ARM946E-S™

Revision: r1p1

Technical Reference Manual

ARM DDI 0201D

ARM946E-S
Technical Reference Manual

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Confidentiality	Change
16 February 2001	A	Non-Confidential	First release
17 May 2002	B	Non-Confidential	Second release
15 May 2003	C	Non-Confidential	Third release
17 April 2007	D	Non-Confidential	Various defects corrected

Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

`http://www.arm.com`

Contents

ARM946E-S Technical Reference Manual

List of Tables
ARM946E-S Technical Reference Manual

List of Figures
ARM946E-S Technical Reference Manual

Preface

This preface introduces the *ARM946E-S r1p1 Technical Reference Manual*. It contains the following sections:

- *About this document* on page xvi
- *Feedback* on page xxi.

About this document

This document is a reference manual for the ARM946E-S processor.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this guide, 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 document has been written for hardware and software engineers who want to design or develop products based on the ARM946E-S processor. It assumes no prior knowledge of ARM products.

Using this manual

This manual is organized into the following chapters:

Chapter 1 *Introduction*

This chapter provides an introduction to the ARM946E-S processor.

Chapter 2 *Programmer's Model*

This chapter describes the programmer's model of the ARM946E-S processor and includes a summary of the ARM946E-S coprocessor registers.

Chapter 3 *Caches*

This chapter describes the ARM946E-S cache implementation.

Chapter 4 *Protection Unit*

This chapter describes the ARM946E-S memory protection unit.

Chapter 5 *Tightly-Coupled Memory Interface*

This chapter describes the requirements and operation of the *Tightly-Coupled Memory* (TCM).

Chapter 6 *Bus Interface Unit and Write Buffer*

This chapter describes the operation of the bus interface unit and write buffer.

Chapter 7 ***Coprocessor Interface***

This chapter describes the coprocessor interface and the operation of common coprocessor instructions.

Chapter 8 ***ETM Interface***

This chapter describes the ETM interface, including details of how to enable the interface.

Chapter 9 ***Debug Support***

This chapter describes the debug support for the ARM946E-S processor and the EmbeddedICE-RT logic.

Chapter 10 ***Test Support***

This chapter describes the test methodology used for the ARM946E-S synthesized logic and memory.

Appendix A ***AC Parameters***

This appendix describes the timing parameters applicable to the ARM946E-S processor.

Appendix B ***Signal Descriptions***

This appendix describes the signals used in the ARM946E-S processor.

Conventions

Conventions that this manual can use are described in:

- *Typographical*
- *Timing diagrams* on page xviii
- *Signals* on page xix
- *Numbering* on page xix.

Typographical

The typographical conventions are:

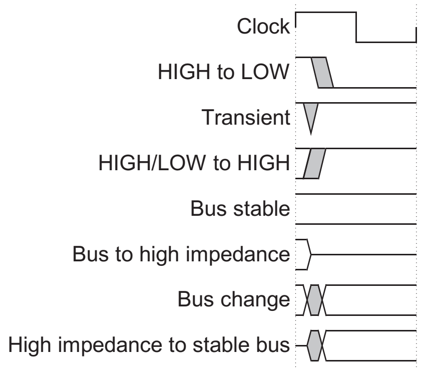
italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace`	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.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.



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	Denotes AXI, AHB, and APB 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 from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

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

ARM publications

This document contains information that is specific to the ARM946E-S processor. See the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0406)

- *ARM9E-S Technical Reference Manual* (ARM DDI 0165)
- *ARM946E-S Configuration and Sign-off Guide* (ARM DII 0171)
- *AMBA Specification* (ARM IHI 0011).

Feedback

ARM Limited welcomes feedback both on the ARM946E-S processor and its documentation.

Feedback on the ARM946E-S processor

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 manual, send email to `errata@arm.com` giving:

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

ARM Limited also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the ARM946E-S processor. It contains the following sections:

- *About the ARM946E-S processor* on page 1-2
- *ARM946E-S block diagram* on page 1-3
- *Differences between processor versions* on page 1-5.

