data is present within the buffer. However, there is dedicated logic that stalls the core until the last buffered write is completed on the AHB.

Drain write buffer instruction

You can use an MCR instruction to CP15 c7 to stall the core until the AHB write buffer is empty and the final write is completed on the AHB. This instruction is described in *CP15 c7, Core Control Register* on page 2-9. This instruction is useful when a write must be completed before program execution can continue.

5.3.3 Enabling the AHB write buffer

Setting bit [3] of the CP15 c1 Control Register enables the write buffer. When this bit is set, all writes to bufferable address locations use the write buffer. If a slave peripheral in a bufferable region returns an AHB Data Abort, the abort is ignored when the write buffer is enabled.

Note

For debugging purposes, you can disable the AHB write buffer to enable AHB Data Aborts to be returned from bufferable regions.

5.3.4 Disabling the AHB write buffer

When data is committed to the write buffer, it is always written to the AHB. Disabling the write buffer by clearing bit [3] of the CP15 c1 Control Register causes any existing data in the write buffer to be written. Performing the wait-for-interrupt operation also causes any data in the write buffer to be written.

To ensure that no more buffered writes occur following write buffer disable or a wait-for-interrupt instruction, the write buffer must first be drained with a drain write buffer command.

5.4 AHB bus master interface

The ARM966E-S processor implements a fully-compliant AHB bus master interface. See the *AMBA Specification (Rev 2.0)* for a detailed description of the AHB protocol.

5.4.1 Overview of AHB

The AHB architecture is based on separate cycles for address and data. The address and control values for an access are broadcast from the rising edge of **HCLK** in the cycle before the data is expected to be read or written. During this data cycle, the address and control values for the next cycle are driven out. This leads to a fully pipelined address architecture.

When an access is in its data cycle, a slave can wait the access by driving the **HREADY** response LOW. This has the effect of stretching the current data cycle and the pipelined address and control for the next access. This creates a system where all AHB masters and slaves sample **HREADY** on the rising edge of the **HCLK** to determine if an access is complete and a new address can be sampled or driven out.

5.5 AHB transfer descriptions

The ARM966E-S BIU performs a subset of the possible AHB bus transfers. This section describes the transfers that can be performed and some back-to-back transfer cases:

- *Back-to-back data transfers*
- *Data burst crossing a 1KB boundary* on page 5-12
- *SWP instruction* on page 5-13.

Note

Figure 5-1 on page 5-4 to Figure 5-6 on page 5-14 in this chapter are examples of specific situations only. You must not use these examples to derive general AHB protocol and timing requirements. For more information on AHB operation, see the *AMBA Specification* (Rev 2.0).

5.5.1 Back-to-back data transfers

Figure 5-3 on page 5-11 shows ARM966E-S bus activity when a sequence of STR instructions is executed with no AHB instruction fetches. The ARM9E-S core is executing instructions from the TCM space.

In cycle 2, the ARM9E-S core starts a nonsequential data write. A series of nonsequential and idle transfers is indicated for each access. The ARM9E-S core is re-enabled in cycle 10.

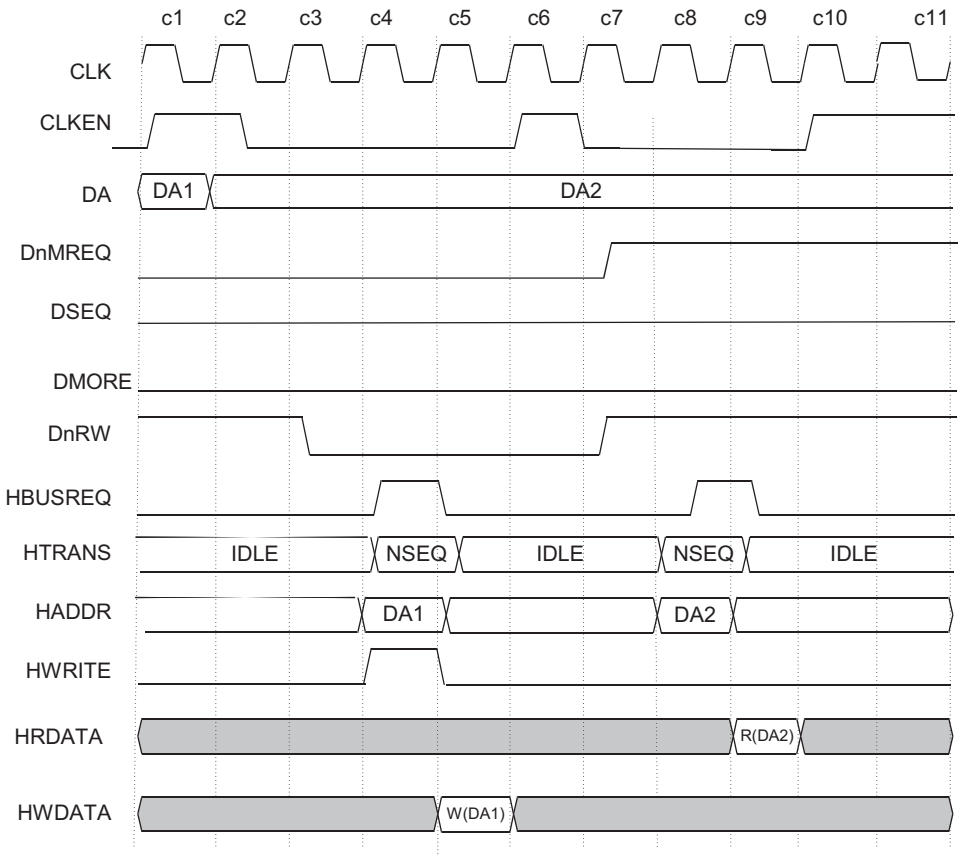


Figure 5-3 Back-to-back data transfer write followed by read

Note

Executing a sequence of back-to-back LDR instructions produces the same series of nonsequential and idle transfers.

STM followed by instruction fetch

Figure 5-4 on page 5-12 shows an example of an STM transferring four words, immediately followed by an instruction fetch. The instruction read begins with a nonsequential/sequential sequence after the final sequential data access. In this example, subsequent instruction fetches are sequential. Instruction prefetching is enabled so instruction fetches appear on the AHB before the ARM9E-S core requests them.

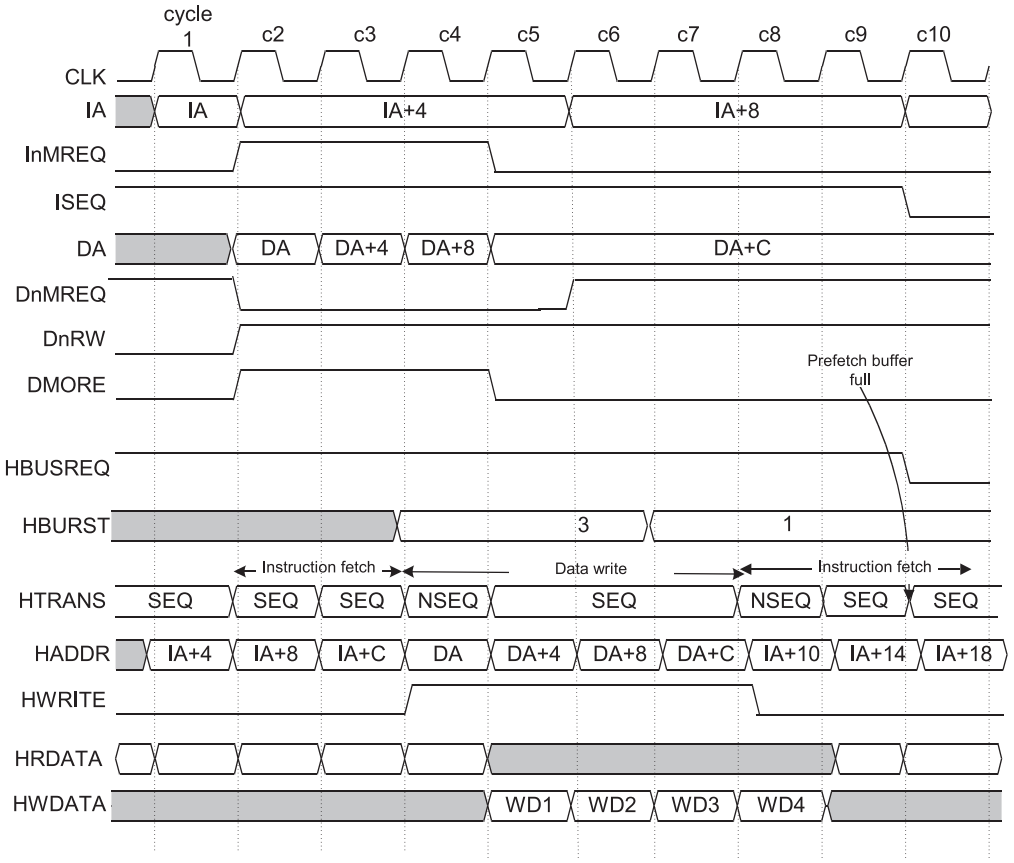


Figure 5-4 Single STM, followed by sequential instruction fetch

Data burst crossing a 1KB boundary

AMBA Specification (Rev 2.0) states that sequential accesses must not cross 1KB boundaries. The ARM966E-S processor splits sequential accesses that cross a 1KB boundary into two sets of separate accesses.

Figure 5-5 on page 5-13 shows bus activity with two back-to-back STM instructions crossing a 1KB boundary. DA+8 is the first address in a new 1KB region. The two sets of transfers each begin with a nonsequential access type, and are separated by idle cycles. In this example, instructions are being fetched from the ITCM.

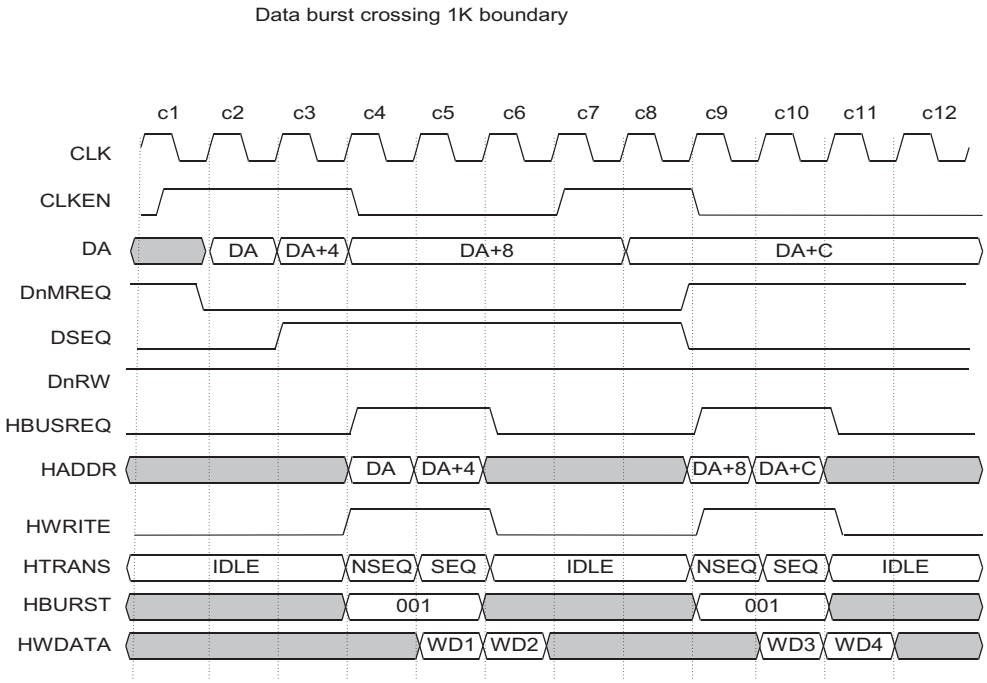


Figure 5-5 Data burst crossing a 1KB boundary

SWP instruction

The ARM SWP instruction performs an atomic read-modify-write operation. It is commonly used with semaphores to guarantee that another process cannot modify a semaphore when it is being read by the current process.

If the ARM966E-S processor performs a SWP operation to an AHB address location, the access is always unbuffered to ensure that the core stalls until the write occurs on the AHB. The BIU asserts the **HLOCK** output to prevent the AHB arbiter from granting a different master, ensuring that the read-modify-write is atomic. In this example, instructions are being fetched from the ITCM.

Figure 5-6 on page 5-14 shows a SWP instruction.

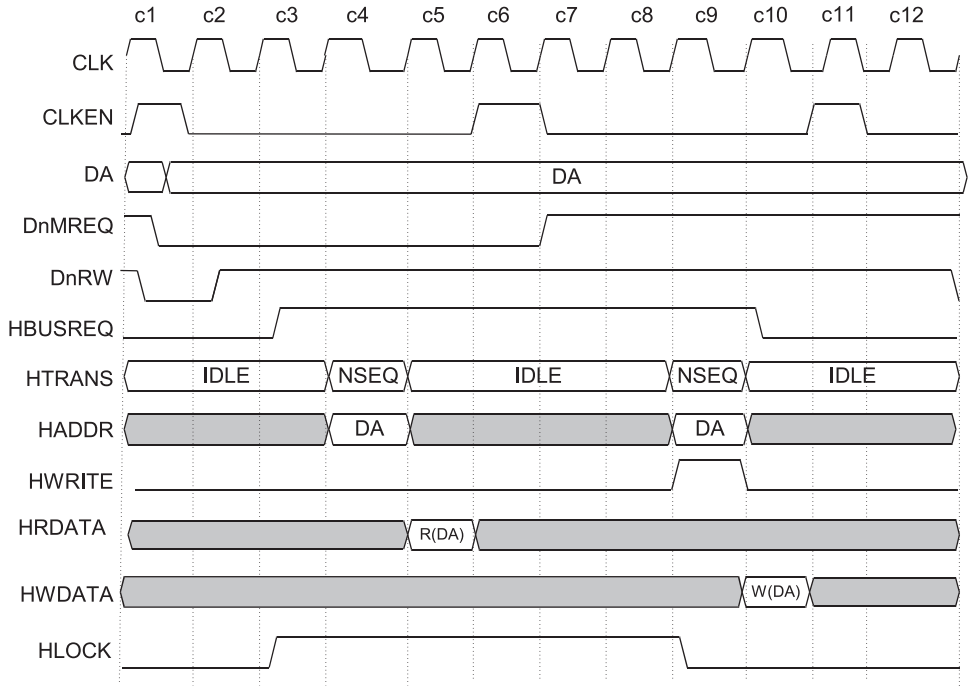


Figure 5-6 SWP instruction

5.5.2 Data burst support

To allow more efficient use of burst-capable memories on the AHB bus, the BIU detects core data burst sizes that align with AHB incrementing burst sizes. The ARM966E-S processor supports incrementing burst sizes of 4, 8, and 16 words. These bursts are then performed on the bus as defined length bursts. Bursts that cross 1K boundaries are split into 2 separate transactions on the AHB, one on each side of the boundary. All bursts that do not map onto AHB incrementing burst sizes are marked as unspecified length.

5.6 AHB clocking

The ARM966E-S design uses a single rising-edge clock, **CLK**, to time all internal activity. In systems where the ARM966E-S processor is embedded, it is best to run the AHB at a lower clock rate. To support this requirement, the ARM966E-S processor must have a clock enable, **HCLKEN**, to time AHB transfers.

The **HCLKEN** input is driven HIGH around a rising edge of the ARM966E-S processor **CLK** to indicate that this rising edge is also a rising edge of **HCLK**. **HCLK** must therefore be synchronous to **CLK**.

The **SYSCLKEN** is an internal signal connected to the **CLKEN** input of the ARM9E-S core, and is used to control the pipeline advance of the core. When **SYSCLKEN** is LOW, the core stalls.

When the ARM9E-S core is running from TCM or performing writes using the write buffer, the **HCLKEN** and **HREADY** inputs are decoupled from the **SYSCLKEN** stall signal. The core is only stalled by TCM stall cycles or if the write buffer overflows. This means that the ARM9E-S core is executing instructions at the faster **CLK** rate and is effectively decoupled from the **HCLK** domain AHB system.

If however, an AHB read access or unbuffered write is required, the core stalls until the AHB transfer is complete. Because the AHB system is being clocked by the slower **HCLK**, the core must examine **HCLKEN** to detect when to drive out the AHB address and control signals to start an AHB transfer. **HCLKEN** must detect the following rising edges of **HCLK** so that the BIU can detect when the access completes. Figure 5-7 shows an example of an AHB read access with a 3:1 ratio of **CLK** to **HCLK**.

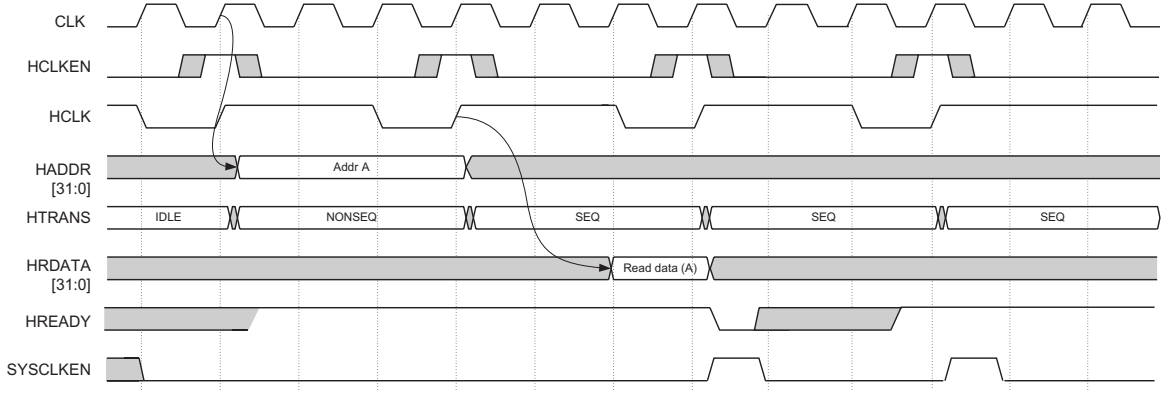


Figure 5-7 AHB 3:1 clocking example

If the slave being accessed at the **HCLK** rate has a multicycle response, the **HREADY** input to the ARM966E-S processor is driven LOW until the data is ready to be returned. The BIU must therefore perform a logical AND on the **HREADY** response with **HCLKEN** to detect that the AHB transfer is complete. When the AND is true, the ARM9E-S core is then enabled by reasserting **SYSCLKEN**.

Note

Before the core can start an AHB access, it must wait until it receives the next **HCLKEN** pulse. Then it must wait until the access is complete. The stall before the start of the access is a synchronization penalty and the worst case can be expressed in **CLK** cycles as the **CLK**-to-**CLK** ratio minus one.

5.7 CLK-to-HCLK skew

The ARM966E-S processor drives out the AHB address on the rising edge of **CLK** when the **HCLKEN** input is true. The AHB outputs have output hold and delay values relative to **CLK**. However, these outputs are used in the AHB system where **HCLK** is used to time the transfers. Similarly, inputs to the ARM966E-S processor are timed relative to **HCLK** but are sampled within the ARM966E-S processor with **CLK**. Minimizing the skew between **HCLK** and **CLK** prevents hold time issues from **CLK** to **HCLK** on outputs and from **HCLK** to **CLK** on inputs.

5.7.1 Clock tree insertion at top level

The ARM966E-S processor has a clock tree inserted to enable an evenly distributed clock to be driven to all the registers in the design. The registers that drive out AHB outputs and sample AHB inputs are timed off **CLK'** at the bottom of the inserted clock tree and subject to the clock tree insertion delay. To maximize performance, when the ARM966E-S processor is embedded in an AHB system, the clock generation logic to produce **HCLK** must be constrained so that it matches the insertion delay of the clock tree within the ARM966E-S processor. This can easily be achieved by performing a top-level clock tree insertion for the ARM966E-S processor and the embedded system at the same time.

Figure 5-8 shows an example of an AHB slave connected to the ARM966E-S processor.

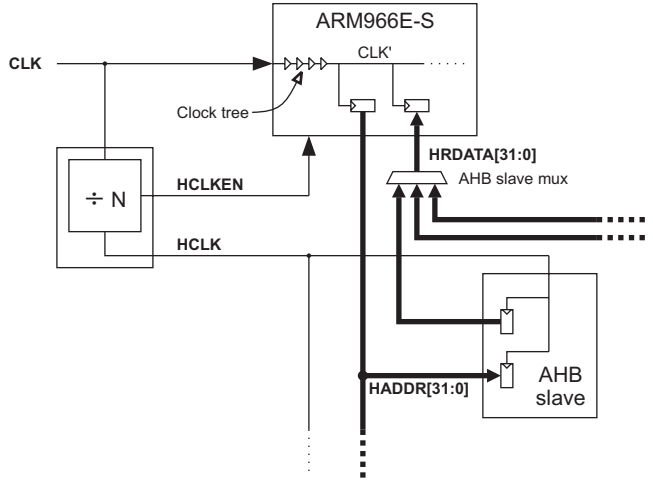


Figure 5-8 ARM966E-S CLK to AHB HCLK sampling

In this example, the slave peripheral has an input setup and hold time and an output hold and valid time relative to **HCLK**. The ARM966E-S processor has an input setup and hold time and an output hold and valid relative to **CLK'**, the clock at the bottom of the clock tree. For optimal performance, clock tree insertion must be used to balance the **HCLK** to match **CLK'**.

5.7.2 Hierarchical clock tree insertion

If clock tree insertion is performed before embedding the ARM966E-S processor, buffers are added on input data to match the clock tree so that the setup and hold is relative to the top level **CLK**. This is guaranteed to be safe at the expense of extra buffers in the data input path.

The **HCLK** domain AHB peripherals must still meet the ARM966E-S processor input setup and hold requirements. Because the ARM966E-S processor inputs and outputs are now relative to **CLK**, the outputs appear comparatively later by the value of the insertion delay. This ultimately leads to lower AHB performance.

Chapter 6
Coprocessor Interface

This chapter describes the ARM966E-S pipelined coprocessor interface. It contains the following sections:

6.1 About the coprocessor interface

The ARM966E-S processor fully supports the connection of on-chip coprocessors through the external coprocessor interface and supports all classes of coprocessor instructions.

The interface differs from the basic ARM9E-S coprocessor interface. To ease integration of an external coprocessor, the interface from the ARM966E-S processor to the coprocessor is pipelined by a single clock cycle as shown in Figure 6-1.

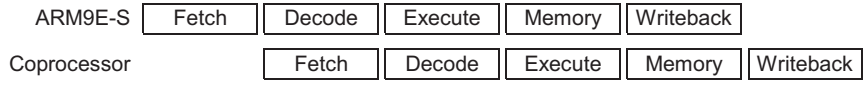


Figure 6-1 Pipeline stages

This ensures that ARM966E-S interface outputs, which otherwise arrive late in the clock cycle, are driven out directly from registers to the external coprocessor. This significantly eases the implementation task for an external coprocessor.

6.2 Coprocessor interface signals

Table 6-1 describes the ARM966E-S coprocessor interface signals.

Table 6-1 Coprocessor interface signals

Name	Direction with respect to ARM966E-S processor	Description
CPBURST[3:0]	Input	This signal indicates the number of words to be transferred as part of a burst from an external coprocessor. This signal allows a maximum of 16 words to be transferred. The values for the burst lengths are: b0000 = 1 word or undefined length b0001 = 2 words b0010 = 3 words b0011 = 4 words b0100 = 5 words b0101 = 6 words b0110 = 7 words b0111 = 8 words b1000 = 9 words b1001 = 10 words b1010 = 11 words b1011 = 12 words b1100 = 13 words b1101 = 14 words b1110 = 15 words b1111 = 16 words Tie off to zero if no external coprocessors are present.
CPCLKEN	Output	Synchronous enable for coprocessor pipeline follower. When HIGH on the rising edge of **CLK**, the pipeline follower logic is able to advance.
CPINSTR[31:0]	Output	The 32-bit coprocessor instruction bus over which instructions are transferred to the coprocessor pipeline follower.
CPDOUT[31:0]	Output	The 32-bit coprocessor read data bus for transferring data to the coprocessor.
CPDIN[31:0]	Input	The 32-bit coprocessor write data bus for transferring data from the coprocessor.
CPPASS	Output	Indicates that there is a coprocessor instruction in the Execute stage of the pipeline, and it must be executed.
CPLATECANCEL	Output	If HIGH during the first memory cycle of a coprocessor instruction, the coprocessor must cancel the instruction without changing any internal state. This signal is only asserted in cycles where the previous instruction caused a Data Abort to occur.
CHSDE[1:0]	Input	The handshake signals from the Decode stage of the coprocessor pipeline follower:ABSENT = 10 WAIT = 00 GO = 01 LAST = 11.
CHSEX[1:0]	Input	The handshake signals from the Execute stage of the coprocessor pipeline follower: ABSENT = 10 WAIT = 00 GO = 01 LAST = 11.

Table 6-1 Coprocessor interface signals (continued)

Name	Direction with respect to ARM966E-S processor	Description
CPTBIT	Output	When HIGH, indicates that the ARM966E-S processor is in Thumb state. When LOW, indicates that the ARM966E-S processor is in ARM state. Sampled by the coprocessor pipeline follower.
nCPMREQ	Output	When LOW on the rising edge of **CLK** and **CPCLKEN** is HIGH, the instruction on **CPINSTR** must enter the coprocessor pipeline.
nCPTRANS	Output	When LOW, indicates that the ARM966E-S processor is in User mode. When HIGH, indicates that the ARM966E-S processor is in Privileged mode. Sampled by the coprocessor pipeline follower.

6.2.1 Synchronizing the external coprocessor pipeline

A coprocessor connected to the ARM966E-S processor determines which instructions it needs to execute by implementing a pipeline follower in the coprocessor. Each instruction that enters the ARM9E-S pipeline also enters the coprocessor pipeline one clock cycle later. The interface to the coprocessor is pipelined and so the coprocessor pipeline follower operates one cycle behind the ARM9E-S core, sampling the **CPINSTR[31:0]** output bus from the ARM966E-S coprocessor interface.

To hide the pipeline delay, a mechanism inside the interface block stalls the ARM9E-S core for a cycle by internally modifying the coprocessor handshake signals whenever an external coprocessor instruction is decoded. This enables the external coprocessor to catch up with the ARM9E-S core.

After this initial stall cycle, the two pipelines can be considered synchronized. The ARM9E-S core then informs the coprocessor when instructions move from Decode into Execute, and whether the instruction has passed its condition codes and is to be executed.

Note

Because the ARM966E-S processor hides the synchronization of the coprocessor pipeline follower, its coprocessor handshake interface is similar to that of the native ARM9E-S core. This implies that an ARM9E-S core designed pipeline follower can interface to the ARM966E-S processor without modification. The data path of the coprocessor differs however, due to the ARM966E-S pipelined output data **CPDOUT[31:0]**.

6.2.2 External coprocessor clocking

The CDP instruction is used for coprocessor instructions that do not operate on values in ARM registers or in main memory. One example is a floating-point multiply instruction for a floating-point accelerator processor.

To enable coprocessors to continue execution of CDP instructions while the ARM9E-S core pipeline stalls, for instance while waiting for an AHB transfer to complete, the coprocessor receives the free-running system clock **CLK**, and a clock enable signal **CPCLKEN**. If **CPCLKEN** is LOW around the rising edge of **CLK** then the ARM9E-S core pipeline stalls and the coprocessor pipeline follower must not advance.

This prevents any new instructions entering Execute within the coprocessor but enables a CDP instruction in Execute to continue execution. The coprocessor only stalls when the current instruction leaves Execute and new instructions are required from the ARM966E-S interface. This goes some way towards decoupling the external coprocessor from the ARM9E-S memory interface.

There are three classes of coprocessor instructions:

- LDC/STC
- MCR/MRC
- CDP.

Examples of how a coprocessor executes these instruction classes are given in the following sections:

- *LDC/STC* on page 6-9
- *MCR/MRC* on page 6-11
- *CDP* on page 6-13.

6.2.3 Coprocessor handshake states

The handshake signals encode one of four states:

ABSENT If there is no coprocessor attached that can execute the coprocessor instruction, the handshake signals indicate the ABSENT state. In this case, the ARM9E-S core takes the undefined instruction trap.

WAIT If there is a coprocessor attached that can handle the instruction, but not immediately, the coprocessor handshake signals are driven to indicate that the ARM9E-S processor core must stall until the coprocessor can catch up. This is known as the *busy-wait* condition. In this case, the ARM9E-S processor core loops in an IDLE state waiting for **CHSEX[1:0]** to be driven to another state, or for an interrupt to occur. If **CHSEX[1:0]** changes to ABSENT, the undefined instruction trap is taken. If **CHSEX[1:0]** changes to GO or LAST, the instruction proceeds

as described here. If an interrupt occurs, the ARM9E-S processor is forced out of the busy-wait state. This is indicated to the coprocessor by the **CPPASS** signal going LOW. The instruction is restarted later and so the coprocessor must not commit to the instruction (it must not change any coprocessor state) until **CPPASS** is asserted HIGH, when the handshake signals indicate the GO or LAST condition.

GO The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires at least another cycle of execution. Both the ARM9E-S processor core and the coprocessor must also consider the state of the **CPPASS** signal before actually committing to the instruction. For an LDC or STC instruction, the coprocessor instruction drives the handshake signals with GO when two or more words must still be transferred. When only one more word must be transferred, the coprocessor drives the handshake signals with LAST. During the Execute stage, the ARM9E-S processor core outputs the address for the LDC/STC. Also in this cycle, **DnMREQ** is driven LOW, indicating to the ARM966E-S memory system that a memory access is required at the data end of the device. The timing for the data on **CPDOUT** and **CPDIN** is shown in Figure 6-4 on page 6-9.

LAST An LDC or STC can be used for more than one item of data. If this is the case, possibly after busy waiting, the coprocessor drives the coprocessor handshake signals with a number of GO states, and in the next to the last cycle (LAST indicating that the next transfer is the final one). If there is only one transfer, the sequence is [WAIT,[WAIT,...]],LAST. LAST is also usually driven for CDP instructions.

6.2.4 Coprocessor handshake encoding

Table 6-2 shows how the handshake signals **CHSDE[1:0]** and **CHSEX[1:0]** are encoded.

Table 6-2 Handshake encoding

[1:0]	Meaning
10	ABSENT
00	WAIT
01	GO
11	LAST

Note

If the ARM966E-S processor does not have an external coprocessor, the **CHSDE[1:0]** and **CHSEX[1:0]** handshake inputs must be tied off to indicate ABSENT.

6.2.5 Multiple external coprocessors

If the ARM966E-S processor has multiple external coprocessors, the handshaking signals can be combined by ANDing bit [1], and ORing bit [0].

In the case of two coprocessors that have handshaking signals **CHSDE1**, **CHSEX1** and **CHSDE2**, **CHSEX2** respectively use:

CHSDE[1] = **CHSDE1[1]** AND **CHSDE2[1]**

CHSDE[0] = **CHSDE1[0]** OR **CHSDE2[0]**

CHSEX[1] = **CHSEX1[1]** AND **CHSEX2[1]**

CHSEX[0] = **CHSEX1[0]** OR **CHSEX2[0]**.

For **CPDIN[31:0]** use:

CPDIN[31:0] = **CPDIN1[31:0]** OR **CPDIN2[31:0].**

For **CPBURST[3:0]** use:

CPBURST[3:0] = **CPBURST1[3:0]** OR **CPBURST2[3:0]**.

6.2.6 Multiple external coprocessor example

Figure 6-2 on page 6-8 shows an example where VFP9 and two other coprocessors are connected to the ARM966E-S processor using the coprocessor interface logic block.

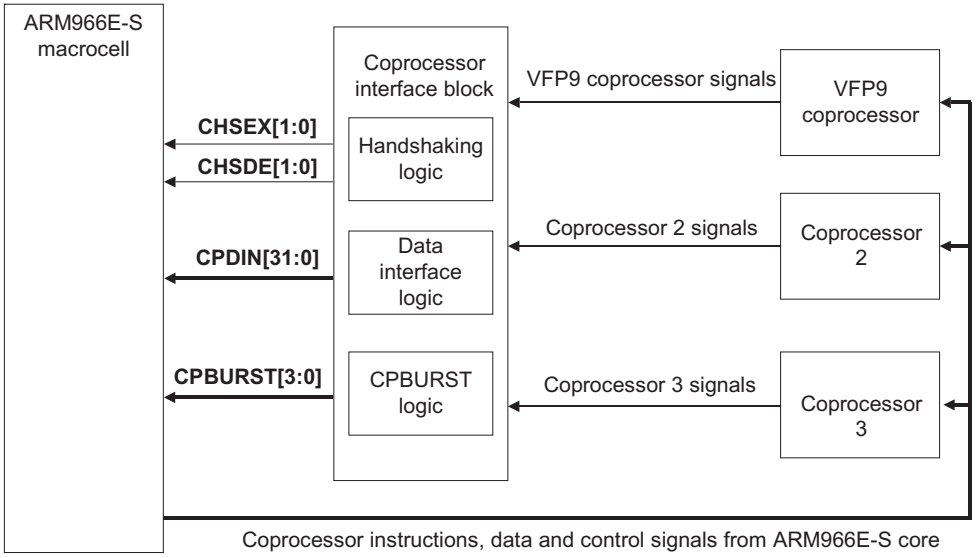


Figure 6-2 Connecting multiple coprocessors

Handshaking logic block

Figure 6-3 shows example components of the handshaking logic in the coprocessor interface logic block.

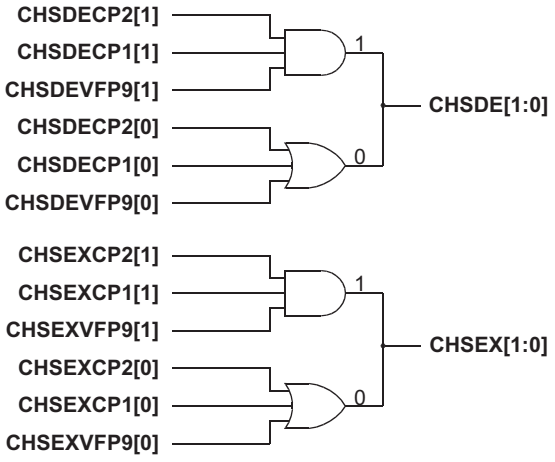


Figure 6-3 Example handshake logic blocks

6.3 LDC/STC

The LDC and STC instructions are used respectively to transfer data to and from external coprocessor registers and memory. In the case of the ARM966E-S processor, the memory can be either tightly-coupled memory or AHB depending on the address range of the access and TCM enable.

The cycle timing for these operations is shown in Figure 6-4.

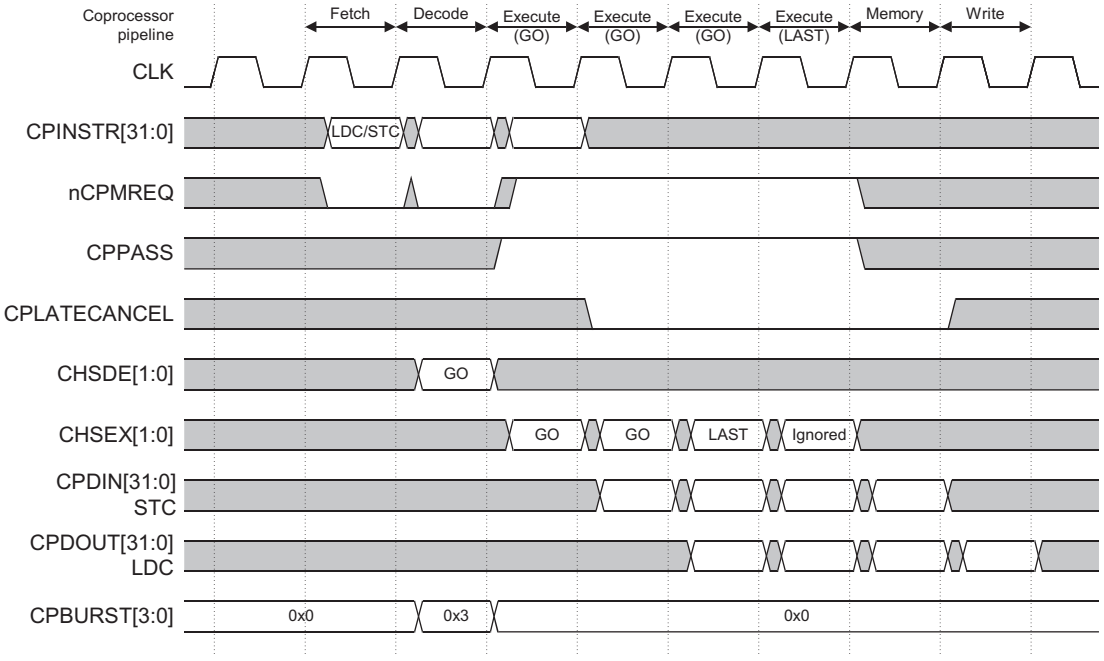


Figure 6-4 LDC/STC cycle timing

In this example, four words of data are transferred. The number of words transferred is determined by how the coprocessor drives the **CHSDE[1:0]** and **CHSEX[1:0]** buses.

As with all other instructions, the ARM9E-S core performs the main decode off the rising edge of the clock during the Decode stage. From this, the core commits to executing the instruction and so performs an instruction fetch. The coprocessor instruction pipeline keeps in step with the ARM9E-S core by monitoring **nCPMREQ**. **nCPMREQ** is a registered version of the ARM9E-S core instruction memory request signal **InMREQ**.

At the rising edge of **CLK**, if **CPCLKEN** is HIGH, and **nCPMREQ** is LOW, an instruction fetch takes place, and **CPINSTR[31:0]** contains the fetched instruction on the next rising edge of the clock, when **CPCLKEN** is HIGH.

This means that:

- the last instruction fetched must enter the Decode stage of the coprocessor pipeline
- the instruction in the Decode stage of the coprocessor pipeline must enter its Execute stage
- the fetched instruction must be sampled.

In all other cases, the ARM9E-S pipeline stalls, and the coprocessor pipeline must not advance.

During the Execute stage, the condition codes are compared with the flags to determine if the instruction really executes. The output **CPPASS** is asserted HIGH if the instruction in the Execute stage of the coprocessor pipeline:

- is a coprocessor instruction
- has passed its condition codes.

If a coprocessor instruction is in a busy-wait state, **CPPASS** is asserted on every cycle until the coprocessor instruction is executed. If an interrupt occurs during busy-waiting, **CPPASS** is driven LOW, and the coprocessor stops execution of the coprocessor instruction.

Another output, **CPLATECANCEL**, cancels a coprocessor instruction when the instruction preceding it caused a Data Abort. This is valid on the rising edge of **CLK** on the cycle that follows the first Execute cycle of the coprocessor instructions. This is the only cycle in which **CPLATECANCEL** can be asserted.

On the rising edge of the clock, the ARM9E-S processor examines the coprocessor handshake signals **CHSDE[1:0]** or **CHSEX[1:0]**:

- if a new instruction enters the Execute stage in the next cycle, it examines **CHSDE[1:0]**.
- if the currently executing coprocessor instruction requires another Execute cycle, it examines **CHSEX[1:0]**.

6.4 MCR/MRC

The MCR and MRC cycles look very similar to the STC and LDC cycles. An example, with a busy-wait state is shown in Figure 6-5. First, **nCPMREQ** is driven LOW to denote that the instruction on **CPINSTR[31:0]** is entering the Decode stage of the pipeline. This causes the coprocessor to decode the new instruction and drive **CHSDE[1:0]**. In the next cycle, **nCPMREQ** is driven LOW to denote that the instruction is now issued to the Execute stage. If the condition codes passes, and the instruction is to be executed, the **CPPASS** signal is driven HIGH and the **CHSDE[1:0]** handshake bus is examined (it is ignored in all other cases).

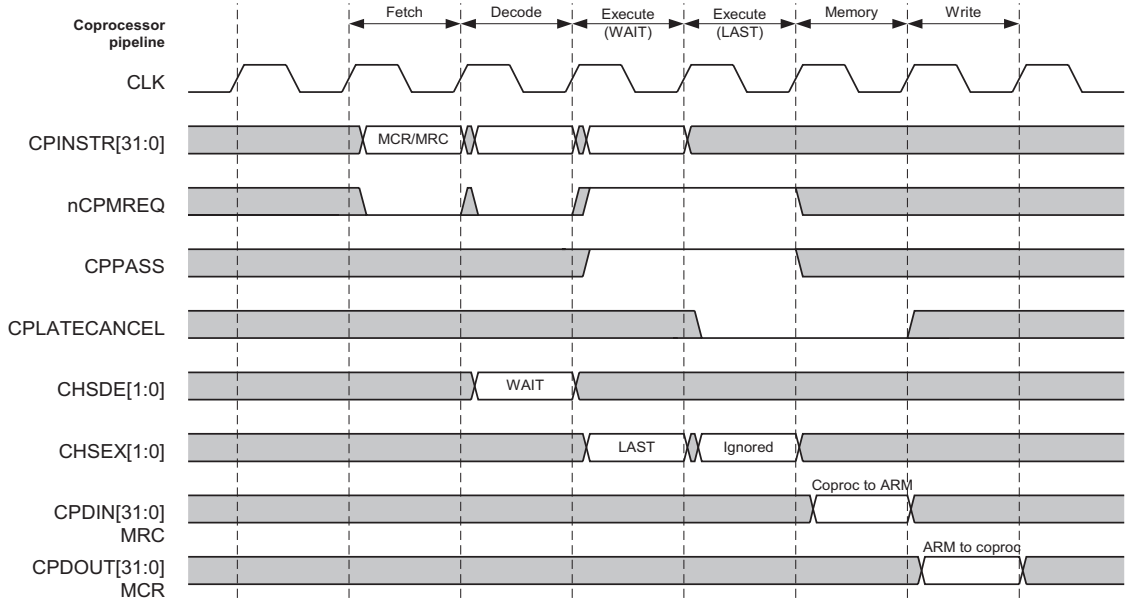


Figure 6-5 MCR/MRC transfer timing with busy-wait

For any successive Execute cycle, the **CHSEX[1:0]** handshake bus is examined. When the LAST condition is observed, the instruction is committed. For an MCR instruction, the **CPDOUT[31:0]** bus is driven with the registered data. For an MRC instruction, **CPDIN[31:0]** is sampled at the end of the ARM9E-S core Memory stage and written to the destination register during the next cycle.

6.5 Interlocked MCR

If the data for an MCR operation is not available inside the ARM9E-S core pipeline during its first Decode cycle, then the ARM9E-S core pipeline interlocks for one or more cycles until the data is available. An example of this is where the register being transferred is the destination of a preceding LDR instruction.

In this situation, the MCR instruction enters the Decode stage of the coprocessor pipeline, and then remains there for a number of cycles before entering the Execute stage. Figure 6-6 gives an example of an interlocked MCR that also has a busy-wait state.

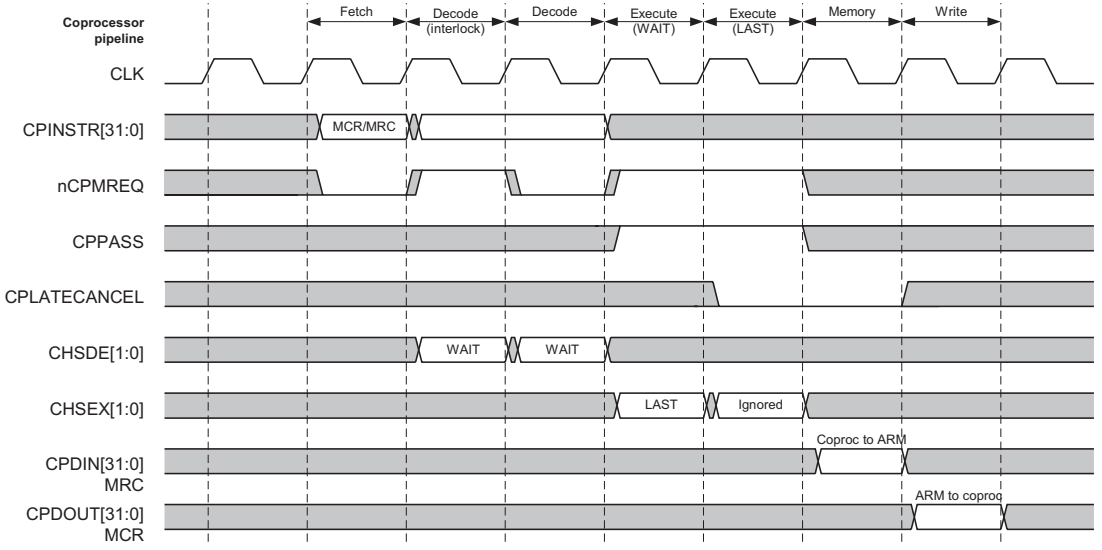


Figure 6-6 Interlocked MCR/MRC timing with busy-wait

6.6 CDP

CDP instructions normally execute in a single cycle. Like all the previous cycles, **nCPMREQ** is driven LOW to signal when an instruction is entering the Decode and then the Execute stage of the pipeline:

- if the instruction is to be executed, the **CPPASS** signal is driven HIGH during the Execute cycle
- if the coprocessor can execute the instruction immediately, it drives **CHSDE[1:0]** with LAST
- if the instruction requires a busy-wait cycle, the coprocessor drives **CHSDE[1:0]** with WAIT and then **CHSEX[1:0]** with LAST.