1.1 About the ARM946E-S processor

The ARM946E-S is a synthesizable processor combining an ARM9E-S™ processor core with a configurable memory system. It is a member of the ARM9E™ family of high-performance, 32-bit system-on-chip processor solutions.

The ARM946E-S is a Harvard architecture cached processor that provides a complete high-performance processor subsystem, including:

- An ARM9E-S RISC integer CPU core featuring:
  - ARMv5TE 32-bit instruction set that has improved ARM/Thumb code interworking and an 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 *Tightly-Coupled Memory* (TCM). A TCM interface is provided for each of the external instruction and data memory blocks. The size of both the Instruction and Data TCM blocks are implementor-specific and can range from 0KB to 1MB.
- Instruction and data caches. The design can be easily modified to enable any combination of caches from 0KB to 1MB.
- A protection unit that enables the memory to be protected in a simple manner, ideal for embedded control applications.
- An AMBA AHB bus interface. The ARM946E-S processor interfaces to the rest of the system are through use of unified address and data buses. This interface is compatible with the AMBA AHB bus standard.
- Support for external coprocessors enabling floating point or other application specific hardware acceleration to be added. For coprocessor support, the instruction and data buses are exported along with simple handshaking signals.
- Support for the use of a scan test methodology for the standard cell logic and *Built-In-Self-Test* (BIST) for the TCM and caches.
- An interface to an external *Embedded Trace Macrocell* (ETM) to support real-time tracing of instructions and data.

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

The ARM946E-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.

1.2 ARM946E-S block diagram

Figure 1-1 shows a block diagram of the ARM946E-S processor.