Figure 6-7 shows a canceled CDP due to the previous instruction causing a Data Abort.

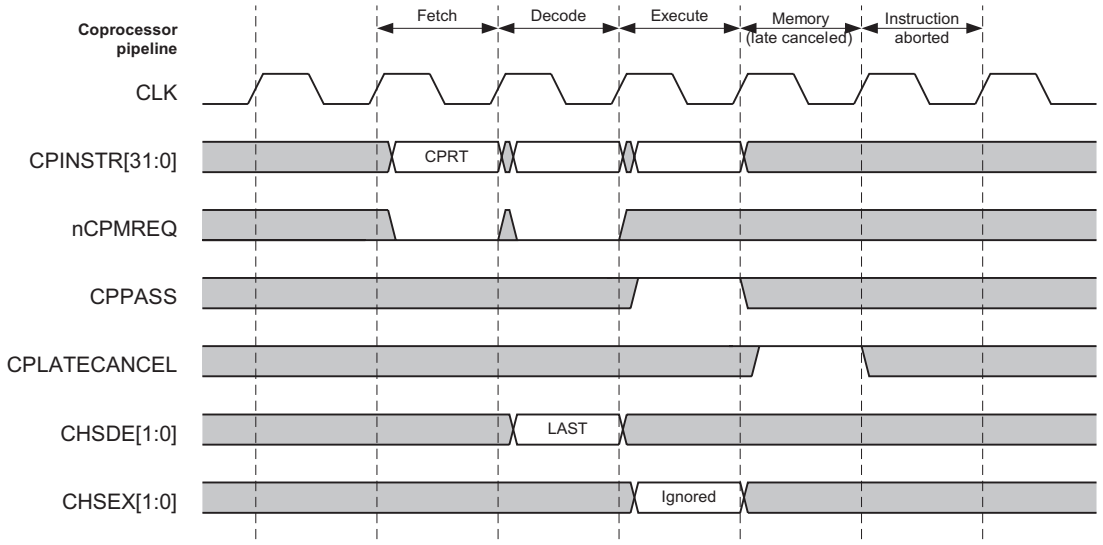


Figure 6-7 Late canceled CDP

The CDP instruction enters the Execute stage of the pipeline and is signaled to execute by **CPASS**. In the following cycle, **CPLATECANCEL** is asserted. This causes the coprocessor to terminate execution of the CDP instruction, resulting in no state changes to the coprocessor.

6.7 Privileged instructions

The coprocessor restricts certain instructions for use in Privileged modes only. To achieve this, the coprocessor tracks the **nCPTRANS** output. Figure 6-8 shows how **nCPTRANS** changes after a mode change.

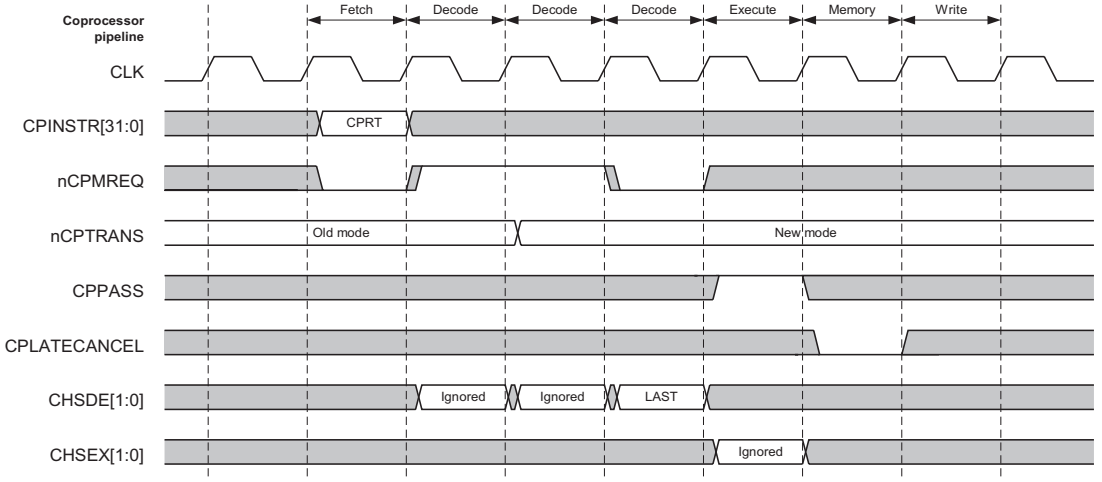


Figure 6-8 Privileged instructions

The ARM9E-S core ignores the first two **CHSDE[1:0]** responses because only the final **CHSDE[1:0]** response, as the instruction moves from Decode into Execute, counts. This enables the coprocessor to change its response when **nCPTRANS** changes.

6.8 Busy-waiting and interrupts

The coprocessor is permitted to stall, or busy-wait the processor during the execution of a coprocessor instruction if, for example, it is still busy with an earlier coprocessor instruction. To do so, the coprocessor associated with the Decode stage instruction drives WAIT onto **CHSDE[1:0]**. When the instruction concerned enters the Execute stage of the pipeline, the coprocessor drives WAIT onto **CHSEX[1:0]** for as many cycles as necessary to keep the instruction in the busy-wait loop.

For interrupt latency reasons, the coprocessor might be interrupted while busy-waiting, causing the instruction to be abandoned. Abandoning execution is done through **CPPASS**. The coprocessor must monitor the state of **CPPASS** during every busy-wait cycle.

If it is HIGH, the instruction must still be executed. If it is LOW, the instruction must be abandoned.

Figure 6-9 shows a busy-waited coprocessor instruction being abandoned due to an interrupt.

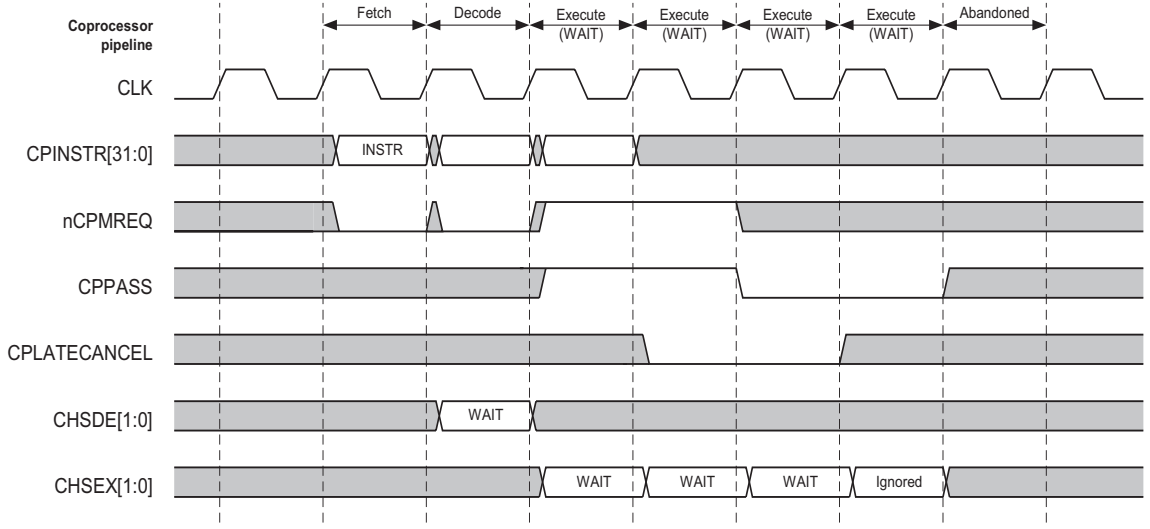


Figure 6-9 Busy-waiting and interrupts

Chapter 7
Debug Support

This chapter describes the ARM966E-S debug interface. It contains the following sections:

The ARM9E-S EmbeddedICE-RT logic is also discussed in this chapter including:

7.1 About the debug interface

The ARM966E-S debug interface is based on IEEE Std. 1149.1- 1990, Standard Test Access Port and Boundary-Scan Architecture. See this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

The ARM9E-S core within the ARM966E-S processor contains hardware extensions for advanced debugging features. These make it easier to develop application software, operating systems, and the hardware itself.

The debug extensions enable you to force the core into debug state. In debug state, the core and ARM966E-S memory system are effectively stopped, and isolated from the rest of the system. This is known as halt mode operation. It enables you to examine the internal state of the ARM9E-S core, ARM966E-S processor, and external state of the AHB while all other system activity continues as normal. When debug is complete, the ARM9E-S core restores the system state and resumes program execution.

In addition, the ARM9E-S core supports a real-time debug mode that generates an internal Instruction Abort or Data Abort instead of a breakpoint or watchpoint. This is known as monitor mode operation.

When used in conjunction with a debug monitor program activated by the abort exception entry, you can debug the ARM966E-S processor while allowing the execution of critical interrupt service routines. The debug monitor program typically communicates with the debug host over the ARM966E-S debug communication channel. Monitor mode debug is described in *Monitor mode debug* on page 7-23.

7.1.1 Stages of debug

A request on one of the external debug interface signals, or on an internal functional unit known as the *EmbeddedICE-RT logic*, forces the ARM9E-S core into debug state. The interrupts that activate debug are:

- a breakpoint (a given instruction fetch)
- a watchpoint (a data access)
- an external debug request.

The internal state of the ARM9E-S core is examined using a JTAG-style serial interface, allowing instructions to be serially inserted into the core pipeline without using the external data bus. For example, when in debug state, a Store Multiple (STM) can be inserted into the instruction pipeline, and this exports the contents of the ARM9E-S registers. This data can be serially shifted out without affecting the rest of the system.

7.1.2 Clocks

The system and test clocks must be synchronized externally to the ARM966E-S macrocell. The Multi-ICE debug agent directly supports one or more cores within an ASIC design. To synchronize off-chip debug clocking with the ARM966E-S macrocell requires a three-stage synchronizer. The off-chip device (for example, Multi-ICE) issues a **TCK** signal, and waits for the **RTCK** (**R**eturned **TCK**) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next **TCK** until after **RTCK** is received.

Figure 7-1 shows this synchronization.

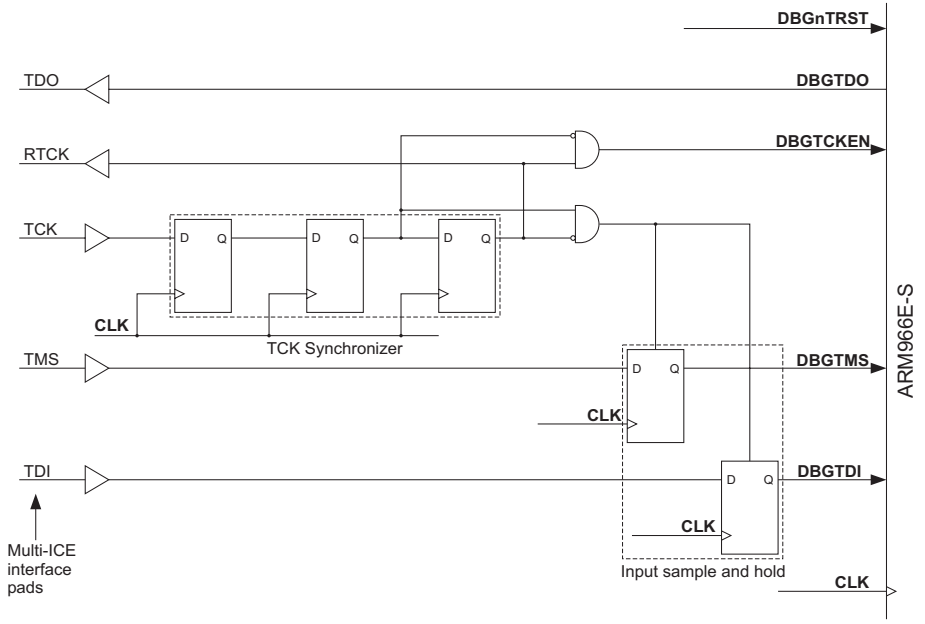


Figure 7-1 Clock synchronization

7.2 Debug systems

The ARM966E-S core forms one component of a debug system that interfaces from the high-level debugging performed by you to the low-level interface supported by the ARM966E-S processor. Figure 7-2 shows a typical debug system.

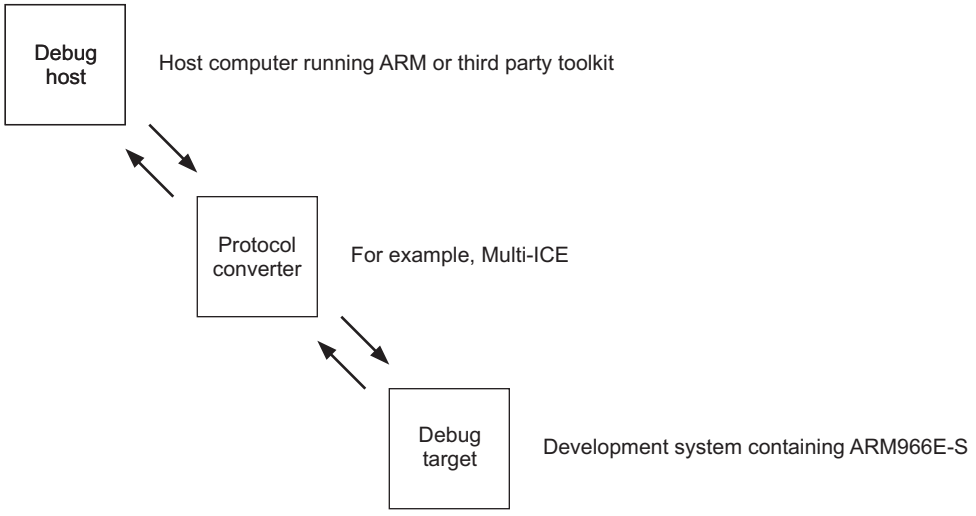


Figure 7-2 Typical debug system

A debug system typically has three parts:

- *The debug host*
- *The protocol converter*
- *ARM966E-S debug target* on page 7-5.

The debug host and the protocol converter are system-dependent.

7.2.1 The debug host

The debug host is a computer that is running a software debugger, such as *armsd*. The debug host enables you to issue high-level commands such as setting breakpoints or examining the contents of memory.

7.2.2 The protocol converter

An interface, such as a parallel port, connects the debug host to the ARM966E-S development system. The messages broadcast over this connection must be converted to the interface signals of the ARM966E-S processor. The protocol converter performs the conversion.

7.2.3 ARM966E-S debug target

The ARM9E-S core within the ARM966E-S processor has hardware extensions that ease debugging at the lowest level. The debug extensions:

- enable you to stall the core from program execution
- examine the core internal state
- examine the state of the memory system
- resume program execution.

The following major blocks of the ARM9E-S debug model are shown in Figure 7-3.

ARM9E-S CPU core

This includes hardware support for debug.

EmbeddedICE-RT logic

This is a set of registers and comparators used to generate debug exceptions (such as breakpoints). This unit is described in *About the EmbeddedICE-RT* on page 7-16.

TAP controller This controls the action of the scan chains using a JTAG serial interface.

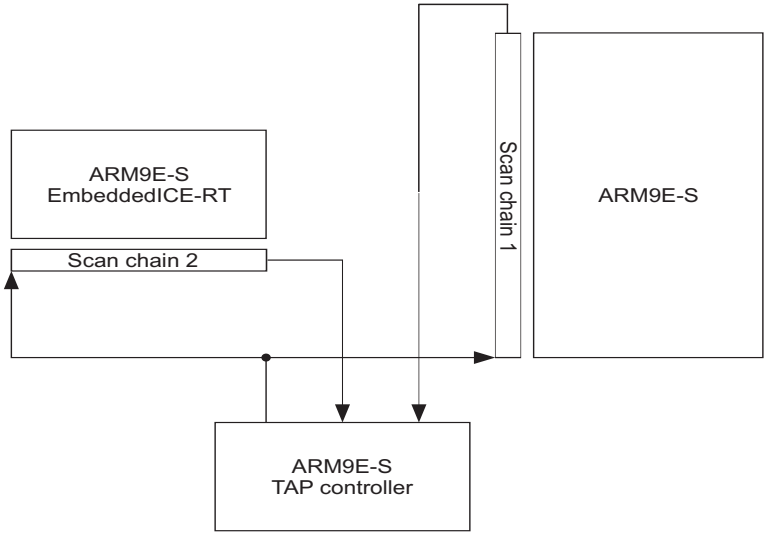


Figure 7-3 ARM9E-S block diagram

The ARM9E-S debug model is extended within the ARM966E-S processor by the addition of scan chain 15. This is used for debug access to the CP15 register bank, to enable the system state within the ARM966E-S processor to be configured while in debug state, for instance to enable or disable the TCM before performing a debug load or store.

The rest of this chapter describes the ARM9E-S and ARM966E-S hardware debug extensions.

7.3 ARM966E-S scan chain 15

Scan chain 15 is provided to enable debug access to the CP15 register bank, to enable the system state within the ARM966E-S processor to be configured while in debug state.

The order of scan chain 15 from the **DBGTDI** input to the **DBGTDO** output is shown in Table 7-1.

Table 7-1 Scan chain 15 addressing mode bit order

Bit	Content
[38]	Read = 0 write = 1
[37:32]	CP15 register address
[31:0]	CP15 register value

The CP15 register address field of scan chain 15 provides debug access to the CP15 registers is shown in Table 7-2.

Table 7-2 Mapping of scan chain 15 address field to CP15 registers

Bit [38]	Bits[37:32]	Bits[31:30]	CP15 register number	Meaning
b0	b0 0000 0	xx	c0	Read ID Register
b0	b0 0001 0	xx	c1	Read Control Register
b1	b0 0001 0	xx	c1	Write Control Register
b0	b1 1111 1	b00	c15	Read BIST Control Register
b1	b1 1111 1	b00	c15	Write BIST Control Register
b0	b1 1111 0	b01	c15	Read IBIST address
b1	b1 1111 0	b01	c15	Write IBIST address
b0	b1 1111 1	b01	c15	Read IBIST general
b1	b1 1111 1	b01	c15	Write IBIST general
b0	b1 1111 0	b11	c15	Read DBIST address

Table 7-2 Mapping of scan chain 15 address field to CP15 registers (continued)

Bit [38]	Bits[37:32]	Bits[31:30]	CP15 register number	Meaning
b1	b1 1111 0	b11	c15	Write DBIST address
b0	b1 1111 1	b11	c15	Read DBIST general
b1	b1 1111 1	b11	c15	Write DBIST general

The scan address decode overloads the existing functional decode logic that is used to access the CP15 registers during MCR and MRC instructions (see *CP15 registers* on page 2-5).

The decode overload is performed as follows:

Bit [37] Corresponds to Opcode_1 of an MCR or MRC instruction.

Bits [36:33] Corresponds to the CRn field of an MCR or MRC instruction.

Bit [32] Corresponds to bit [0] of the Opcode_2 field of an MCR or MRC instruction.

Bits [2:1] Of Opcode_2 are tied to b00 during debug state.

The debug scan chain 15 allows access to only bit[0] of the Opcode_2 field by default. To enable access to the address and general BIST registers within CP15 c15, bits [31:30] of scan chain 15 are overloaded as shown in Table 7-2 on page 7-7.

There are certain restrictions with the overloading; when writing to the BIST general registers (writing a new seed, for example). Bits[31:30] of the seed are restricted to those values shown in Table 7-2 on page 7-7. These bits are not used in the BIST address registers, therefore there are no debug restrictions when accessing these registers.

The ability to control the ARM966E-S system state through scan chain 15 provides extra debug visibility. For example, if the debugger wants to compare the contents of an address that maps to the ITCM or DTCM with the same address in AHB memory, the debugger can:

1. Load from the address with the TCM enabled to return the TCM data.
2. Disable the TCM.
3. Perform the load again. The second load now accesses the AHB because the TCM is disabled, returning the value from AHB memory.

7.4 Debug interface signals

There are four primary external signals associated with the debug interface:

- **DBGIEBKPT**, **DBGDEWPT**, and **EDBGRQ** are system requests for the ARM966E-S processor to enter debug state
- **DBGACK** is used by the ARM966E-S processor to flag back to the system that it is in debug state.

7.4.1 Entry into debug state on breakpoint

Any instruction being fetched from memory is sampled at the end of a cycle. To apply a breakpoint to that instruction, the breakpoint signal must be asserted by the end of the same cycle. This is shown in Figure 7-4 on page 7-10.

You can build external logic, such as additional breakpoint comparators, to extend the breakpoint functionality of the EmbeddedICE-RT logic. These outputs must be applied to the **DBGIEBKPT** input. This signal is ORed with the internally-generated breakpoint signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external breakpoints are possible.

A breakpointed instruction is allowed to enter the Execute stage of the pipeline, but any state change as a result of the instruction is prevented. All writes from previous instructions complete as normal.

The Decode cycle of the debug entry sequence occurs during the Execute cycle of the breakpointed instruction. The latched breakpoint signal forces the processor to start the debug sequence.

Note

The ARM966-S processor performs Thumb instruction fetches as 32-bit accesses to the AHB or TCM interfaces. As a result, external breakpoint hardware cannot identify which halfword has been requested by the ARM9E-S core as an instruction. If an external hardware breakpoint detector generates an external breakpoint, it applies to both instructions in the 32-bit word fetched from memory. External breakpoints in Thumb state must be avoided as program execution might be interrupted unintentionally. To ensure precise debug entry use the Embedded-ICE module within the ARM9E-S core.

Figure 7-4 on page 7-10 shows breakpoint timing.

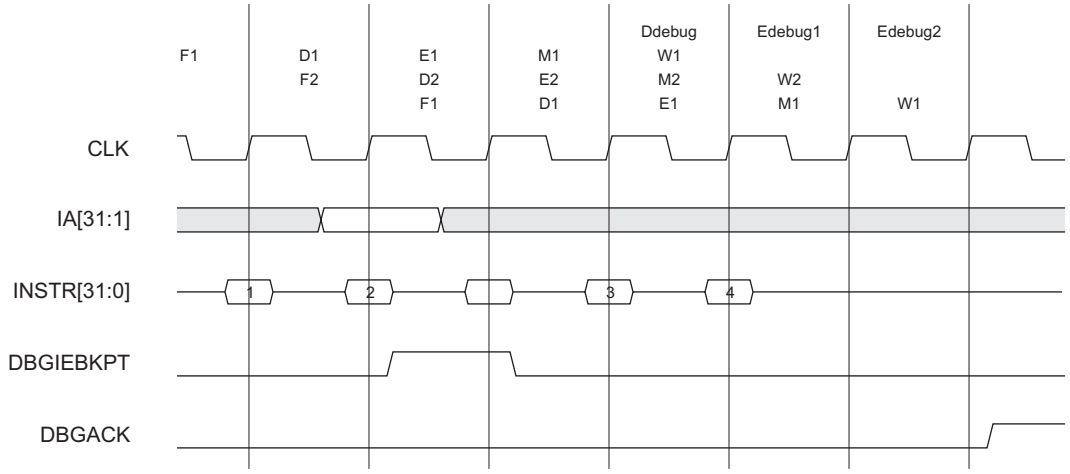


Figure 7-4 Breakpoint timing

7.4.2 Breakpoints and exceptions

If a breakpointed instruction has a Prefetch Abort associated with it, the Prefetch Abort takes priority and the breakpoint is ignored.

SWI and Undefined instructions are treated in the same way as any other instruction that might have a breakpoint set on it. Therefore, the breakpoint takes priority over the SWI or Undefined instruction.

If there is a breakpointed instruction on an instruction boundary and an **nIRQ** or **nFIQ** interrupt, the interrupt is taken and the breakpointed instruction is discarded. When the interrupt is being serviced, the execution flow returns to the original program. The instruction that was previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters debug state when the instruction reaches the Execute stage of the pipeline.

When the processor enters halt mode debug state, it is important that further interrupts do not affect the instructions executed. As soon as the processor enters halt mode debug state, interrupts are disabled, although the state of the I and F bits in the *Program Status Register* (PSR) are not affected.

7.4.3 Watchpoints

Because of the nature of the pipeline, entry into debug state following a watchpointed memory access is imprecise.

External logic, such as external watchpoint comparators, can extend the functionality of the EmbeddedICE-RT logic. Their output must be applied to the **DBGDEWPT** input. This signal is ORed with the internally-generated **Watchpoint** signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external watchpoints are possible.

After a watchpointed access, the next instruction in the processor pipeline is always allowed to complete execution. When this instruction is a single-cycle data-processing instruction, entry into debug state is delayed for one cycle while the instruction completes. The timing of debug entry following a watchpointed load in this case is shown in Figure 7-5.

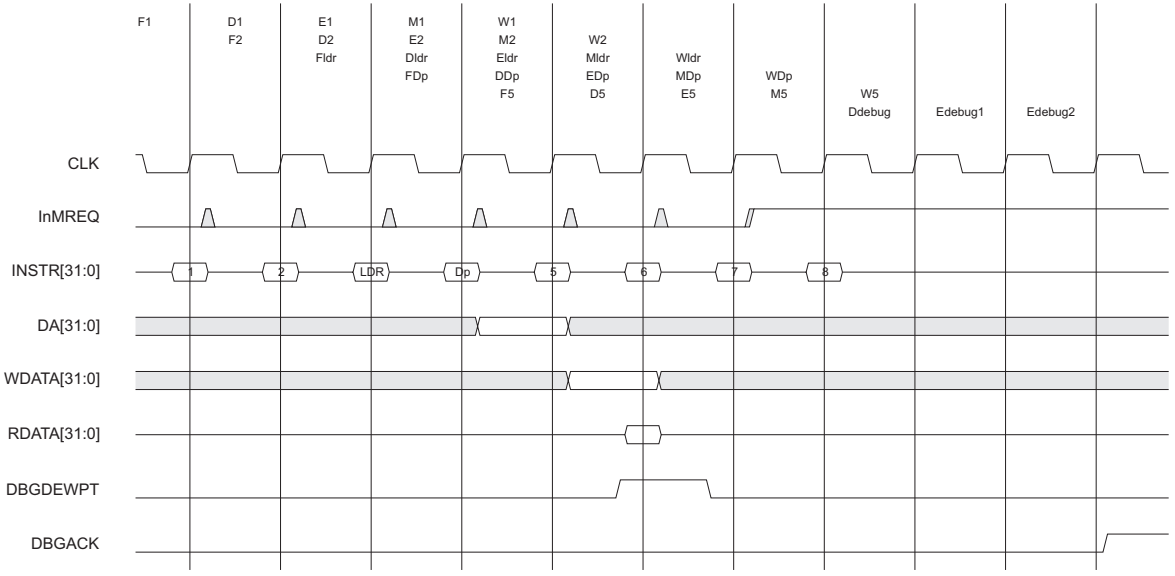


Figure 7-5 Watchpoint entry with data processing instruction

Note

Although instruction 5 enters the Execute stage, it is not executed, and there is no state update as a result of this instruction. When the debugging session is complete, normal continuation involves a return to instruction 5, the next instruction in the code sequence to be executed.

The instruction following the instruction that generated the watchpoint might have modified the *Program Counter (PC)*. If this happens, it is not possible to determine the instruction that caused the watchpoint. A timing diagram showing debug entry after a watchpoint where the next instruction is a branch is shown in Figure 7-6. However, it is always possible to restart the processor.

When the processor enters debug state, the ARM9E-S core is interrogated to determine its state. In the case of a watchpoint, the PC contains a value that is five instructions on from the address of the next instruction to be executed. Therefore, if on entry to debug state, in ARM state, the instruction `SUB PC, PC, #20` is scanned in and the processor restarted, execution flow returns to the next instruction in the code sequence.

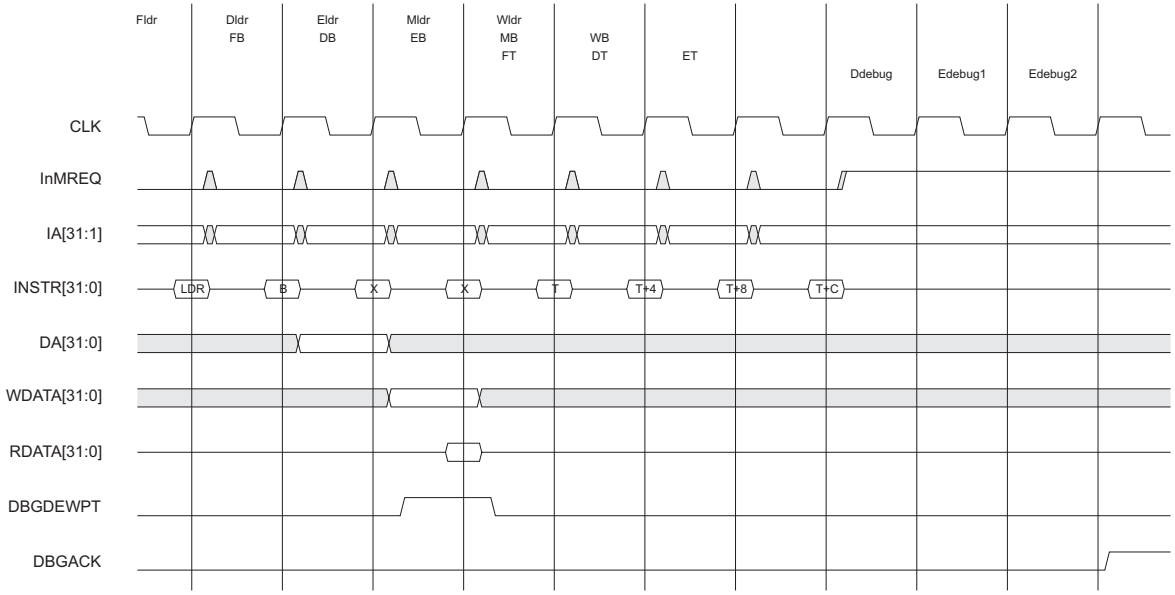


Figure 7-6 Watchpoint entry with branch

7.4.4 Watchpoints and exceptions

If there is an abort with the data access as well as a watchpoint, the watchpoint condition is latched, the exception entry sequence performed, and then the processor enters debug state. If there is an interrupt pending, again the ARM9E-S core allows the exception entry sequence to occur and then enters debug state.

7.4.5 Debug request

A debug request can take place through the EmbeddedICE-RT logic or by asserting the **EDBGRQ** signal. The request is synchronized and passed to the processor. Debug request takes priority over any pending interrupt. Following synchronization, the core enters debug state when the instruction at the Execute stage of the pipeline is completed (when Memory and Write stages of the pipeline have completed). While waiting for the instruction to finish executing, no more instructions are issued to the Execute stage of the pipeline.

Caution

Asserting **EDBGRQ** in monitor mode results in Unpredictable behavior.

7.4.6 Actions of the ARM9E-S core in debug state

When the ARM9E-S core is in debug state, both memory interfaces indicate internal cycles. This ensures that both the tightly-coupled memory that is connected to the ARM966E-S processor and the AHB interface are quiescent, allowing the rest of the AHB system to ignore the ARM9E-S core and function as normal. Because the rest of the system continues operation, the ARM9E-S core ignores aborts and interrupts.

The **HRESETn** signal must be held stable during debug. If the system applies reset to the ARM966E-S processor (**HRESETn** is driven LOW), the ARM9E-S core changes state without the knowledge of the debugger.

7.5 ARM9E-S core clock domains

The ARM966E-S processor has a single clock, **CLK**, that is qualified by two clock enables:

- **SYSCLKEN** controls access to the memory system
- **DBGTCKEN** controls debug operations.

During normal operation, **SYSCLKEN** conditions **CLK** to clock the core. When the ARM966E-S processor is in debug state, **DBGTCKEN** conditions **CLK** to clock the core.

7.6 Determining the core and system state

When the ARM966E-S processor is in debug state, you can examine the core and system state by forcing the load and store multiples into the instruction pipeline.

Before you can examine the core and system state, the debugger must determine if the processor entered debug from manual state or manual state, by examining bit[4] of the EmbeddedICE-RT debug status register. When bit [4] is HIGH, the core enters debug from Thumb state.

7.7 About the EmbeddedICE-RT

The ARM9E-S EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM9E-S core within the ARM966E-S processor.

EmbeddedICE-RT is programmed serially using the ARM9E-S TAP controller. Figure 7-7 shows the relationship between the core, EmbeddedICE-RT, and the TAP controller, with only signals that are pertinent to EmbeddedICE-RT.

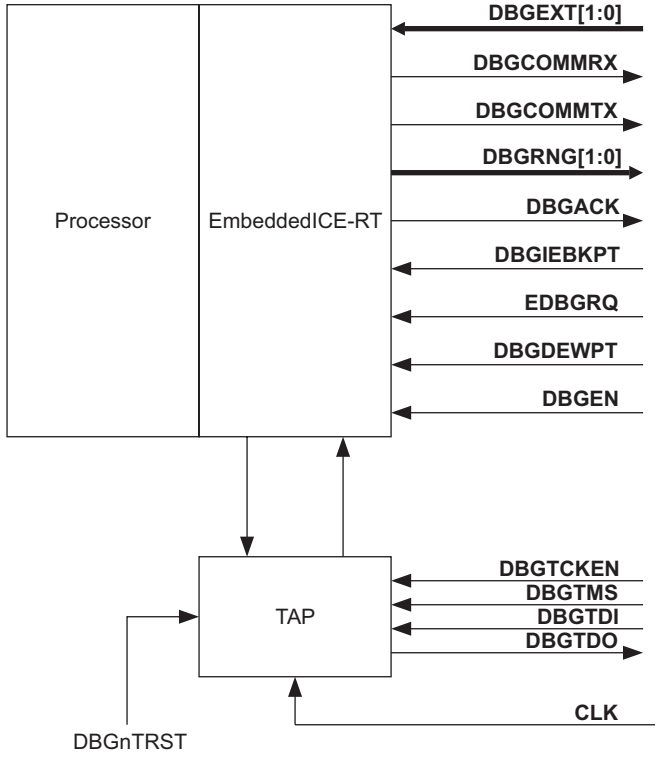


Figure 7-7 The ARM9E-S, TAP controller, and EmbeddedICE-RT

The EmbeddedICE-RT logic comprises:

- two real-time watchpoint units
- two independent registers, the Debug Control Register and the Debug Status Register
- debug communications channel.

The Debug Control Register and the Debug Status Register provide overall control of EmbeddedICE-RT operation.

You can program one or both watchpoint units to halt the execution of instructions by the core. Execution halts when the values programmed into EmbeddedICE-RT match the values currently appearing on the address bus, data bus, and various control signals.

Note

Any bit can be masked so that its value does not affect the comparison.

Each watchpoint unit can be configured to be either a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). Watchpoints and breakpoints can be data-dependent.

7.8 Disabling EmbeddedICE-RT

You can disable EmbeddedICE-RT by setting the **DBGEN** input LOW.

Caution

Permanently tying the **DBGEN** input LOW disables debug access.

When **DBGEN** is LOW, it inhibits **DBGDEWPT**, **DBGIEBKPT**, and **EDBGRQ** to the core, and **DBGACK** from the ARM966E-S processor is always LOW.

7.9 The debug communications channel

The ARM9E-S EmbeddedICE-RT logic contains a communications channel for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The communications channel comprises:

- a 32-bit Communications Data Read Register
- a 32-bit wide Communications Data Write Register
- a 6-bit wide Communications Control Register for synchronized handshaking between the processor and the asynchronous debugger.

These registers are located in fixed locations in the EmbeddedICE-RT logic register map and are accessed from the processor using MCR and MRC instructions to coprocessor 14.

In addition to the communications channel registers, the processor can access a 1-bit Debug Status Register for use in the real-time debug configuration.

7.9.1 Debug Communication Channel Registers

Table 7-3 shows the CP14 registers.

Table 7-3 Coprocessor 14 register map

Register name	Notes
CP14 c0 Communications Channel Status	Read-only
CP14 c1 Communications Channel Data Read	For reads
CP14 c1 Communications Channel Data Write	For writes
CP14 c2 Communications Channel Monitor Mode Debug Status	Read or write

7.9.2 Communications Channel Status Register

The Communications Channel Status Register is read-only. It controls synchronized handshaking between the processor and the debugger.

To read this register:

```
MRC p14, 0, <Rd>, c14, c0, 0; read Communications Channel Status Register
```

Figure 7-8 on page 7-20 shows the format of the Communications Channel Status Register.

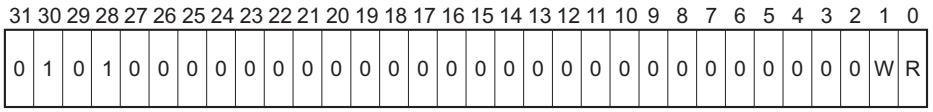


Figure 7-8 Communications Channel Status Register

Table 7-4 shows the bit fields of the Communications Channel Status Register.

Table 7-4 Communications Channel Status Register bit functions

Bit	Function
[31:28]	EmbeddedICE-RT version number
[27:2]	Reserved
[1]	Communications Channel Data Write Register ready flag: 0 = Channel ready for new data from processor 1 = Data ready for scanout.
[0]	Communications Channel Data Read Register ready flag: 0 = Channel ready for new data from debugger 1 = Data ready for processor to read.

From the viewpoint of the debugger, the registers are accessed using the scan chain in the usual way. From the viewpoint of the processor, these registers are accessed using coprocessor register transfer instructions.

You must use the following instructions:

```
MRC p14, 0, <Rd>, c0, c0
```

This returns the Debug Communications Control Register into Rd.

```
MCR p14, 0, <Rn>, c1, c0
```

This writes the value in Rn to the Communications Channel Data Write Register.

```
MRC p14, 0, <Rd>, c1, c0
```

This returns the Communications Channel Data Read Register into Rd.

Because the Thumb instruction set does not contain coprocessor instructions, you are advised to access this data using SWI instructions when in Thumb state.

7.9.3 Communications Channel Monitor Mode Debug Status register

The coprocessor 14 Debug Status Register is provided for use by a debug monitor when the ARM9E-S core is configured into monitor mode.

The coprocessor 14 Debug Status Register is a 1-bit wide read or write register having the format shown in Figure 7-9.

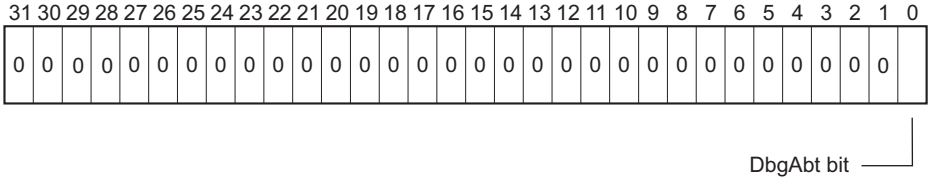


Figure 7-9 Debug Status Register

Bit [0] of the register, the **DbgAbt** bit, indicates whether the processor took a Prefetch or Data Abort in the past because of a breakpoint or watchpoint. If the ARM9E-S core takes a Prefetch Abort as a result of a breakpoint or watchpoint, then the bit is set. If on a particular instruction or data fetch, both the debug abort and external abort signals are asserted, the external abort takes priority and the **DbgAbt** bit is not set. You can read or write the **DbgAbt** bit with the MRC or MCR instructions.

This bit can be used by a real-time debug aware abort handler. This examines the **DbgAbt** bit to determine if the abort is externally or internally generated. If the **DbgAbt** bit is set, the abort handler initiates communication with the debugger over the communications channel.

7.9.4 Communications using the communications channel

Messages can be sent and received using the communications channel as described in:

- *Sending a message to the debugger*
- *Receiving a message from the debugger* on page 7-22.

Sending a message to the debugger

When the processor wants to send a message to the debugger, it must check that the Communications Channel Data Write Register is free for use by determine if the W bit of the Debug Communications Control Register is clear.

The processor reads the Debug Communications Control Register to check status of the W bit.

- If the W bit is clear, the Communications Channel Data Write Register is clear.

- If the W bit is set, previously written data is not read by the debugger. The processor must continue to poll the control register until the W bit is clear.

When the W bit is clear, a message is written by a register transfer to coprocessor 14. Because the data transfer occurs from the processor to the Communications Channel Data Write Register, the W bit is set in the Debug Communications Control Register.

The debugger sees both the R and W bits when it polls the Debug Communications Control Register through the JTAG interface. When the debugger sees that the W bit is set, it can read the Communications Channel Data Write Register, and scan the data out. The action of reading this data register clears the Debug Communications Control Register W bit. At this point, the communications process can begin again.

Receiving a message from the debugger

Transferring a message from the debugger to the processor is similar to sending a message to the debugger. In this case, the debugger polls the R bit of the Debug Communications Control Register.

- if the R bit is LOW, the Communications Channel Data Read Register is free, and data can be placed there for the processor to read
- if the R bit is set, previously deposited data is not yet collected, so the debugger must wait.

When the Communications Channel Data Read Register is free, data is written there using the JTAG interface. The action of this write sets the R bit in the Debug Communications Control Register.

The processor polls the Debug Communications Control Register. If the R bit is set, there is data that can be read using an MRC instruction to coprocessor 14. The action of this load clears the R bit in the Debug Communications Control Register. When the debugger polls this register and sees that the R bit is clear, the data is taken, and the process can be repeated.

7.10 Monitor mode debug

The ARM9E-S core within ARM966E-S processor contains logic that enables the debugging of a system without stopping the core entirely. This enables the continued servicing of critical interrupt routines while the core is being interrogated by the debugger. Setting bit [4] of the Debug Control Register enables the real-time debug features of ARM9E-S core. When this bit is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the core to enter abort mode, taking the Prefetch Abort or Data Abort vectors respectively. When the core is configured for real-time debugging, you must be aware of the following restrictions:

- Breakpoints or watchpoints cannot be data-dependent. No support is provided for use of the range functionality. Breakpoints or watchpoints can only be based on:
  - instruction or data addresses
  - external watchpoint conditioner (**DBGEXTERN**)
  - user or privileged mode access (**DnTRANS** and **InTRANS**)
  - read or write access (watchpoints)
  - access size (breakpoints, **ITBIT**, and watchpoints, **DMAS[1:0]**)
  - chained comparisons.
- The single-step hardware is not enabled.
- External breakpoints and watchpoints are not supported.
- The vector catching hardware can be used but must not be configured to catch the Prefetch or Data Abort exceptions.

Caution

No support is provided to mix halt mode and monitor mode debug functionality. When the core is configured into the monitor mode, asserting the external **EDBGRQ** signal results in Unpredictable behavior. Setting the internal **EDBGRQ** bit results in Unpredictable behavior.

When an abort is generated by the monitor mode, it is recorded in the Communications Channel Monitor Mode Debug Status Register in coprocessor 14 (see *Communications Channel Monitor Mode Debug Status register* on page 7-21).

Because the monitor mode debug does not put the ARM9E-S core into debug state, it is necessary to change the contents of the watchpoint registers while TCM accesses are taking place, rather than being changed when in debug state. If the watchpoint registers are written to during an access, all matches from the affected watchpoint unit using the register being updated are disabled for the cycle of the update.

If there is a possibility of false matches occurring during changes to the watchpoint registers, caused by old data in some registers and new data in others, then you must:

1. Disable that watchpoint unit using the Control Register for that watchpoint unit.
2. Change the other registers.
3. Re-enable the watchpoint unit by rewriting the Control Register.

7.11 Debug additional reading

A more detailed description of the ARM9E-S debug features and JTAG interface is provided in Appendix B Debug in Depth in the *ARM9E-S Technical Reference Manual*.

7.11.1 ARM966E-S JTAG TAP ID

The ARM966E-S processor TAP ID inputs are connected to the TAP ID Register inside the ARM9E-S core. Figure 7-10 shows the bit order in the TAP ID Register.

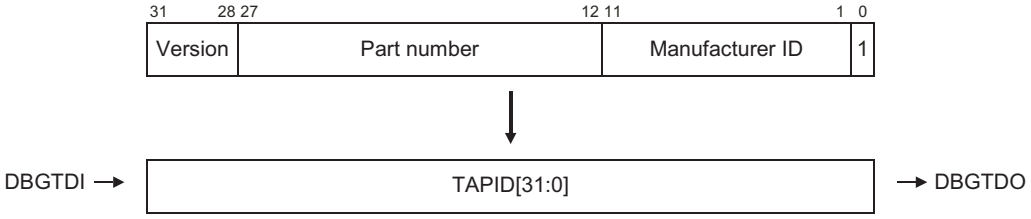


Figure 7-10 TAP ID Register bit order

Table 7-5 shows the settings for the ARM966E-S TAPID inputs on the debug interface. Set your TAPID according to the rules contained in *Application Note 99 - Core Type and Revision Identification*. The default TAPID for the ARM966E-S processor is `0x25966477`.