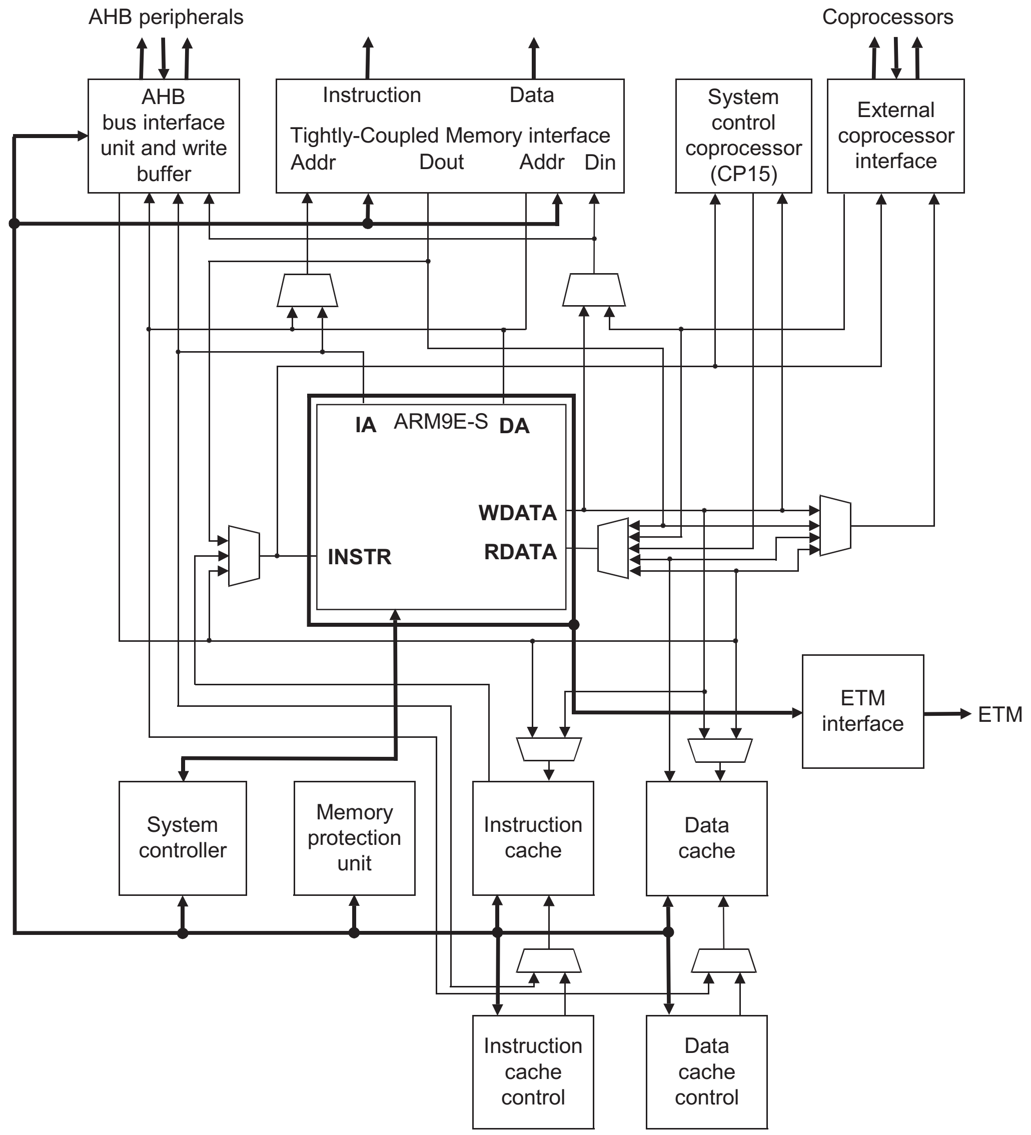


Figure 1-1 ARM946E-S processor block diagram

The blocks shown in Figure 1-1 on page 1-3 are described in the locations listed in Table 1-1.

Table 1-1 Location of block descriptions

Block	Location of description
ARM9E-S (Rev 1)	*ARM9E-S (Rev 1) Technical Reference Manual*
AHB bus interface unit and write buffer	Chapter 6 *Bus Interface Unit and Write Buffer*
Tightly-coupled Memory interface	Chapter 5 *Tightly-Coupled Memory Interface*
System control coprocessor (CP15)	Chapter 2 *Programmer's Model*
External coprocessor interface	Chapter 7 *Coprocessor Interface*
ETM interface	Chapter 8 *ETM Interface*
System controller	Chapter 2 *Programmer's Model*
Memory protection unit	Chapter 4 *Protection Unit*
Instruction cache	Chapter 3 *Caches*
Data cache	Chapter 3 *Caches*
Instruction cache control	Chapter 2 *Programmer's Model* and Chapter 3 *Caches*
Data cache control	Chapter 2 *Programmer's Model* and Chapter 3 *Caches*

1.3 Differences between processor versions

The differences between the current version of the processor and Rev 0 are as follows:

- Tightly coupled memories are external to the macrocell. See Chapter 5 *Tightly-Coupled Memory Interface*.
- The AHB interface is changed to improve the input timing. See Chapter 6 *Bus Interface Unit and Write Buffer*.
- The AHB interface is changed to generate BUSY cycles during access types. See Chapter 5 *Tightly-Coupled Memory Interface*.

Chapter 2
Programmer's Model

This chapter describes the programmer's model for the ARM946E-S processor. It contains the following sections:

2.1 About the ARM946E-S programmer's model

The programmer's model for the ARM946E-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 ARM946E-S internal coprocessors, and any coprocessor connected to the external coprocessor interface.

There are two internal coprocessors within the ARM946E-S processor:

- CP14 within the ARM9E-S core enables software access to the debug communication channel
- CP15 enables configuration of the caches, *Tightly-Coupled Memory* (TCM), protection unit, write buffer, and other ARM946E-S system options such as big or little-endian operation.