Table 7-5 TAP ID Register bit functions

Bit	Function	Value
[31:28]	Specification revision	b0010
[27:12]	Product code	b0101 1001 0110 0110
[11:1]	Manufacturer ID, manufacturer-specific	Default is b01000111011
[0]	IEEE standard specified, always 1	1

Chapter 8
Embedded Trace Macrocell Interface

This chapter describes the ARM966E-S *Embedded Trace Macrocell* (ETM) interface. It contains the following sections:

8.1 About the ETM interface

The ARM966E-S processor supports the connection of an external *Embedded Trace Macrocell* (ETM) to provide real-time code tracing of the ARM966E-S processor in an embedded system. See the *ETM9 Technical Reference Manual* for further information.

The ETM interface is primarily one way. To provide code tracing, the ETM block must be able to monitor various ARM9E-S inputs and outputs. The required ARM9E-S inputs and outputs are collected and driven out from the ARM966E-S processor from the ETM interface registers, as shown in Figure 8-1.

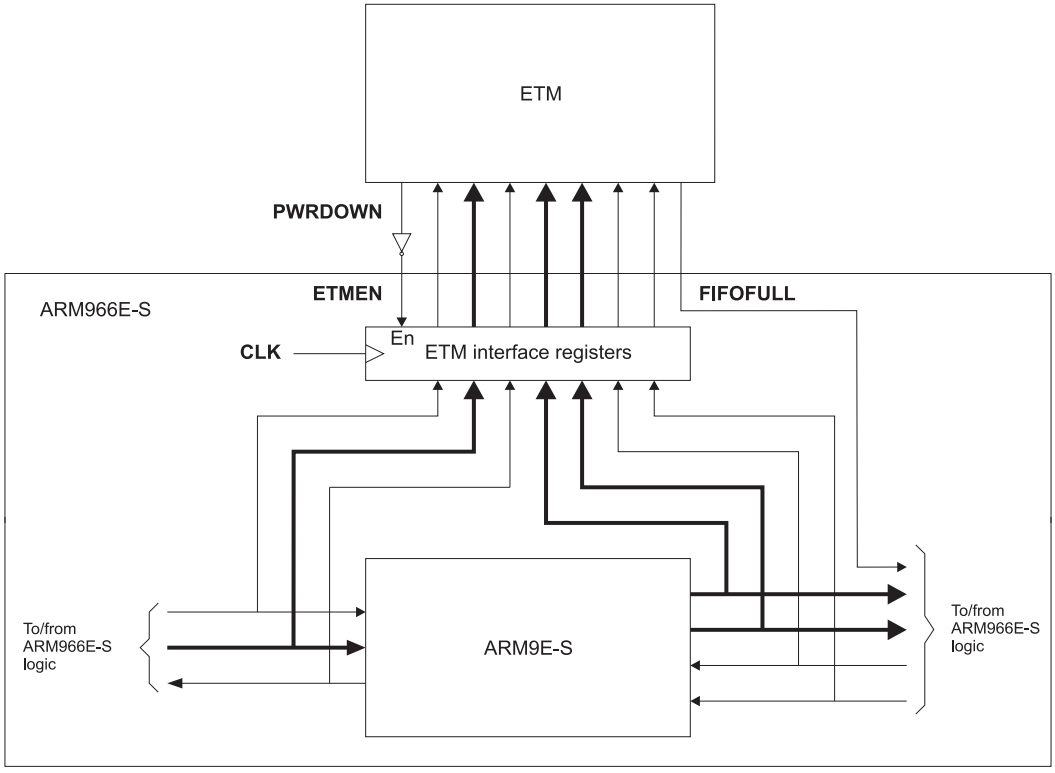


Figure 8-1 ARM966E-S ETM interface

The ETM interface outputs are pipelined to provide early output timing and to isolate any ETM input load from the critical ARM966E-S signals. The latency of the pipelined outputs does not effect ETM trace behavior because all outputs are delayed by the same amount.

8.2 Enabling the ETM interface

The ETM interface on the ARM966E-S processor is enabled by the top-level pin **ETMEN**. When this input is HIGH, the ETM interface is enabled and the outputs are driven so that an external ETM can begin code tracing.

When the **ETMEN** input is driven LOW, the ETM interface outputs are held at their last value before the interface was disabled.

The **ETMEN** input is usually driven by the ETM, and driven HIGH when the ETM is programmed using its TAP controller.

ARM Limited recommends that you connect the **ETMEN** input to the **PWRDOWN** output of the ETM9 macrocell through an inverter as shown in Figure 8-1 on page 8-2.

Note

If an ETM is not used in an embedded ARM966E-S design, you must tie the **ETMEN** input LOW to save power.

8.3 ARM966E-S trace support features

The trace support uses the following features:

- *FIFOFULL*
- *Configuration Control Register*
- *Trace Process Identifier Register*.

8.3.1 FIFOFULL

The **FIFOFULL** signal is an input to the ARM966E-S processor driven by the ETM9 macrocell. Whenever the programmed upper watermark of the ETM FIFO is filled, **FIFOFULL** is asserted. The ARM966E-S processor uses **FIFOFULL** to stall the ARM9E-S core, preventing trace loss. The ARM9E-S core remains stalled until **FIFOFULL** is deasserted.

The ARM966E-S processor can only stall on instruction boundaries enabling any current AHB transfers to complete. You must take this into consideration when programming the ETM FIFO watermark. If the instruction being executed when FIFOFULL is asserted is either LDM or STM, the FIFO must be able to accept up to 16 words after the assertion of **FIFOFULL**, to prevent trace lost.

Note

Using **FIFOFULL** to stall the ARM966E-S processor affects real-time operating performance.

8.3.2 Configuration Control Register

The Configuration Control Register CP15 c15 enables the masking of interrupts during trace. This register enables **nIRQ** and **nFIQ** interrupt priority over **FIFOFULL** to be programmed. See *CP15 c15, Test and Configuration Register* on page 2-11 for a description of this register.

8.3.3 Trace Process Identifier Register

The Trace Process Identifier Register enables real-time trace tools to identify the currently executing process in multitasking environments. See *CP15 c13, Trace Process Identifier Register* on page 2-10 for a description of this register.

Chapter 9
Test Support

This chapter describes the test methodology employed for the ARM966E-S synthesized logic and tightly-coupled memory. It contains the following sections:

9.1 About the ARM966E-S test methodology

To achieve a high level of fault coverage, scan insertion and ATPG techniques are used on the ARM9E-S core and ARM966E-S control logic as part of the synthesis flow. BIST is used to provide high fault coverage of the compiled TCM.

9.2 Scan insertion and ATPG

This technique is covered in detail in the *ARM966E-S Implementation Guide*. Scan insertion requires that all register elements are replaced by scannable versions that are then connected up into a number of large scan chains. These scan chains are used to set up data patterns on the combinatorial logic between the registers, and capture the logic outputs. The logic outputs are then scanned out while the next data pattern is scanned in.

Automatic Test Pattern Generation (ATPG) tools are used to create the necessary scan patterns to test the logic, when the scan insertion has been performed. This technique achieves a very high fault coverage for the standard cell combinatorial logic, typically in the 95-99% range.

Because of the larger scan register elements and the serial routing between them, scan insertion does have an impact on the area and performance of the synthesized design. To minimize these effects, perform scan insertion early in the synthesis cycle and reoptimize the design with the scan elements in place.

9.2.1 ARM966E-S test wrapper

To improve test coverage where there is no internal visibility of the macrocell, you can use a test wrapper. For logic to be testable, the input to the logic must be controllable using a scan chain, and so must be driven by a register. The output of the logic must also be observable through a scan chain, and so must be registered.

If the ARM966E-S macrocell is integrated into an ASIC as a black box, the test tools do not have visibility of the internal ARM966E-S scan chains and cannot create vectors to cover any logic between the last register in the ARM966E-S macrocell and the next register in the ASIC. This is known as a *test shadow* and leads to a reduction in test coverage.

The addition of a test wrapper enables this shadow logic to be tested. The test wrapper is a scan chain around the periphery of the ARM966E-S macrocell that connects to each input and output. The test wrapper scan chain can be used in two modes:

- INTEST mode
- EXTEST mode.

The INTEST wrapper is required only for embedded ARM966E-S macrocells. In INTEST mode, all macrocell inputs are driven using the test wrapper scan chain, and all macrocell outputs are observable through the test wrapper scan chain. This enables a complete set of ATPG vectors to be produced for the ARM966E-S macrocell in isolation. In EXTEST mode, all macrocell outputs are driven using the test wrapper scan chain, and all macrocell inputs are observable through the test wrapper scan chain. This enables the logic surrounding the ARM966E-S macrocell to be tested without the test tools requiring internal visibility of the ARM966E-S macrocell.

9.3 BIST of tightly-coupled memory

Adding a simple memory test controller enables an exhaustive test of the memory arrays to be performed. You can perform a BIST test with an MCR instruction to the CP15 BIST Control Register. You can run the BIST test on one or both of the ITCM and DTCM simultaneously.

When you perform a BIST test on a TCM, the functional enable for that TCM is automatically disabled, forcing all memory accesses to that TCM address space to go to the AHB. This enables BIST tests to be run in the background. For instance, the instruction TCM can be BIST tested, while code is executed over the AHB.

You can achieve full programmer control over the BIST mechanism through five registers that are mapped to the CP15 c15 address space. See *CP15 c15, Test and Configuration Register* on page 2-11 for details of the MCR or MRC instructions used to access these registers. Access to these registers is also available in debug mode, see *Debug interface signals* on page 7-9.

Note

The IBIST and DBIST blocks are functionally independent.

9.3.1 BIST algorithm

The BIST test algorithm is a 6N test. Figure 9-1 on page 9-5 shows the test flow. The first pass starts from the bottom of the memory to be tested. A fixed value is written into each memory address to be tested and the address is incremented until the top of memory is reached.

The second pass starts from the bottom of the memory to be tested. In the second pass, the fixed pattern is checked. If the pattern match fails then the BIST fail flags are set and the test fails. If the pattern match is successful then the inverse pattern is written to each memory address. The inverse pattern is checked. If the pattern match fails then the BIST fail flags are set and the test fails. The address is incremented until the top of memory is reached.

The third pass starts from the top of memory. A fixed value is written into each memory address to be tested. The pattern is then checked. If the pattern match fails then the BIST fail flags are set and the test fails. The address is decremented until the bottom of the area of memory under test is reached.

During all reads, the TCMERROR flag is monitored and if an error is found, the test fails.

All BIST accesses are marked nonsequential by the SEQ flag being LOW.

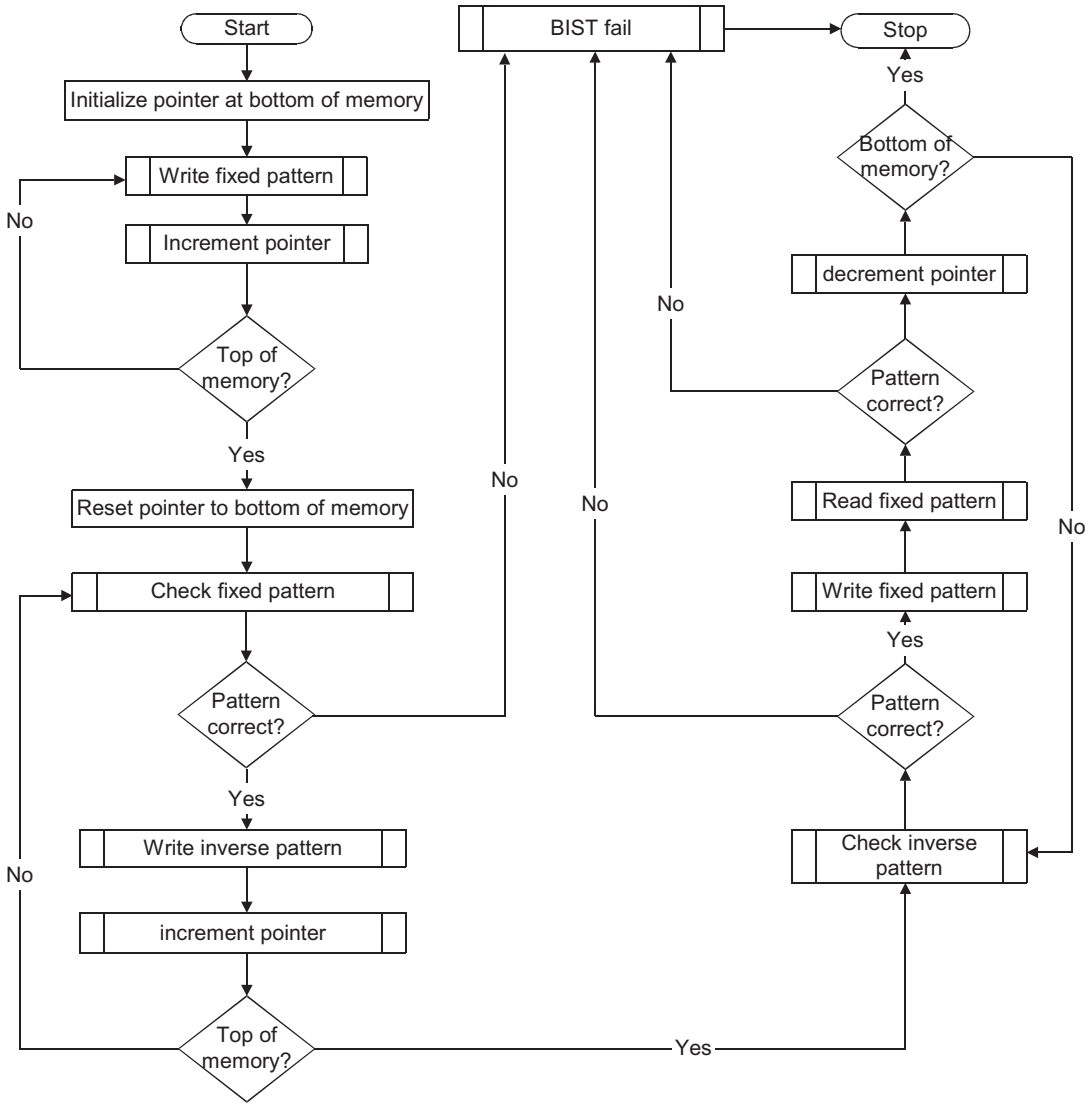


Figure 9-1 Test flow for BIST

9.3.2 BIST control register

This controls the operation of the TCM memory BIST. Before initiating a BIST test, an MCR instruction is first performed to the BIST Control Register to set up the size of the test and enable the TCM to be tested. A further MCR is required to initiate the test.

You can access the current status of a BIST test and the result of a completed test by performing an MRC to the BIST Control Register. This operation returns flags to indicate that a test is:

- running
- paused
- failed
- completed.

In addition to returning the state for the size of the test and TCM enable status after completing a BIST test, the BIST test enable must first be cleared by writing to the BIST Control Register if you are to use the TCM for functional operation. You must then re-enable the TCM by writing to CP15 c1. This is necessary because the BIST test enable automatically clears the functional enable.

Note

Clearing the functional TCM enable when BIST is enabled prevents the programmer from trying to run from TCM following a BIST test, without having first reprogrammed the TCM. This reprogramming is necessary because the BIST algorithm corrupts all tested TCM locations.

9.3.3 BIST address and general registers

The BIST Control Register enables standard BIST operations to be performed on each TCM and the size of the test to be specified. Additional registers provide functionality for:

- testing of the BIST hardware
- changing the fixed pattern data for a BIST test
- providing a nonzero starting address for a BIST test
- peek and poke of the TCM
- returning an address location for a failed BIST test
- returning failed data from the failing address location.

This additional functionality is most useful for debugging faulty silicon during production test. The exception to this is the start address for a BIST test. You can perform BIST of the TCM periodically throughout program execution a block at a time, rather than all at once, by incrementing the start address by the test block size each time a test is initiated.

Table 9-1 and Table 9-2 on page 9-8 show how the registers are used. The pause bits in the BIST Control Register provide extra decode of these registers. The BIST hardware is only capable of 32-bit word addressing. When writing to an address register, bits [1:0] are truncated, and when reading from an address register, bits [1:0] always read as zero. Similarly, bits above the usable address range are truncated and are read as zero.

Table 9-1 Instruction BIST address and general registers

BIST register	IBIST pause	Read	Write
IBIST address register	0	IBIST fail address	IBIST start address
IBIST address register	1	IBIST fail address	IBIST peek/poke address
IBIST general register	0	IBIST fail data	IBIST pattern data
IBIST general register	1	IBIST peek data	IBIST poke data

The IBIST start address and peek or poke address registers share a hardware register, therefore writing to one corrupts the state of the other. The IBIST start address must be restored to the correct state after peek/poke operations, if the test is to be restarted from a User or Auto pause state.

The IBIST pattern data and poke data registers share a hardware register, therefore writing to one corrupts the state of the other. The IBIST pattern data must be restored to the correct state after poke operations, if the test is to be restarted from a User or Auto pause state.

The IBIST fail address and fail data registers only contain valid failure information after a failure has occurred and are invalidated when a new test is started or whenever TCM peek or poke operations are performed. Because the top address bit of IBIST fail address is not needed to give the fail address, it is used to indicate if a failure was due to **ITCMERROR**.

Table 9-2 Data BIST address and general registers

BIST register	IBIST pause	Read	Write
DBIST address register	0	DBIST fail address	DBIST start address
DBIST address register	1	DBIST fail address	DBIST peek/poke address
DBIST general register	0	DBIST fail data	DBIST pattern data
DBIST general register	1	DBIST peek data	DBIST poke data

The DBIST start address and peek or poke address registers share a hardware register, therefore writing to one corrupts the state of the other. The IBIST start address must be restored to the correct state after peek or poke operations, if the test is to be restarted from a User or Auto pause state.

The DBIST pattern data and poke data registers share a hardware register, therefore writing to one corrupts the state of the other. The IBIST pattern data must be restored to the correct state after poke operations, if the test is to be restarted from a User or Auto pause state.

The DBIST fail address and fail data registers only contain valid failure information after a failure has occurred and are invalidated when a new test is started or whenever TCM peek or poke operations are performed. Because the top address bit of DBIST fail address is not needed to give the fail address, it is used to indicate if a failure was due to **DTCMERROR**.

9.3.4 Running a test

To start a test, perform the following:

1. Write to the BIST Control Register with relevant pause bit and start strobe bits cleared, enable bits set, and a suitable size value. The TCM is disabled for normal core accesses from this time onwards.
2. Write suitable values to the BIST start address and pattern data registers.

3. Write the BIST Control Register with the BIST start strobe bit set, and the pause bit cleared (for Normal or User pause mode) or set (for Auto pause mode).

The test runs, and the BIST running flag is set. If a failure occurs, the test hardware stores the failed address and data, and then goes to the idle state. At this point the running flag is cleared, the completion and fail flags are set. If the test completes without failures, the BIST running flag is cleared and the completion flag is set. If the test is paused (User or Auto), the BIST running flag is cleared, and is set again when the test is restarted.

Note

The completion and fail flags retain their state between test invocations. They are only reset when a new test is started.

9.3.5 Peek and poke

Peek and poke functions require the relevant pause bit in the BIST Control Register to be set.

To read a location in the TCM:

1. Write the address to the BIST peek or poke address register.
2. Read the BIST peek data register.

The address write in step 1 initiates the TCM read operation. If necessary (because of TCM wait states), the peek data register read causes the core to stall until the data is available.

To write to a location in the TCM:

1. Write the address to the BIST peek and poke address register.
2. Write the data to the BIST poke data register.

The data write in step 1 initiates the TCM write operation. If necessary (because of TCM wait states), subsequent peek or poke operations cause the core to stall until the write has completed.

9.3.6 Pause modes

The suggested production test sequence for the TCM is:

1. Test each TCM using a full test.

2. Test the BIST hardware for each TCM. To enable testing of the BIST hardware, a pause mechanism enables the BIST test to be halted and data within the TCM to be corrupted. The sequence for this is:
   a. Writing the address for the location to be corrupted with an MCR to the relevant BIST address register.
   b. Writing the corrupted data using an MCR to the BIST general register.
   c. Restarting the test by an MCR to the BIST Control Register.
   d. Checking that the corrupted data causes the test to fail by reading the failed address and data from the BIST address and general registers.

In addition to controlling the addressing within the address and general registers, the pause bit also controls the progression of the BIST algorithm as follows:

- *Auto pause*
- *User pause*.

Auto pause

If the pause bit is set in the BIST Control Register when the BIST start strobe is asserted, the test runs in Auto pause mode. The BIST test will then pause at the end of the first pass or at the end of the second pass of the BIST algorithm, whichever comes first. This point is the top (for the first pass) or the bottom (for the second pass) of the memory area being tested.

The programmer can poll the BIST Control Register to detect when a test has paused (the running flag is LOW). Data can then be corrupted before restarting the BIST test. To restart the test, perform an MCR to the BIST Control Register with the start strobe bit set, after first ensuring the pause bit is in the correct state for subsequent operation.

User pause

If the pause bit is clear when the BIST start strobe is asserted, the test is run in User pause mode. The BIST algorithm is paused by an MCR to the BIST Control Register, setting the pause bit for the TCM being tested. The TCM contents are then corrupted as previously. This stops the BIST algorithm at a potentially unknown point, resulting in the possibility that the BIST algorithm overwrites and therefore does not cause a test to fail.

To restart the test, perform an MCR to the BIST Control Register with the start strobe bit set, after first ensuring the pause bit is in the correct state for subsequent operation.

Note

User pause mode is provided for production test debugging to shorten a test by pausing the algorithm early. The Auto pause mode is recommended to provide BIST hardware testing for all other occasions.

Appendix A
Signal Descriptions

This appendix describes the ARM966E-S signals. It contains the following sections:

A.1 Signal properties and requirements

To ensure ease of integration of the ARM966E-S processor into embedded applications and to simplify synthesis flow, the following design techniques have been used:

- a single rising edge clock times all activity
- all signals and buses are unidirectional
- all inputs are required to be synchronous to the single clock.

These techniques simplify the definition of the top-level ARM966E-S signals because all outputs change from the rising edge and all inputs are sampled with the rising edge of the clock. In addition, all signals are either input or output only, as bidirectional signals are not used.

Note

Asynchronous signals, for example interrupt sources, must first be synchronized by external logic before being applied to the ARM966E-S processor.

A.2 Clock interface signals

Table A-1 describes the ARM966E-S clock interface signals.