The registers defined in CP14 are accessible with `MCR` and `MRC` instructions, and are described in *The debug communication channel* on page 9-29.

The registers defined in CP15 are accessible with `MCR` and `MRC` instructions, and are described in *CP15 register map summary* on page 2-4. These instructions permit conditional access using the optional {cond} field.

Registers and operations provided by any coprocessors 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, which 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 implements the base restored Data Abort model, which 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 contains before the instruction is executed. This removes the requirement for the Data Abort handler to repair 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.

2.3 CP15 register map summary

The ARM946E-S processor incorporates CP15 for system control. CP15 enables configuration of the caches, *Tightly-Coupled Memory* (TCM), and protection unit. It also enables configuration of the ARM946E-S system options including big or little-endian operation.

This section contains the following:

- *Accessing CP15 registers* on page 2-6
- *Register 0, ID Code Register* on page 2-7
- *Register 0, Cache Type Register* on page 2-7
- *Register 0, Tightly-coupled Memory Size Register* on page 2-10
- *Register 1, Control Register* on page 2-11
- *Register 2, Cache Configuration Registers* on page 2-15
- *Register 3, Write Buffer Control Register* on page 2-16
- *Register 5, Access Permission Registers* on page 2-17
- *Register 6, Protection Region Base and Size Registers* on page 2-19
- *Register 7, Cache Operations Register* on page 2-22
- *Register 9, Cache Lockdown Registers* on page 2-25
- *Register 9, Tightly-coupled Memory Region Registers* on page 2-26.
- *Register 13, Trace Process Identifier Register* on page 2-28
- *Register 15, BIST Control Registers* on page 2-29
- *Register 15, Test State Register* on page 2-31
- *Register 15, Cache Debug Index Register* on page 2-32
- *Register 15, Trace Control Register* on page 2-35.

Table 2-1 shows the register map for CP15.

Table 2-1 CP15 register map

Register	Read	Write
0	ID code [a]	Unpredictable
0	Cache type [a]	Unpredictable
0	Tightly-coupled memory size [a]	Unpredictable
1	Control	Control
2	Cache configuration [b]	Cache configuration [b]
3	Write buffer control	Write buffer control

Table 2-1 CP15 register map (continued)

Register	Read	Write
4	Unpredictable	Unpredictable
5	Access permission [b]	Access permission [b]
6	Protection region base and size [a]	Protection region base and size [a]
7	Unpredictable	Cache operations [a]
8	Unpredictable	Unpredictable
9	Cache lockdown [ab]	Cache lockdown [ab]
9	Tightly-coupled memory region [ab]	Tightly-coupled memory region [ab]
10	Unpredictable	Unpredictable
11	Unpredictable	Unpredictable
12	Unpredictable	Unpredictable
13	Trace process ID	Trace process ID
14	Unpredictable	Unpredictable
15	BIST control [a]	BIST control [a]
15	Test state [a]	Test state [a]
15	Cache debug index [a]	Cache debug index [a]
15	Trace control [a]	Trace control [a]

a. Register location provides access to more than one register. The register accessed depends on the value of the `opcode_2` or `CRm` field. See the register description for details.
b. Separate registers for instruction and data. See the register description for details.

2.3.1 Accessing CP15 registers

Table 2-2 shows the terms and abbreviations used in this section.

Table 2-2 CP15 terms and abbreviations

Term	Abbreviation	Description
Unpredictable	UNP	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.
Undefined	UND	An instruction that accesses CP15 in the manner indicated takes the undefined instruction trap.
Should Be Zero	SBZ	When writing to this location, all bits of this field should be 0.
Should Be One	SBO	When writing to this location, all bits of this field should be 1.

In all cases, reading from, or writing any data values to any CP15 registers, including those fields specified as Unpredictable or Should Be Zero, does not cause any permanent damage.

All CP15 register bits that are defined and contain state are set to 0 by **HRESETn** unless otherwise stated in this chapter.

CP15 registers can only be accessed with `MRC` and `MCR` instructions in a privileged mode. The instruction bit pattern of the `MCR` and `MRC` instructions is shown in Figure 2-1.

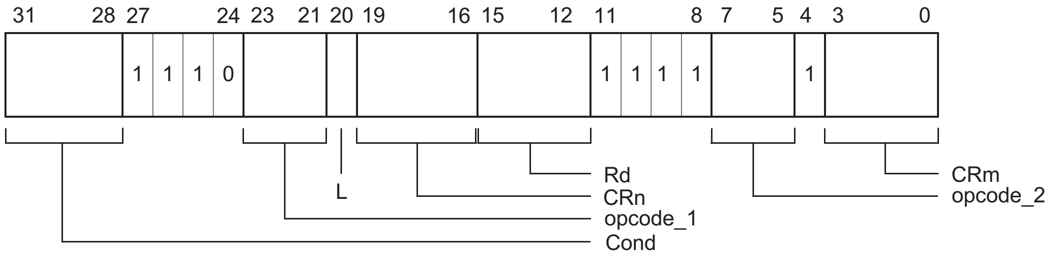


Figure 2-1 CP15 MRC and MCR bit pattern

The assembler for these instructions is:

```
MCR/MRC{cond} p15,opcode_1,Rd,CRn,CRm,opcode_2
```

Instructions CDP, LDC, and STC, along with unprivileged MRC and MCR instructions to CP15, cause the Undefined instruction trap to be taken. The CRn field of MRC and MCR instructions specifies the coprocessor register to access. The CRm field and opcode_2 field specify a particular action when addressing registers.

Attempting to read from a nonreadable register, or writing to a nonwritable register causes Unpredictable results.

The opcode_1, opcode_2, and CRm fields Should Be Zero, except when the values specified are used to select the desired operations, in all instructions that access CP15. Using other values results in Unpredictable behavior.

2.3.2 Register 0, ID Code Register

This is a read-only register that returns a 32-bit device ID code. The ID Code Register is accessed by reading CP15 register 0 with the opcode_2 field set to any value other than 1 or 2. For example:

```
MRC p15, 0, Rd, c0, c0, {0,3-7}; returns ID register
```

Table 2-3 shows the contents of the ID code.

Table 2-3 Register 0, ID code

Register bits	Function	Value
[31:24]	Implementor	0x41
[23:20]	Variant (reserved)	0x0
[19:16]	ARM architecture 5TE	0x5
[15:4]	Primary part number	0x946
[3: 0]	Revision (major product revision)	0x1

2.3.3 Register 0, Cache Type Register

This is a read-only register that contains information aboutthe size and architecture of the instruction cache and data cache, enabling operating systems to establish how to perform operations such as cache cleaning and lockdown. Future ARM cached processors will contain this register, enabling RTOS vendors to produce future-proof versions of their operating systems.

The cache type register is accessed by reading CP15 register 0 with the opcode_2 field set to 1. For example:

```
MCR p15,0,Rd,c0,c0,1; returns cache details
```

Table 2-4 shows the format of the register.

Table 2-4 Cache Type Register format

Register bits	Function	Value
[31:29]	Reserved	b000
[28:25]	Cache type	b0111
[24]	Harvard/unified	b1 (defines Harvard cache)
[23:22]	Reserved	b00
[21:18]	Data cache size	Implementation-specific
[17:15]	Data cache associativity	Implementation-specific
[14]	Data cache absent	Implementation-specific
[13:12]	Data cache words per line	b10 (defines 8 words per line)
[11:10]	Reserved	b00
[9:6]	Instruction cache size	Implementation-specific
[5:3]	Instruction cache associativity	Implementation-specific
[2]	Instruction cache absent	Implementation-specific
[1:0]	Instruction cache words per line	b10 (defines 8 words per line)

Bits [28:25] indicate which major cache class the implementation falls into. b0111 means that the cache provides:

- cache-clean-step operation
- cache-flush-step operation
- lock-down facilities.