Table A-1 Clock interface signals

Name	Direction	Description
CLK	Input	This clock times all operations in the ARM966E-S design. All outputs change from the rising edge and all inputs are sampled on the rising edge. The clock might be stretched in either phase. Through the use of the **HCLKEN** signal, this clock also times AHB operations. Through the use of the **DBGTCKEN** signal, this clock also times debug operations.
HCLKEN	Input	Synchronous enable for AHB transfers. When HIGH, indicates that the next rising edge of **CLK** is also a rising edge of **HCLK** in the AHB system in which the ARM966E-S processor is embedded. **HCLK** must be tied HIGH in systems where **CLK** and **HCLK** are intended to be the same frequency.
DBGTCKEN	Input	Synchronous enable for debug logic accessed by the JTAG interface. When HIGH on the rising edge of **CLK**, the debug logic is able to advance.
HRESETn	Input	Asynchronously asserted LOW input used to initialize the ARM966E-S system state. Synchronously deasserted.

A.3 AHB signals

Table A-2 describes the ARM966E-S AHB signals.

Table A-2 AHB signals

Name	Direction	Description
HADDR[31:0]	Output	The 32-bit AHB system address bus.
HTRANS[1:0]	Output	Indicates the type of ARM966E-S transfer, which can be: b00 = idle b10 = nonsequential b11 = sequential.
HWRITE	Output	When HIGH, indicates a write transfer. When LOW, indicates a read transfer.
HSIZE[2:0]	Output	Indicates the size of an ARM966E-S transfer, which can be: b000 = byte b001 = halfword b010 = word.
HBURST[2:0]	Output	Indicates if the transfer forms part of a burst. The ARM966E-S processor supports: b000 = single transfer b001 = incremental burst of unspecified length b011 = 4-beat incremental burst b101 = 8-beat incremental burst b111 = 16-beat incremental burst.
HPROT[3:0]	Output	Protection control signals. The signals indicate if: • the transfer is an opcode fetch or data access • the transfer is a Supervisor or User mode access • the current access is Cachable or Bufferable.
HWDATA[31:0]	Output	The 32-bit write data bus is used to transfer data from the ARM966E-S processor to a selected bus slave during write operations.
HRDATA[31:0]	Input	The 32-bit read data bus is used to transfer data from a selected bus slave to the ARM966E-S processor during read operations.
HREADY	Input	When HIGH, indicates that a transfer has finished on the bus. You can drive this signal LOW to extend a transfer.
HRESP[1:0]	Input	The transfer response from the selected slave provides additional information on the status of the transfer. The response can be: 00 = Okay 01 = Error 10 = Retry 11 = Split.

Table A-2 AHB signals (continued)

Name	Direction	Description
HBUSREQ	Output	Indicates that the ARM966E-S processor requires the bus.
HLOCK	Output	When HIGH, indicates that the ARM966E-S processor requires locked access to the bus and no other master is granted until this signal has gone LOW. Asserted by the ARM966E-S processor when executing SWP instructions to AHB address space.
HGRANT	Input	Indicates that the ARM966E-S processor is currently the highest priority master. Ownership of the address and control signals changes at the end of a transfer when **HREADY** is HIGH, so the ARM966E-S processor gets access to the bus when both **HREADY** and **HGRANT** are HIGH.

A.4 TCM interface signals

This section contains the following signal descriptions:

- *Data TCM interface signals*
- *Instruction TCM interface signals* on page A-7.

A.4.1 Data TCM interface signals

Table A-3 shows the data TCM interface signals.

Table A-3 Data TCM interface signals

Signal	Direction	Function
DTCMERROR	Input	Data TCM error signal This signal enables the ARM9E-S core to be informed of error conditions during read accesses.
DTCMnRW	Output	Data TCM not read, write: 0 = Read access 1 = Write access.
DTCMADDR[23:0]	Output	Data TCM address. Addresses up to 64Mbytes.
DTCMWD[31:0]	Output	Data TCM write data.
DTCMCS	Output	Data TCM chip select. Indicates a write or a read access. Active HIGH.
DTCMWE[3:0]	Output	Data TCM byte write indicator. Each bit indicates which of the corresponding bytes in **DTCMWD** are written to the RAM. The least significant bit of **DTCMWE** refers to the least significant byte of **DTCMWD**. For example: b0000 = No write b0001 = Byte write to the least significant byte b1000 = Byte write to the most significant byte b0011 = A half-word write to the least significant two bytes. Bits of **DTCMWE** are set only when a write is taking place, so when **DTCMnRW** is not set, bits of **DTCMWE** are not set.
DTCMSEQ	Output	Data address sequential signal: 0 = non sequential address 1 = sequential address.
DTCMRD[31:0]	Input	Data TCM read data.

Table A-3 Data TCM interface signals (continued)

Signal	Direction	Function
DTCMWAIT	Input	Data TCM wait. If HIGH, the Data TCM cannot service any requests in the following cycle. This signal is used to stall the ARM9E-S for multiple cycle access RAM on the data TCM interface. This signal is active HIGH. For single port RAM, this signal is used to stall the core for direct memory access to the DTCM.
DTCMSIZE[4:0]	Input	Data TCM size. Static configuration pins. The data TCM size is configurable in the range 0 bytes to 64 MB: b00000 = 0 byte b00001 = 1KB b00010 = 2KB b00011 = 4KB b00100 = 8KB b00101 = 16KB b00110 = 32KB b00111 = 64KB b01000 = 128KB b01001 = 256KB b01010 = 512KB b01011 = 1 MB b01100 = 2 MB b01101 = 4 MB b01110 = 8 MB b01111 = 16 MB b10000 = 32 MB b10001 = 64 MB The supported sizes are 0 and 2^nKB for n = 0 to 16
DTCMCANCEL	Output	Data TCM access cancel signal. Pipelined by one clock cycle relative to the **DTCMCS** it refers to: 0 = Data from read access started in previous clock cycle is used by the ARM9E-S 1= Data from read access started in previous clock cycle is not used by the ARM9E-S.

A.4.2 Instruction TCM interface signals

Table A-4 shows the instruction TCM interface signals.

Table A-4 Instruction TCM interface signals

Signal	Direction	Function
ITCMERROR	Input	Instruction TCM error signal. Enables the ARM9E-S core to read error conditions during read accesses.
ITCMnRW	Output	Instruction TCM not read, write: 0 = Read access 1 = Write access.
ITCMADDR[23:0]	Output	Instruction TCM address. Addresses up to 64MB. Output delay 90% of clock cycle.
ITCMWD[31:0]	Output	Instruction TCM write data.
ITCMCS	Output	Instruction TCM chip select. Indicates a write or a read access. Active HIGH.

Table A-4 Instruction TCM interface signals (continued)

Signal	Direction	Function
ITCMWE[3:0]	Output	Instruction TCM byte write indicator. Each bit indicates which of the corresponding bytes in **ITCMWD** are written to the RAM. The least significant bit of **ITCMWE** refers to the least significant byte of **ITCMWD**. For example: b0000 = No write b0001 = Byte write to the least significant byte b1000 = Byte write to the most significant byte b0011 = A half-word write to the least significant two bytes. Bits of **ITCMWE** are set only when a write is taking place, so when **ITCMnRW** is not set, bits of **ITCMWE** are not set.
ITCMSEQ	Output	Instruction address sequential signal: 0 = Nonsequential address 1 = Sequential address.
ITCMRD[31:0]	Input	Instruction TCM read data.
ITCMWAIT	Input	Instruction TCM wait. If HIGH, the Instruction TCM cannot service any requests in the following cycle. This signal is used to stall the ARM9E-S for multiple cycle access RAM on the instruction TCM interface. This signal is active HIGH. For single port RAM, this signal is used to stall the core for direct memory access to the ITCM.
ITCMSIZE[4:0]	Input	Instruction TCM size. Static configuration pins. The instruction TCM size is configurable in the range 0 bytes to 64 MB: b00000 = 0 byte b00001 = 1KB b00010 = 2KB b00011 = 4KB b00100 = 8KB b00101 = 16KB b00110 = 32KB b00111 = 64KB b01000 = 128KB b01001 = 256KB b01010 = 512KB b01011 = 1 MB b01100 = 2 MB b01101 = 4 MB b01110 = 8 MB b01111 = 16 MB b10000 = 32 MB b10001 = 64 MB The supported sizes are 0 and 2^nKB for n = 0 to 16
ITCMCANCEL	Output	Instruction TCM access cancel signal. Pipelined by one clock cycle relative to the **ITCMCS** it refers to: 0 = Data from read access started in previous clock cycle is used by the ARM9E-S 1= Data from read access started in previous clock cycle is not used by the ARM9E-S.

A.5 Coprocessor interface signals

Table A-5 describes the ARM966E-S coprocessor interface signals.

Table A-5 Coprocessor interface signals

Name	Direction	Description
CPBURST[3:0]	Input	This signal indicates the number of words to be transferred as part of a burst from an external coprocessor. This signal allows a maximum of 16 words to be transferred.**CPBURST** must be driven to b0000 when there is no coprocessor instruction in the Decode stage of the coprocessor pipeline. Tie off to zero if no external coprocessors are present.
CPCLKEN	Output	Synchronous enable for coprocessor pipeline follower. When HIGH on the rising edge of **CLK**, the pipeline follower logic can advance.
CPINSTR[31:0]	Output	The 32-bit coprocessor instruction bus for transferring instructions to the coprocessor pipeline follower.
CPDOUT[31:0]	Output	The 32-bit coprocessor read data bus for transferring data to the coprocessor.
CPDIN[31:0]	Input	The 32-bit coprocessor write data bus for transferring data from the coprocessor.
CPPASS	Output	Indicates that there is a coprocessor instruction in the Execute stage of the pipeline, and it must be executed.
CPLATECANCEL	Output	If HIGH during the first memory cycle of a coprocessor instruction, then the coprocessor must cancel the instruction without changing any internal state. This signal is only asserted in cycles where the previous instruction caused a Data Abort to occur.
CHSDE[1:0]	Input	The handshake signals from the Decode stage of the coprocessor pipeline follower. Indicates: ABSENT = 10 WAIT = 00 GO = 01 LAST = 11.
CHSEX[1:0]	Input	The handshake signals from the Execute stage of the coprocessor pipeline follower. Indicates: ABSENT = 10 WAIT = 00 GO = 01 LAST = 11.

Table A-5 Coprocessor interface signals (continued)

Name	Direction	Description
CPTBIT	Output	When HIGH, indicates that the ARM966E-S processor in is Thumb state. When LOW, indicates that the ARM966E-S processor is in ARM state. Sampled by the coprocessor pipeline follower.
nCPMREQ	Output	When LOW on the rising edge of **CLK** and **CPCLKEN** is HIGH, the instruction on **CPINSTR** must enter the coprocessor pipeline.
nCPTRANS	Output	When LOW, indicates that the ARM966E-S processor is in User mode. When HIGH, indicates that the ARM966E-S processor is in Privileged mode. Sampled by the coprocessor pipeline follower.

A.6 Debug signals

Table A-6 describes the ARM966E-S debug signals.

Table A-6 Debug signals

Name	Direction	Description
DBGIR[3:0]	Output	These four bits reflect the current instruction loaded into the TAP controller control register. These bits change when the TAP controller is in the UPDATE-IR state.
DBGnTRST	Input	This is the active LOW reset signal for the EmbeddedICE internal state. This signal can be asserted asynchronously but must be deasserted synchronously.
DBGnTDOEN	Output	When LOW, this signal denotes that the serial data is being driven out of the **DBGTDO** output. Normally used as an output enable for a **DBGTDO** pin in a packaged part.
DBGSCREG[4:0]	Output	These five bits reflect the ID number of the scan chain currently selected by the TAP controller. These bits change when the TAP controller is in the UPDATE-DR state.
DBGSDIN	Output	Contains the serial data to be applied to an external scan chain.
DBGSDOUT	Input	Contains the serial data out of an external scan chain. When an external scan chain is not connected, this signal must be tied LOW.
DBGTAPSM[3:0]	Output	This bus reflects the current state of the TAP controller state machine.
DBGTDI	Input	Test data input for debug logic.
DBGTDO	Output	Test data output from debug logic.
DBGTMS	Input	Test mode select for TAP controller.
COMMRX	Output	When HIGH, denotes that the communications channel receive buffer contains valid data waiting to be read.
COMMTX	Output	When HIGH, denotes that the communications channel transmit buffer is empty.

Table A-6 Debug signals (continued)

Name	Direction	Description
DBGACK	Output	When HIGH, indicates that the processor is in debug state.
DBGEN	Input	A static configuration signal that disables the debug features of the processor when held LOW. This signal must be HIGH to enable the EmbeddedICE logic.
DBGRQI	Output	Represents the debug request signal that is presented to the core debug logic. This is a combination of **EDBGRQ** and bit [1] of the Debug Control Register.
EDBGRQ	Input	An external debugger forces the processor into debug state by asserting this signal.
DBGEXT[1:0]	Input	Input to the EmbeddedICE-RT logic enables breakpoints/watchpoints to be dependent on external conditions.
DBGINSTREXEC	Output	Indicates that the instruction in the Execute stage of the processor pipeline has been executed.
DBGRNG[1:0]	Output	Indicates that the corresponding EmbeddedICE-RT watchpoint register has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint enable control bit.
TAPID[31:0]	Input	Specifies the ID code value shifted out on **DBGTDO** when the IDCODE instruction is entered into the TAP controller. For more information on TAP ID, see *ARM966E-S JTAG TAP ID* on page 7-25.
DBGIEBKPT	Input	Asserted by external hardware to halt execution of the processor for debug purposes. If HIGH at the end of an instruction fetch, it causes the ARM966E-S processor to enter debug state if that instruction reaches the Execute stage of the processor pipeline.
DBGDEWPT	Input	Asserted by external hardware to halt execution of the processor for debug purposes. If HIGH at the end of a data memory request cycle, it causes the ARM966E-S processor to enter debug state.

A.7 Miscellaneous signals

Table A-7 describes the ARM966E-S miscellaneous signals.

Table A-7 Miscellaneous signals

Name	Direction	Description
nFIQ	Input	This is the Fast Interrupt Request signal. This signal must be synchronous to **CLK**.
nIRQ	Input	This is the Interrupt Request signal. This signal must be synchronous to **CLK**.
VINITHI	Input	Determines the reset location of the exception vectors. When LOW, the vectors are located at `0x00000000`. When HIGH, the vectors are located at `0xFFFF0000`.
INITRAM	Input	Determines the TCM reset enable.When HIGH, the instruction and data TCM are both enabled during reset. When LOW, the TCM are disabled during reset.
BIGENDOUT	Output	When HIGH, the ARM966E-S processor treats bytes in memory as big-endian. When LOW, memory is treated as little-endian.

A.8 ETM interface signals

Table A-8 describes the ARM966E-S ETM interface signals.

Table A-8 ETM interface signals

Name	Direction	Description
ETMEN	Input	Synchronous ETM interface enable. This signal must be tied LOW if there is no ETM.
FIFOFULL	Input	Asserted when ETM FIFO fills. This signal must be tied LOW if there is no ETM.
ETMBIGEND	Output	Big-endian configuration indication for the ETM.
ETMHIVECS	Output	Exception vectors configuration for the ETM.
ETMIA[31:1]	Output	Instruction address for the ETM.
ETMInMREQ	Output	Instruction memory request for the ETM.
ETMISEQ	Output	Sequential instruction access for the ETM.
ETMITBIT	Output	Thumb state indication for the ETM.
ETMDA[31:0]	Output	Data address for the ETM.
ETMDMAS[1:0]	Output	Data size indication for the ETM.
ETMDnMREQ	Output	Data memory request for the ETM.
ETMDnRW	Output	Data not read or write for the ETM.
ETMDSEQ	Output	Sequential data indication for the ETM.
ETMRDATA[31:0]	Output	Read data for the ETM.
ETMWDATA[31:0]	Output	Write data for the ETM.
ETMDABORT	Output	Data Abort for the ETM.
ETMnWAIT	Output	ARM9E-S stalled indication for the ETM.
ETMDBGACK	Output	Debug state indication for the ETM.
ETMINSTREXEC	Output	Instruction execute indication for the ETM.
ETMINSTRVALID	Output	Instruction valid indication for the ETM.
ETMRNGOUT[1:0]	Output	Watchpoint register match indication for the ETM.
ETMID31TO25[31:25]	Output	Instruction data field for the ETM.

Table A-8 ETM interface signals (continued)

Name	Direction	Description
ETMID15TO11[15:11]	Output	Instruction data field for the ETM.
ETMCHSD[1:0]	Output	Coprocessor handshake decode signals for the ETM.
ETMCHSE[1:0]	Output	Coprocessor handshake execute signals for the ETM.
ETMPASS	Output	Coprocessor instruction execute indication for the ETM.
ETMLATECANCEL	Output	Coprocessor late cancel indication for the ETM.
ETMPROCID[31:0]	Output	Process ID for the ETM.
ETMPROCIDWR	Output	Asserted when **ETMPROCID** is written.

A.9 Test wrapper signals

Table A-9 describes the ARM966E-S test wrapper signals.

Table A-9 Test wrapper signals

Name	Direction	Description
WSI	Input	Serial input data for the test wrapper scan chain.
WSO	Output	Serial output data from the test wrapper scan chain.
WSEI	Input	Enables scanning of data through the test wrapper scan chain inputs.
WSEO	Input	Enables scanning of data through the test wrapper scan chain outputs.
MUXINSEL	Input	Selects the test wrapper scan chain as the source for ARM966E-S processor inputs: 1 = test wrapper in INTEST mode 0 = functional mode. If **MUXINSEL** is set to 1, **MUXOUTSEL** must be set to 0.
MUXOUTSEL	Input	Selects the test wrapper scan chain as the source for ARM966E-S processor outputs: 1 = test wrapper in EXTEST mode 0 = functional mode. If **MUXOUTSEL** is set to 1, **MUXINSEL** must be set to 0.
WEDGE	Input	Controls which edge the wrapper chain output activates on: 1 = activate on rising edge 0 = activate on falling edge.
CPMUXINSEL	Input	Enables the ARM966E-S processor to exercise its coprocessors in INTEST mode: 1 = standard INTEST mode 0 = functional mode or coprocessor test mode. If **CPMUXINSEL** is set to 1, then **MUXINSEL** must be set to 1.
TCMMUXINSEL	Input	Enables the ARM966E-S processor to test TCMs using BIST in INTEST mode: 1 = standard INTEST mode 0 = functional mode or TCM test mode. If **TCMMUXINSEL** is set to 1, then **MUXINSEL** must be set to 1.

Appendix B
AC Parameters

This appendix describes the AC timing parameters for the ARM966E-S processor. It contains the following section:

- *Timing diagrams and timing parameters* on page B-2.

B.1 Timing diagrams and timing parameters

All figures are expressed as percentages of the **CLK** period at maximum operating frequency.

The figures quoted are relative to the rising clock edge after the clock skew for internal buffering has been added. Inputs given a 0% hold figure require a positive hold relative to the top-level clock input. The amount of hold required is equivalent to the internal clock skew.

The timing diagrams and timing parameter tables in this section are:

- *Clock, reset and AHB enable timing parameters*
- *AHB bus master timing parameters* on page B-4
- *Coprocessor interface timing parameters* on page B-6
- *Debug interface timing parameters* on page B-8
- *JTAG interface timing parameters* on page B-10
- *Exception and configuration timing parameters* on page B-12
- *AHB bus request and grant related timing parameters* on page B-13
- *INTEST wrapper timing parameters* on page B-14
- *ETM interface timing parameters* on page B-16
- *TCM interface timing parameters* on page B-18.

B.1.1 Clock, reset, and AHB enable timing

Figure B-1 shows the clock, reset, and AHB enable timing parameters.

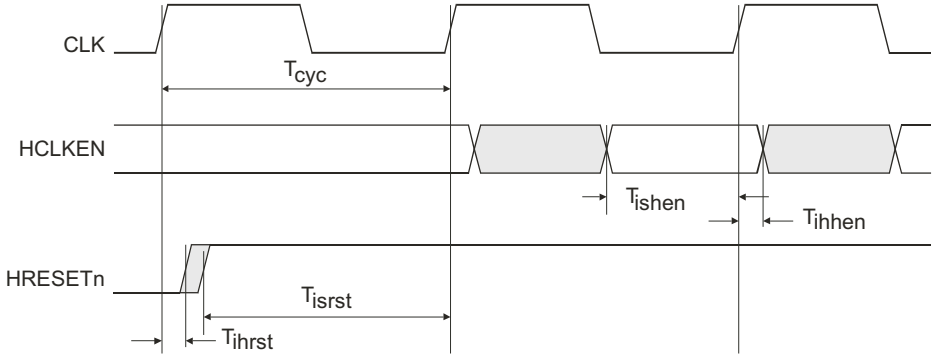


Figure B-1 Clock, reset and AHB enable timing parameters

Table B-1 describes the timing parameters shown in Figure B-1 on page B-2.

Table B-1 Clock, reset and AHB enable parameters

Symbol	Parameter	Min	Max
T_{cyc}	**CLK** cycle time	100%	-
T_{ishen}	**HCLKEN** input setup time to rising **CLK**	85%	-
T_{ihhen}	**HCLKEN** input hold time from rising **CLK**	-	0%
T_{isrst}	**HRESETn** deassertion input setup time to rising **CLK**	90%	-
T_{ihrst}	**HRESETn** deassertion input hold time from rising **CLK**	-	0%

B.1.2 AHB bus master timing

Figure B-2 on page B-4 shows the AHB bus master timing parameters.

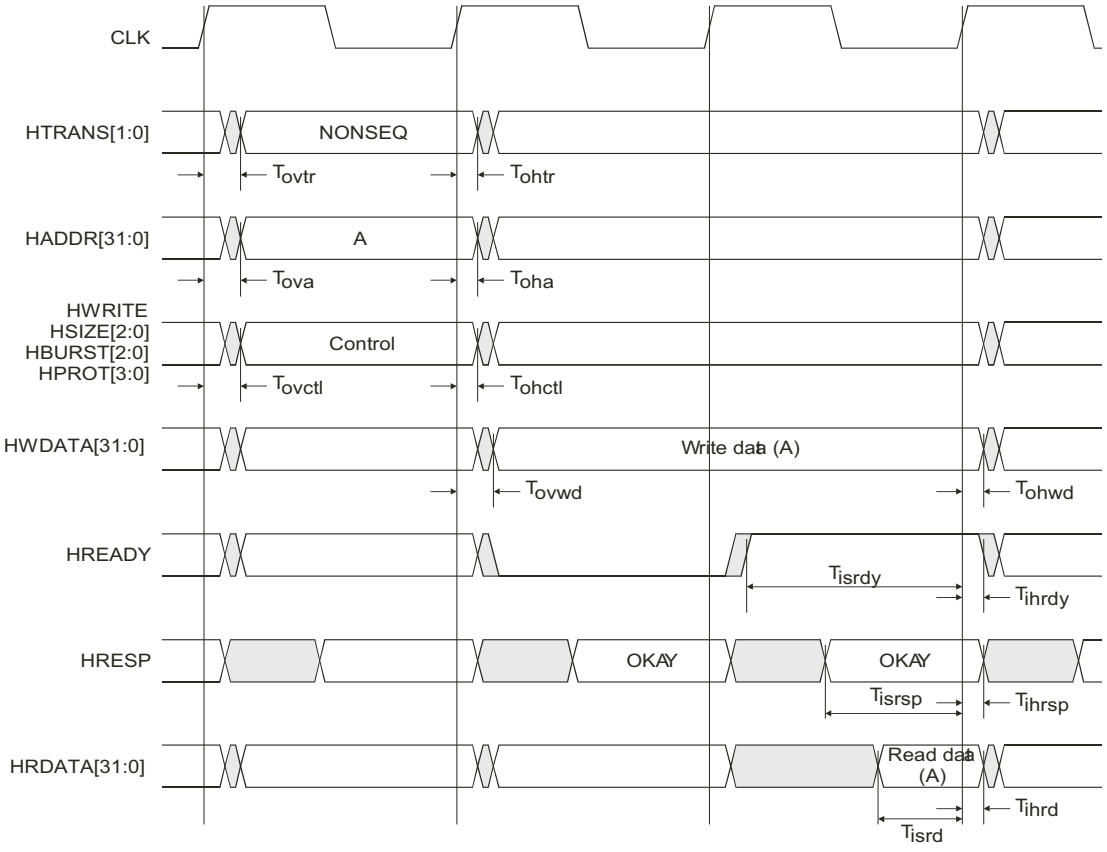


Figure B-2 AHB bus master timing parameters

Table B-2 describes the AHB bus master timing parameters shown in Figure B-2 on page B-4.

Table B-2 AHB bus master timing parameters

Symbol	Parameter	Min	Max
T_{ovtr}	Rising **CLK** to **HTRANS[1:0]** valid	-	30%
T_{ohtr}	**HTRANS[1:0]** hold time from rising **CLK**	>0%	-
T_{ova}	Rising **CLK** to **HADDR[31:0]** valid	-	30%
T_{oha}	**HADDR[31:0]** hold time from rising **CLK**	>0%	-
T_{ovctl}	Rising **CLK** to AHB control signals valid	-	30%
T_{ohctl}	AHB control signals hold time from rising **CLK**	>0%	-
T_{ovwd}	Rising **CLK** to **HWDATA[31:0]** valid	-	30%
T_{ohwd}	**HWDATA[31:0]** hold time from rising **CLK**	>0%	-
T_{isrdy}	**HREADY** input setup time to rising **CLK**	40%	-
T_{ihrdy}	**HREADY** input hold time from rising **CLK**	-	0%
T_{isrsp}	**HRESP[1:0]** input setup time to rising **CLK**	40%	-
T_{ihrsp}	**HRESP[1:0]** input hold time from rising **CLK**	-	0%
T_{isrd}	**HRDATA[31:0]** input setup time to rising **CLK**	30%	-
T_{ihrd}	**HRDATA[31:0]** input hold time from rising **CLK**	-	0%

B.1.3 Coprocessor interface timing

Figure B-3 on page B-6 shows the coprocessor interface timing parameters.

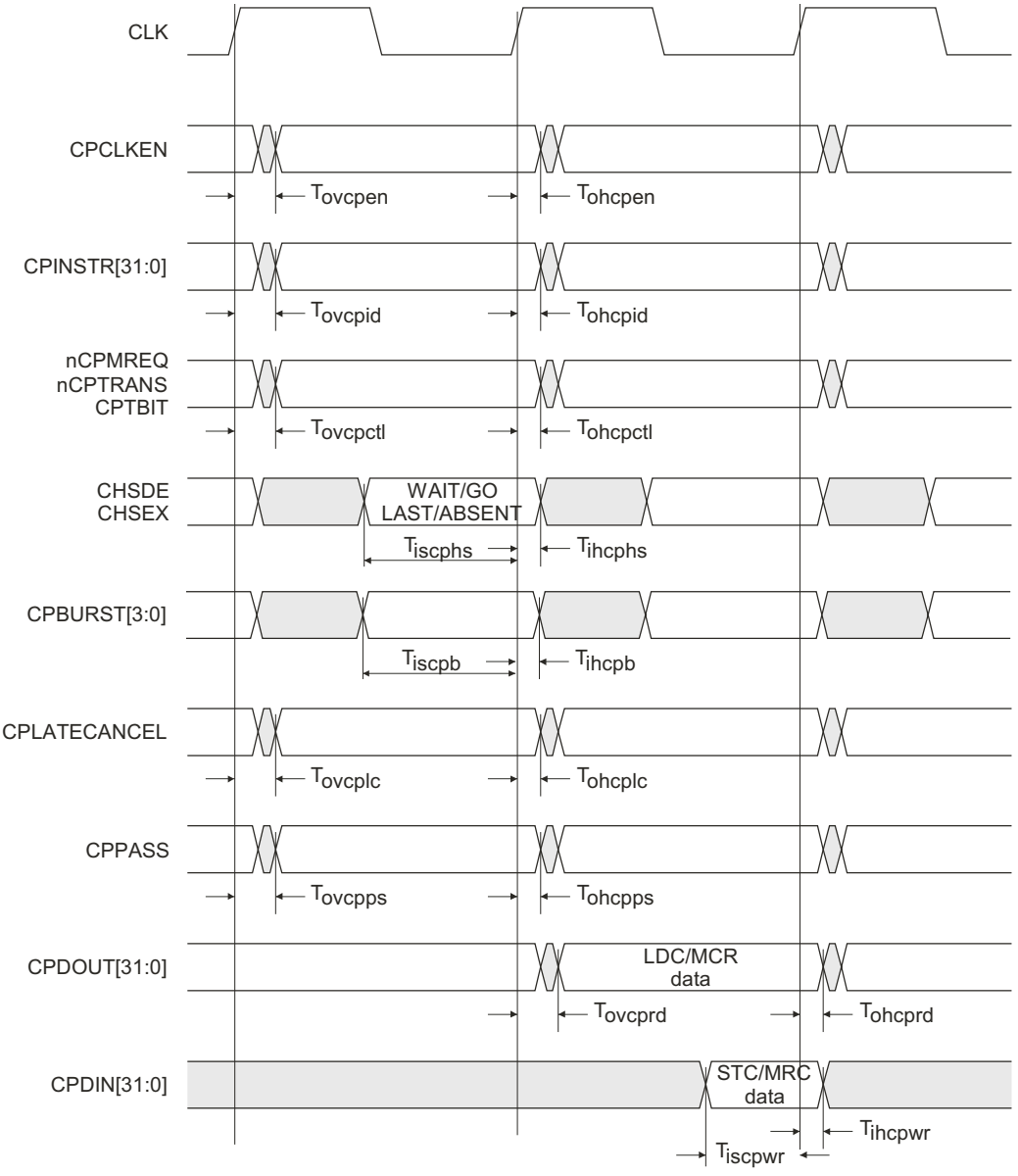


Figure B-3 Coprocessor interface timing parameters

Table B-3 describes the coprocessor interface timing parameters shown in Figure B-3 on page B-6.

Table B-3 Coprocessor interface parameters

Symbol	Parameter	Min	Max
T_{ovcpen}	Rising **CLK** to **CPCLKEN** valid	-	30%
T_{ohcpen}	**CPCLKEN** hold time from rising **CLK**	>0%	-
T_{ovcpid}	Rising **CLK** to **CPINSTR[31:0]** valid	-	30%
T_{ohcpid}	**CPINSTR[31:0]** hold time from rising **CLK**	>0%	-
$T_{ovcpctl}$	Rising **CLK** to transaction control valid	-	30%
$T_{ohcpctl}$	Transaction control hold time from rising **CLK**	>0%	-
T_{iscphs}	Coprocessor handshake input setup time to rising **CLK**	50%	-
T_{ihcphs}	Coprocessor handshake input hold time from rising **CLK**	-	0%
T_{iscpb}	**CPBURST[3:0]** input setup time to rising **CLK**	50%	-
T_{ihcpb}	**CPBURST[3:0]** input hold time from rising **CLK**	-	0%
T_{ovcplc}	Rising **CLK** to **CPLATECANCEL** valid	-	30%
T_{ohcplc}	**CPLATECANCEL** hold time from rising **CLK**	>0%	-
T_{ovcpps}	Rising **CLK** to **CPPASS** valid	-	30%
T_{ohcpps}	**CPPASS** hold time from rising **CLK**	>0%	-
T_{ovcprd}	Rising **CLK** to **CPDOUT[31:0]** valid	-	30%
T_{ohcprd}	**CPDOUT[31:0]** hold time from rising **CLK**	>0%	-
T_{iscpwr}	**CPDIN[31:0]** input setup time to rising **CLK**	50%	-
T_{ihcpwr}	**CPDIN[31:0]** input hold time from rising **CLK**	-	0%

B.1.4 Debug interface timing

Figure B-4 on page B-8 shows the debug interface timing parameters.

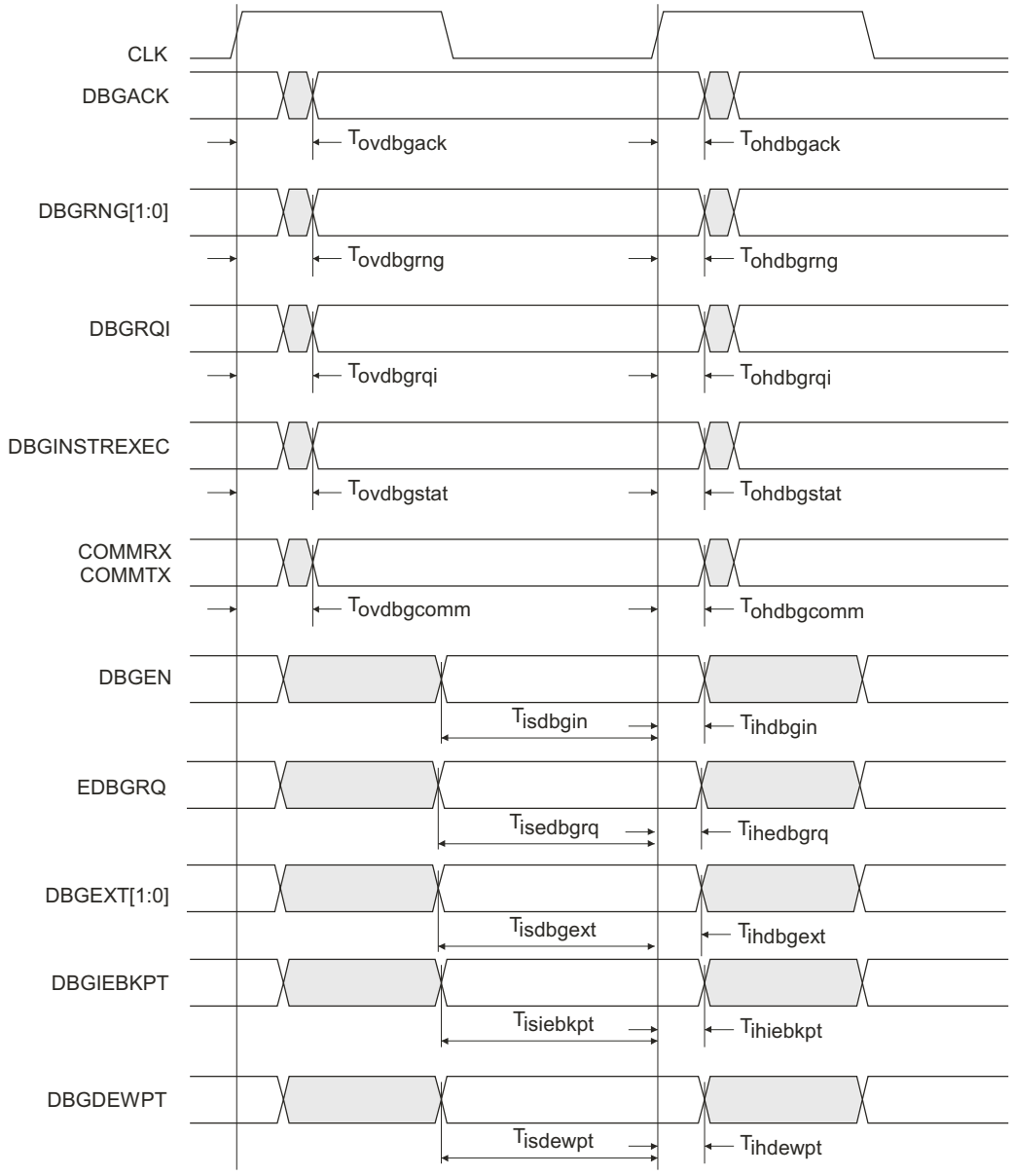


Figure B-4 Debug interface timing parameters

Table B-4 describes the Debug interface timing parameters shown in Figure B-4 on page B-8.

Table B-4 Debug interface parameters

Symbol	Parameter	Min	Max
$T_{ovdbgack}$	Rising **CLK** to **DBGACK** valid	-	60%
$T_{ohdbgack}$	**DBGACK** hold time from rising **CLK**	>0%	-
$T_{ovdbgrng}$	Rising **CLK** to **DBGRNG[1:0]** valid	-	80%
$T_{ohdbgrng}$	**DBGRNG[1:0]** hold time from rising **CLK**	>0%	-
$T_{ovdbgrqi}$	Rising **CLK** to **DBGRQI** valid	-	45%
$T_{ohdbgrqi}$	**DBGRQI** hold time from rising **CLK**	>0%	-
$T_{ovdbgstat}$	Rising **CLK** to **DBGINSTREXEC** valid	-	45%
$T_{ohdbgstat}$	**DBGINSTREXEC** hold time from rising **CLK**	>0%	-
$T_{ovdbgcomm}$	Rising **CLK** to communications channel outputs valid	-	60%
$T_{ohdbgcomm}$	Communications channel outputs hold time from rising **CLK**	>0%	-
$T_{isdbgen}$	**DBGEN** input setup time to rising **CLK**	35%	-
$T_{ihdbgen}$	**DBGEN** input hold time from rising **CLK**	-	0%
$T_{isedbgrq}$	**EDBGRQ** input setup hold time to rising **CLK**	30%	-
$T_{ihedbgrq}$	**EDBGRQ** input hold time from rising **CLK**	-	0%
$T_{isdbgext}$	**DBGEXT** input setup time to rising **CLK**	20%	
$T_{ihdbgext}$	**DBGEXT** input hold time from rising **CLK**	-	0%
$T_{isiebkpt}$	**DBGIEBKPT** input setup time to rising **CLK**	50%	-
$T_{ihiebkpt}$	**DBGIEBKPT** input hold time from rising **CLK**	-	0%
$T_{isdewpt}$	**DBGDEWPT** input setup time to rising **CLK**	50%	-
$T_{ihdewpt}$	**DBGDEWPT** input hold time from rising **CLK**	-	0%

B.1.5 JTAG interface timing

Figure B-5 on page B-10 shows the JTAG interface timing parameters.

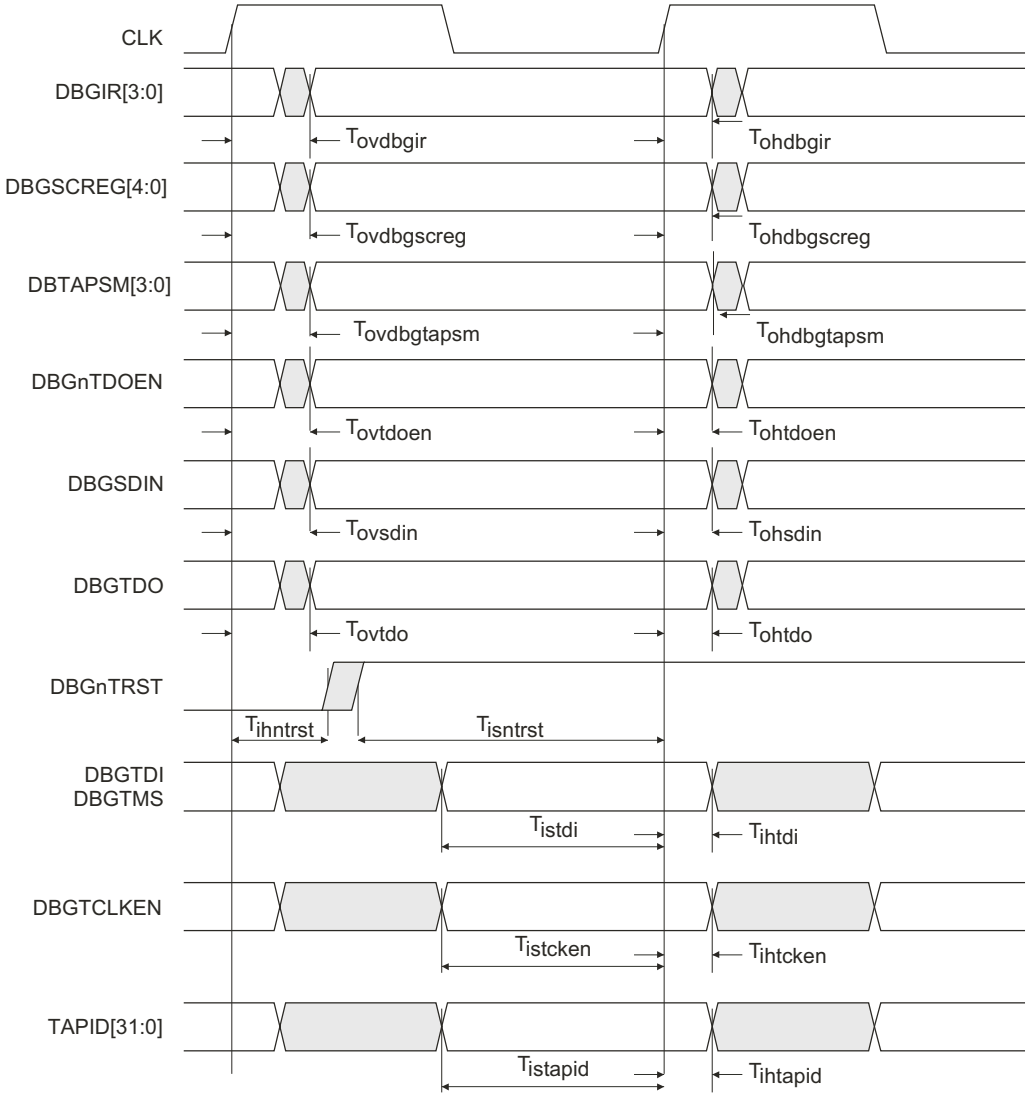


Figure B-5 JTAG interface timing parameters

Table B-5 describes the JTAG interface timing parameters as shown in Figure B-5 on page B-10.

Table B-5 JTAG interface parameters

Symbol	Parameter	Min	Max
$T_{ovdbgir}$	Rising **CLK** to **DBGIR** valid	-	25%
$T_{ohdbgir}$	**DBGIR** hold time from rising **CLK**	>0%	-
$T_{ovdbgscreg}$	Rising **CLK** to **DBGSCREG** valid	-	30%
$T_{ohdbgscreg}$	**DBGSCREG** hold time from rising **CLK**	>0%	-
$T_{ovdbgtapsm}$	Rising **CLK** to **DBGTAPSM** valid	-	30%
$T_{ohdbgtapsm}$	**DBGTAPSM** hold time from rising **CLK**	>0%	-
$T_{ovtdoen}$	Rising **CLK** to **DBGnTDOEN** valid	-	40%
$T_{ohtdoen}$	**DBGnTDOEN** hold time from rising **CLK**	>0%	-
T_{ovsdin}	Rising **CLK** to **DBGSDIN** valid	-	25%
T_{ohsdin}	**DBGSDIN** hold time from rising **CLK**	>0%	-
T_{ovtdo}	Rising **CLK** to **DBGTDO** valid	-	65%
T_{ohtdo}	**DBGTDO** hold time from rising **CLK**	>0%	-
$T_{isntrst}$	**DBGnTRST** deasserted input setup time to rising **CLK**	25%	-
$T_{ihntrst}$	**DBGnTRST** input hold time from rising **CLK**	-	0%
T_{istdi}	TAP state control input setup time to rising **CLK**	30%	-
T_{ihtdi}	TAP state control input hold time from rising **CLK**	-	0%
$T_{istcken}$	**DBGTCKEN** input setup time to rising **CLK**	50%	-
$T_{ihtcken}$	**DBGTCKEN** input hold time from rising **CLK**	-	0%
$T_{istapid}$	**TAPID[31:0]** input setup time to rising **CLK**	35%	-
$T_{ihtapid}$	**TAPID[31:0]** input hold time from rising **CLK**	-	0%

B.1.6 Exception and configuration timing

Figure B-6 on page B-12 shows the exception and configuration timing parameters.

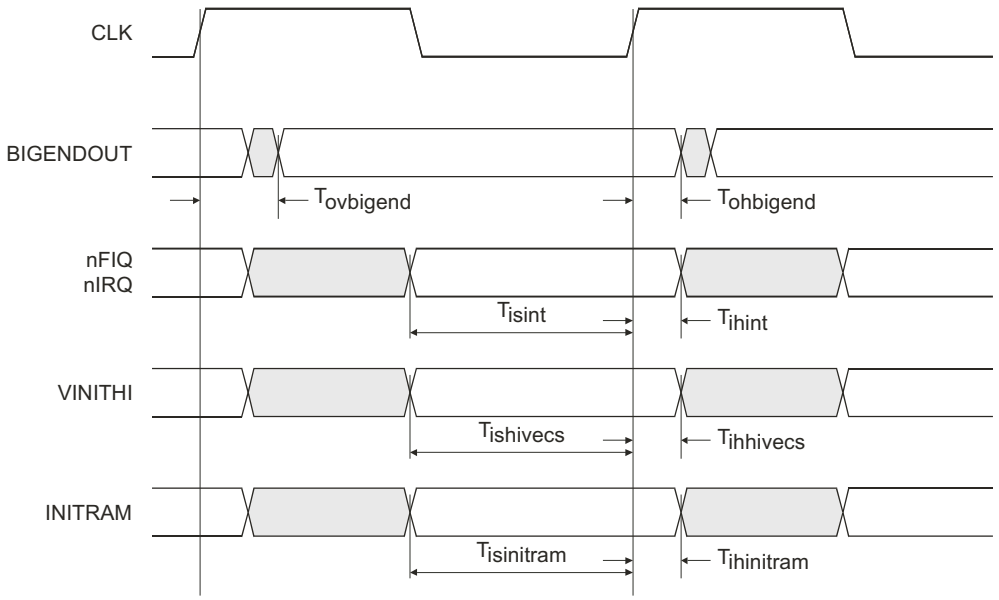


Figure B-6 Exception and configuration timing parameters

Table B-6 describes the exception and configuration timing parameters shown in Figure B-6.