Bits [21:18] give the data cache size. Bits [9:6] give the instruction cache size. Table 2-5 lists the meaning of values used for cache size encoding.

Table 2-5 Cache size encoding

Bits [21:18] and bits[9:6]	Cache size
b0000	0KB
b0011	4KB
b0100	8KB
b0101	16KB
b0110	32KB
b0111	64KB
b1000	128KB
b1001	256KB
b1010	512KB
b1011	1MB

Bits [17:15] give the data cache associativity. Bits [5:3] give the instruction cache associativity. Table 2-6 lists the meaning of values used for cache associativity encoding.

Table 2-6 Cache associativity encoding

Bits [17:15] and bits [5:3]	Associativity
b000	Direct mapped
b010	4

The cache associativity fields in the cache type register are implementation-specific (implementor-defined). Therefore, if the implementation has an instruction or data cache, the associativity for that cache is set to b010 to indicate a four-way set associative cache. If either cache is not included in a specific implementation, then the associativity field for that cache is set to b000 to indicate that the cache is absent.

Bit 14 gives the data cache base size and bit 2 gives the instruction cache base size.

The base size bits are implementation-specific. If the implementation has an instruction or data cache, the base size bit for that cache is set to 0, indicating that the cache type parameters are valid. If either cache is not included for a specific implementation, the relevant base size is set to 1, indicating that the cache is absent.

The cache base size and cache size fields are generated within the cache blocks to avoid having to resynthesize the design for different cache sizes.

2.3.4 Register 0, Tightly-coupled Memory Size Register

This is a read-only register that returns the size of the Instruction and Data *Tightly-coupled Memory* (TCM) integrated with the ARM946E-S processor. The register contents reflect the state of input signals **PhyITCMSize[3:0]** and **PhyDTCMSize[3:0]**.

The tightly-coupled memory size register is accessed by reading CP15 register 0 with the `opcode_2` field set to 2. For example:

```
MRC p15, 0, Rd, c0, c0, 2; returns tightly-coupled memory size register
```

The register contains information about the size of the Instruction TCM and Data TCM. Table 2-7 shows the format of the register.

Table 2-7 Tightly-coupled Memory Size Register

Register bits	Meaning	Value
[31:22]	Reserved	b0000000000
[21:18]	Data TCM size	Implementation-specific
[17:15]	Reserved	b000
[14]	Data TCM absent	Implementation-specific
[13:10]	Reserved	b0000
[9:6]	Instruction TCM size	Implementation-specific
[5:3]	Reserved	b000
[2]	Instruction TCM absent	Implementation-specific
[1:0]	Reserved	b00

The memory size parameters are implementation-specific. The values used are generated within the memory blocks. This enables the memory size to be changed without having to re-synthesize the full design. Bits [21:18] define the Data TCM size. Bits [9:6] define the Instruction TCM size. Table 2-8 shows the memory size field definitions for Instruction and Data TCM sizes.

Table 2-8 Memory size field

Bits [21:18] and bits [9:6]	TCM size
b0000	0KB
b0011	4KB
b0100	8KB
b0101	16KB
b0110	32KB
b0111	64KB
b1000	128KB
b1001	256KB
b1010	512KB
b1011	1MB

If the TCM is absent, the relevant TCM absent bit (bit 14 or bit 2) in the tightly-coupled memory size register should be one. If TCM is present in the design, the relevant TCM absent bit should be zero.