Table B-6 Exception and configuration parameters

Symbol	Parameter	Min	Max
$T_{ovbigend}$	Rising **CLK** to **BIGENDOUT** valid	-	30%
$T_{ohbigend}$	**BIGENDOUT** hold time from rising **CLK**	>0%	-
T_{isint}	Interrupt input setup time to rising **CLK**	30%	-
T_{ihint}	Interrupt input hold time from rising **CLK**	-	0%
$T_{ishivecs}$	**VINITHI** input setup time to rising **CLK**	90%	-
$T_{ihhivecs}$	**VINITHI** input hold time from rising **CLK**	-	0%
$T_{isinitram}$	**INITRAM** input setup time to rising **CLK**	95%	-
$T_{ihinitram}$	**INITRAM** input hold time from rising **CLK**	-	0%

The **VINITHI** and **INITRAM** pins are specified as 95% of the cycle because they are for input configuration during reset and can be considered static.

B.1.7 AHB bus request and grant-related timing

Figure B-7 shows the AHB bus request and grant-related timing parameters.

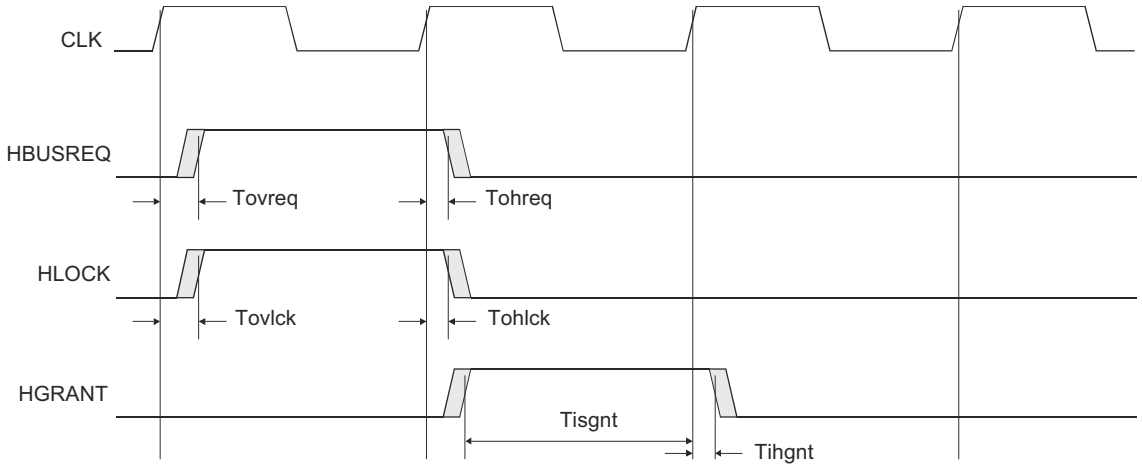


Figure B-7 AHB bus request and grant related timing parameters

Table B-7 describes the AHB bus request and grant-related timing parameters shown in Figure B-7.

Table B-7 AHB bus request and grant-related parameters

Symbol	Parameter	Min	Max
T_{ovreq}	Rising **CLK** to **HBUSREQ** valid	-	30%
T_{ohreq}	**HBUSREQ** hold time from rising **CLK**	>0%	-
T_{ovlck}	Rising **CLK** to **HLOCK** valid	-	30%

Table B-7 AHB bus request and grant-related parameters (continued)

Symbol	Parameter	Min	Max
T_{ohlck}	**HLOCK** hold time from rising **CLK**	>0%	-
T_{isgnt}	**HGRANT** input setup time to rising **CLK**	40%	-
T_{ihgnt}	**HGRANT** input hold time from rising **CLK**	-	0%

B.1.8 INTEST wrapper timing

Figure B-8 shows the INTEST wrapper timing parameters. The INTEST wrapper inputs and outputs are specified as 95% of the cycle because they are production test related and expected to operate at typically 50% of the functional clock rate.

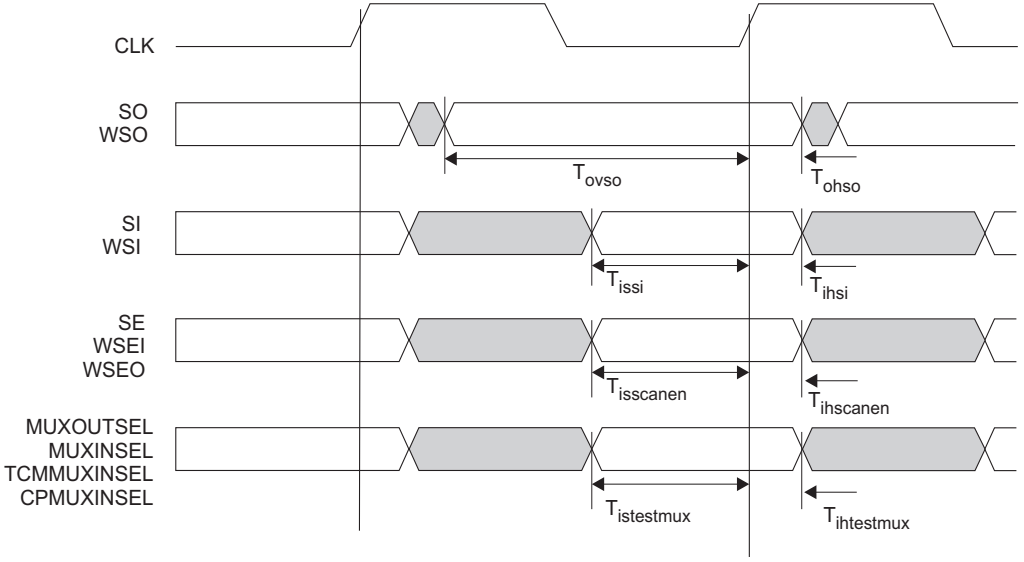


Figure B-8 INTEST wrapper timing parameters

Table B-8 describes the INTEST wrapper parameters shown in Figure B-8 on page B-14.

Table B-8 INTEST wrapper

Symbol	Parameter	Min	Max
T_{ovso}	Rising **CLK** to **SO** valid	-	30%
T_{ohso}	**SO** hold time from rising **CLK**	>0%	-
T_{issi}	**SI** input setup time to rising **CLK**	95%	-
T_{ihsi}	**SI** input hold time from rising **CLK**	-	0%
$T_{isscanen}$	**SCANEN** input setup time to rising **CLK**	95%	-
$T_{ihscanen}$	**SCANEN** input hold time from rising **CLK**	-	0%
$T_{istestmux}$	Test mux input setup time to rising **CLK**	95%	-
$T_{ihtestmux}$	Test mux input hold time from rising **CLK**	-	0%

B.1.9 ETM interface timing

Figure B-9 on page B-16 shows the ETM interface timing parameters.

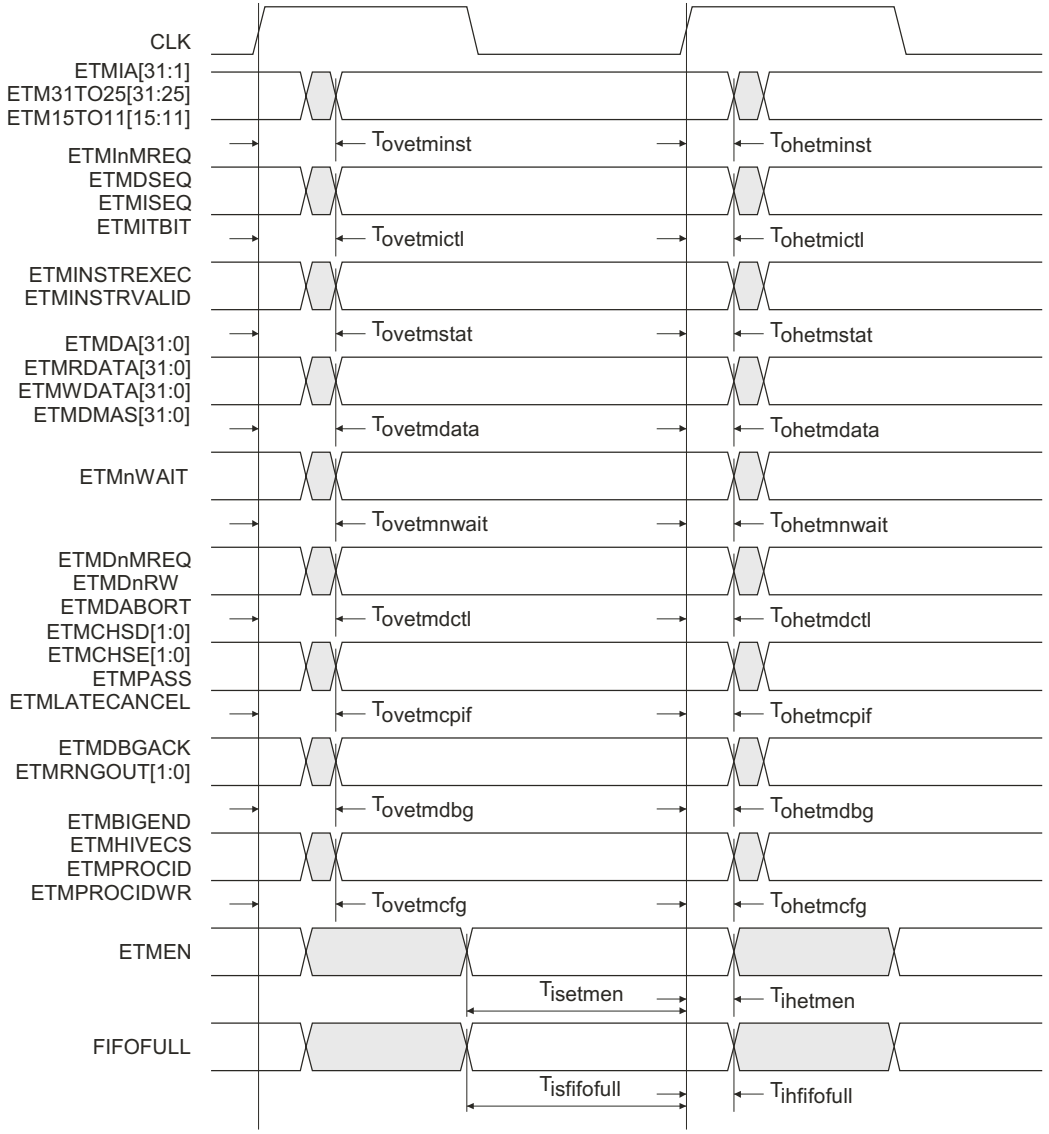


Figure B-9 ETM interface timing parameters

Table B-9 describes the ETM timing parameters shown in Figure B-9 on page B-16.

Table B-9 ETM parameters

Symbol	Parameter	Min	Max
$T_{ovetminst}$	Rising **CLK** to ETM instruction interface valid	-	30%
$T_{ohetminst}$	ETM instruction interface hold time from rising **CLK**	>0%	-
$T_{ovetmictl}$	Rising **CLK** to ETM instruction control valid	-	30%
$T_{ohetmictl}$	ETM instruction control hold time from rising **CLK**	>0%	-
$T_{ovetmstat}$	Rising **CLK** to **ETMINSTREXEC** valid	-	30%
$T_{ohetmstat}$	**ETMINSTREXEC** hold time from rising **CLK**	>0%	-
$T_{ovetmdata}$	Rising **CLK** to ETM data interface valid	-	30%
$T_{ohetmdata}$	ETM data interface hold time from rising **CLK**	>0%	-
$T_{ovetmnwait}$	Rising **CLK** to **ETMnWAIT** valid	-	30%
$T_{ohetmnwait}$	**ETMnWAIT** hold time from rising **CLK**	>0%	-
$T_{ovetmdctl}$	Rising **CLK** to ETM data control valid	-	30%
$T_{ohetmdctl}$	ETM data control hold time from rising **CLK**	>0%	-
$T_{ovetmcfg}$	Rising **CLK** to ETM configuration valid	-	30%
$T_{ohetmcfg}$	ETM configuration hold time from rising **CLK**	>0%	-
$T_{ovetmcpif}$	Rising **CLK** to ETM coprocessor signals valid	-	30%
$T_{ohetmcpif}$	ETM coprocessor signals hold time from rising **CLK**	>0%	-
$T_{ovetmdbg}$	Rising **CLK** to ETM debug signals valid	-	30%
$T_{ohetmdbg}$	ETM debug signals hold time from rising **CLK**	>0%	-
$T_{isetmen}$	**ETMEN** input setup time to rising **CLK**	50%	-
$T_{ihetmen}$	**ETMEN** input hold time from rising **CLK**	-	0%
$T_{isfifofull}$	**FIFOFULL** input setup time to rising **CLK**	50%	-
$T_{ihfifofull}$	**FIFOFULL** input hold time from rising **CLK**	-	0%

B.1.10 TCM interface timing

Figure B-10 shows the TCM interface timing parameters.

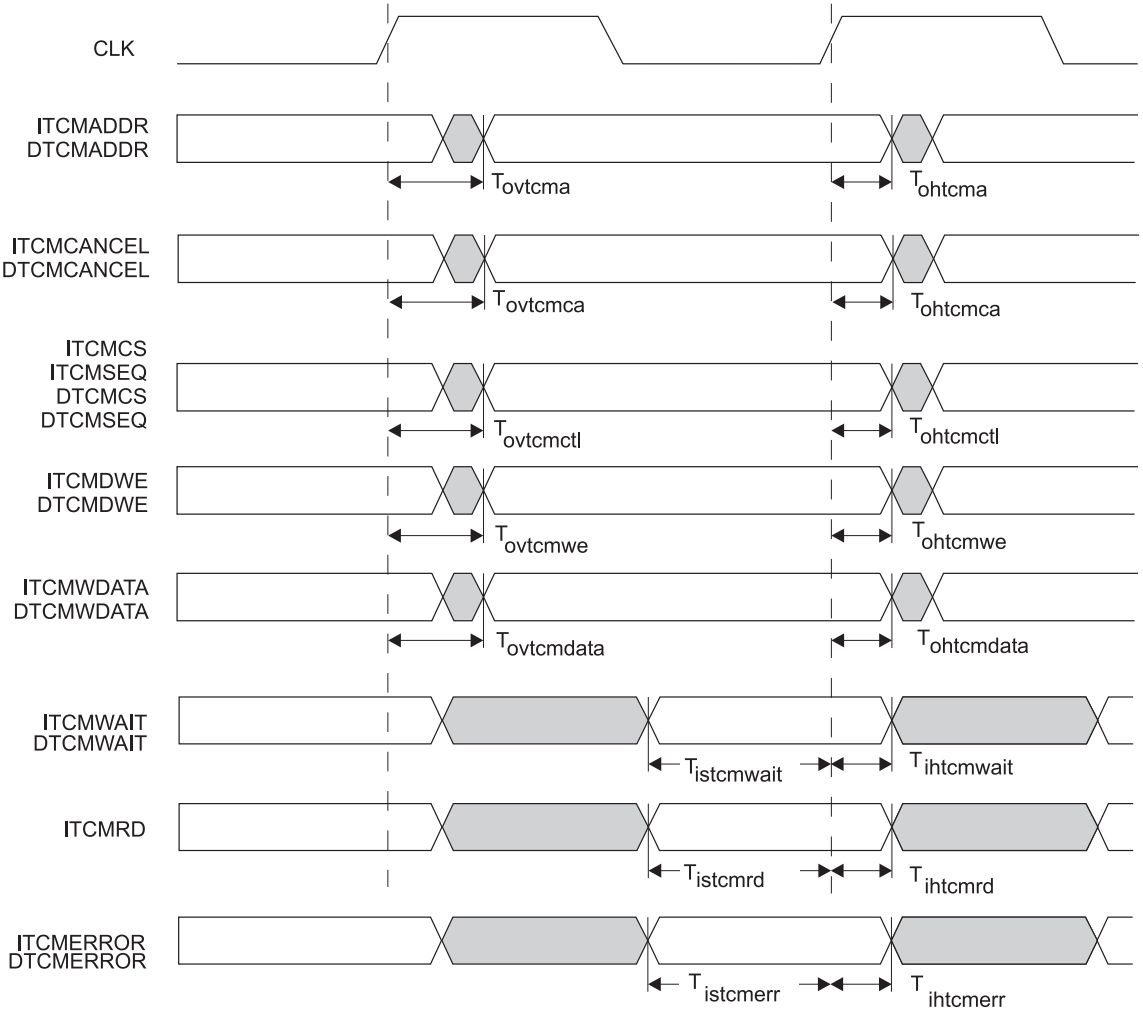


Figure B-10 TCM interface timing parameters

Table B-10 describes the TCM interface timing parameters shown in Figure B-10 on page B-18.

Table B-10 TCM parameters

Symbol	Parameter	Min	Max
T_{ovtcma}	Rising **CLK** to **TCMADDR** valid	-	85%
T_{ohtcma}	**TCMADDR** hold time from rising **CLK**	>0%	-
$T_{ovtcmca}$	Rising **CLK** to **TCMCANCEL** valid	-	75%
$T_{ohtcmca}$	**TCMCANCEL** hold time from rising **CLK**	>0%	-
$T_{ovtcmctl}$	Rising **CLK** to TCM data control valid	-	85%
$T_{ohtcmctl}$	TCM data control valid hold time from rising **CLK**	>0%	-
$T_{ovtcmwe}$	Rising **CLK** to TCM write enable valid	-	85%
$T_{ohtcmwe}$	TCM write enable hold time from rising **CLK**	>0%	-
$T_{ovtcmdata}$	Rising **CLK** to TCM write data valid	-	50%
$T_{ohtcmdata}$	TCM write data hold time from rising **CLK**	>0%	-
$T_{istcmwait}$	**TCMWAIT** input setup time to rising **CLK**	15%	-
$T_{ihtcmwait}$	**TCMWAIT** input hold time from rising **CLK**	-	0%
$T_{istcmrd}$	TCM read data input setup time to rising **CLK**	40%	-
$T_{ihtcmrd}$	TCM read data input hold time from rising **CLK**	-	0%
$T_{istcmerr}$	**TCMERROR** input setup time to rising **CLK**	40%	-
$T_{ihtcmerr}$	**TCMERROR** input hold time from rising **CLK**	-	0%

Glossary

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

Abort A mechanism that indicates to a core that it must halt execution of an attempted illegal memory access. 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 Abort, a Data Abort, or an External Abort.

See also Data Abort, External Abort and Prefetch Abort.

Abort model An abort model is the defined behavior of an ARM processor in response to a Data Abort exception. Different abort models behave differently with regard to load and store instructions that specify base register write-back.

Advanced Microcontroller Bus Architecture (AMBA)
AMBA is the ARM open standard for multi-master on-chip buses, capable of running with multiple masters and slaves. It is an on-chip bus specification that details a strategy for the interconnection and management of functional blocks that make up a System-on-Chip (SoC). It aids in the development of embedded processors with one or more CPUs or signal processors and multiple peripherals. AMBA complements a reusable design methodology by defining a common backbone for SoC modules. AHB conforms to this standard.

AMBA *See* Advanced Microcontroller Bus Architecture.

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.

ARM instruction
Is a word that specifies an operation for an ARM processor to perform. ARM instructions must be word-aligned.

ARM state
A processor that is executing ARM (32-bit) word-aligned instructions is operating in ARM state.

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.

BIST
See Built-In Self Test.

Built-in Self Test
The technique of designing circuits with additional logic that can be used to test proper operation of the primary (functional) logic. This is used only for TCMs in the ARM966E-S macrocell.

Breakpoint
A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, 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
An 8-bit data item.

Communications channel
The hardware used for communicating between the software running on the processor, and an external host, using the debug interface. When this communication is for debug purposes, it is called the Debug Comms Channel.

Coprocessor
A processor that supplements the main CPU. It carries out additional functions that the main CPU cannot perform. Usually used for floating-point math calculations, signal processing, or memory management.

Data Abort
An indication from a memory system to a core that it must halt execution of an attempted illegal memory access. A Data Abort is caused by attempting to access invalid data memory.

See also Abort, External Abort, and Prefetch Abort.

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

Doubleword A 64-bit data item. The contents are taken as being an unsigned integer unless otherwise stated.

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.

Embedded Trace Macrocell (ETM)
A hardware macrocell which, 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.

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.

Exception 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 service routine.

ETM See *Embedded Trace Macrocell*.

External Abort An indication from an external memory system to a core that it must halt execution of an attempted illegal memory access. 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.

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.

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.

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.

Memory coherency A memory is coherent if the value read by a data read or instruction fetch is the value that was most recently written to that location. Memory coherency is made difficult when there are multiple possible physical locations that are involved, such as a system that has main memory, a write buffer and a cache.

Prefetch Abort An indication from a memory system to a core that it must halt execution of an attempted illegal memory access. 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.

Processor A contraction of microprocessor. A processor includes the CPU or core, plus additional components such as memory, and interfaces. These are combined as a single macrocell, that can be fabricated on an integrated circuit.

Region A partition of instruction or data memory space.

Register A temporary storage location used to hold binary data until it is ready to be used.

SBO *See* Should Be One.

SBZ *See* Should Be Zero.

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

Should Be One (SBO)

Should be written as 1 (or all 1s for bit fields) by software. Writing a 0 produces Unpredictable results.

Should Be Zero (SBZ)

Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 produces Unpredictable results.

TAP See Test access port.

TCM *See* Tightly-coupled memory.

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**.

Thumb instruction A halfword that specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

Thumb state A processor that is executing Thumb (16-bit) halfword aligned instructions is operating in Thumb state.

Tightly-coupled memory An area of low latency memory that provides predictable instruction execution or data load timing in cases where deterministic performance is required. TCMs are suited to holding: - critical routines (such as for interrupt handling) - scratchpad data - data types whose locality is not suited to caching - critical data structures (such as interrupt stacks).

Instructions and data can be shifted into and out of TCMs using loads and stores.

Undefined Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architecture Reference Manual* for more details on ARM exceptions.

UNP *See* Unpredictable.

Unpredictable For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. Unpredictable instructions must not halt or hang the processor, or any part of the system.

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 allow 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.

Word A 32-bit data item.

Index

C

D

E

F

G

H

I

J

L

M

N

P

R

S

T

V

W

Z