2.3.5 Register 1, Control Register

This register contains the control bits of the ARM946E-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
```

Table 2-9 shows the functions controlled by register 1.

Table 2-9 Register 1, Control Register

Register bits	Function
[31:20]	Reserved (SBZ)
[19]	Instruction TCM load mode
[18]	Instruction TCM enable
[17]	Data TCM load mode
[16]	Data TCM enable
[15]	Disable loading TBIT
[14]	Round-robin replacement
[13]	Alternate vector select
[12]	Instruction cache enable
[11:8]	Reserved (SBZ)
[7]	Big-endian
[6:3]	Reserved (SBO)
[2]	Data cache enable
[1]	Reserved (SBZ)
[0]	Protection unit enable

The bits in the Control Register are described in this section.

Bit 19, Instruction TCM load mode

This bit controls the operation of the Instruction TCM load mode.

You can use the Instruction TCM load mode for initializing the Instruction TCM. The Instruction TCM load mode enables you to load data into the ARM946E-S processor from either data cache or main memory, and then write to the same address but within the Instruction TCM. This enables you to copy boot code from memory located at address `0x0` into the Instruction TCM which, when enabled, also exists at address `0x0`. The operation of the load mode is described in *Initializing the Instruction TCM* on page 5-3.

At reset this bit is cleared.

Bit 18, Instruction TCM enable

This bit controls operation of the Instruction TCM. When the Instruction TCM is enabled, all instruction and data accesses to the Instruction TCM address range access the Instruction TCM.

At reset this bit takes the value of the input pin **INITRAM**.

Bit 17, Data TCM load mode

This bit controls the operation of the Data TCM load mode.You can use the Data TCM load mode for initializing the Data TCM. The Data TCM load mode enables you to load data into ARM registers from either data cache or main memory, and then write to the same address but within the Data TCM. The operation of the load mode is described in *Initializing the Data TCM* on page 5-5.

At reset this bit is cleared.

Bit 16, Data TCM enable

This bit controls operation of the Data TCM. When the Data TCM is enabled, it takes precedence over the data cache and AHB for data accesses.

At reset this bit is cleared.

Bit 15, Disable loading TBIT

This bit controls the behavior of load PC instructions. When clear the ARMv5TE-specific behavior is enabled, and bit 0 of the loaded data is used to control the entry into Thumb state when the PC (r15) is the destination register. When set, this ARMv5TE behavior is disabled.

At reset this bit is cleared.

Bit 14, Round-robin replacement

This bit controls the cache replacement algorithm.

When set, round-robin replacement is used. When clear, a pseudo-random replacement algorithm is used.

At reset this bit is cleared.

Bit 13, Alternate vectors select

This bit controls the base address used for the exception vectors.

When clear, the base address for the exception vectors is `0x00000000`. When set, the base address is `0xFFFF0000`.

Note

This bit is initialized either set or clear during system reset, depending on the value of the input pin, **VINITHI**. This enables you to define the exception vector location during reset to suit the boot mechanism of the application. You can then reprogram this bit as required following system reset.

Bit 12, Instruction cache enable

Caution

You must not enable the instruction cache if your implementation is configured with zero size cache. Enabling the instruction cache when no cache is present can lead to Unpredictable behavior.

Controls the behavior of the instruction cache. To use the instruction cache, both the protection unit enable bit (bit 0) and the instruction cache enable bit must be set. This can be done with a single write to register 1.

At reset this bit is cleared.

Bit 7, Endian configuration

Selects the endian configuration of the ARM946E-S processor. When this bit is set, big-endian configuration is selected. When clear, little-endian configuration is selected.

At reset this bit is cleared.

Bit 2, Data cache enable

Caution

You must not enable the data cache if your implementation is configured with zero size cache. Enabling the data cache when no cache is present can lead to Unpredictable behavior.

This bit controls the behavior of the data cache.

To use the data cache, both the protection unit enable bit (bit 0) and the data cache enable bit must be set. This can be done with a single write to register 1.

At reset this bit is cleared.

Bit 0, Protection unit enable

This bit controls the operation of the ARM946E-S protection unit.

At reset this bit is cleared. This disables the protection unit, and as a result disables the instruction and data caches and the write buffer.

At least one protection region (see *Register 6, Protection Region Base and Size Registers* on page 2-19 and Chapter 4 *Protection Unit*) must be programmed before the protection unit is enabled.

2.3.6 Register 2, Cache Configuration Registers

These registers contain the cachable attributes for the eight areas of memory. Individual control is provided for the instruction and data caches. If the `opcode_2` field is 0, then the data cache bits are programmed. If the `opcode_2` field is 1, then the instruction cache bits are programmed. To read and write these registers:

```
MRC p15, 0, Rd, c2, c0, 0; read data cachable bits
MRC p15, 0, Rd, c2, c0, 1; read instruction cachable bits
MCR p15, 0, Rd, c2, c0, 0; write data cachable bits
MCR p15, 0, Rd, c2, c0, 1; write instruction cachable bits
```

The format for the cachable bits in data and instruction areas is the same, and is shown in Table 2-10.

Table 2-10 Programming instruction and data cachable bits

Register bits	Function
[7]	Cachable bit (C_7) for area 7
[6]	Cachable bit (C_6) for area 6
[5]	Cachable bit (C_5) for area 5
[4]	Cachable bit (C_4) for area 4
[3]	Cachable bit (C_3) for area 3

Table 2-10 Programming instruction and data cachable bits (continued)

Register bits	Function
[2]	Cachable bit (C_2) for area 2
[1]	Cachable bit (C_1) for area 1
[0]	Cachable bit (C_0) for area 0

2.3.7 Register 3, Write Buffer Control Register

This register contains the write buffer control (bufferable) attribute for the eight areas of memory.

Note

This register only applies to data accesses.

To read and write the write buffer control register:

```
MCR p15, 0, Rd, c3, c0, 0; write data bufferable bits
MRC p15, 0, Rd, c3, c0, 0; read data bufferable bits
```

Table 2-11 shows the format for the bufferable bits in the data areas.

Table 2-11 Programming data bufferable bits

Register bits	Function
[7]	Bufferable bit (B_7) for data area 7
[6]	Bufferable bit (B_6) for data area 6
[5]	Bufferable bit (B_5) for data area 5
[4]	Bufferable bit (B_4) for data area 4
[3]	Bufferable bit (B_3) for data area 3
[2]	Bufferable bit (B_2) for data area 2
[1]	Bufferable bit (B_1) for data area 1
[0]	Bufferable bit (B_0) for data area 0

2.3.8 Register 5, Access Permission Registers

There are four access permission registers. These contain the access permission bits for the instruction and data protection regions. The opcode_2 field of the MCR/MRC instruction determines whether the standard or extended registers are accessed, and if the instruction or data access permissions are accessed. To read and write the extended registers:

```
MRC p15, 0, Rd, c5, c0, 2; read data access permission bits
MRC p15, 0, Rd, c5, c0, 3; read instruction access permission bits
MCR p15, 0, Rd, c5, c0, 2; write data access permission bits
MCR p15, 0, Rd, c5, c0, 3; write instruction access permission bits
```

The format for the access permission bits in instruction and data areas is the same, and is shown in Table 2-12.

Table 2-12 Programming instruction and data access permission bits (extended)

Register bits	Function
[31:28]	Ap7[3:0] bits for area 7
[27:24]	Ap6[3:0] bits for area 6
[23:20]	Ap5[3:0] bits for area 5
[19:16]	Ap4[3:0] bits for area 4
[15:12]	Ap3[3:0] bits for area 3
[11:8]	Ap2[3:0] bits for area 2
[7:4]	Ap1[3:0] bits for area 1
[3:0]	Ap0[3:0] bits for area 0

The values of the IApn[3:0] and DApn[3:0] bits define the access permission for each area of memory, n. Table 2-13 shows the encoding.

Table 2-13 Access permission encoding (extended)

I/DApn[3:0]	Access permission	
	Privileged	**User**
b0000	No access	No access
b0001	Read/write access	No access
b0010	Read/write access	Read-only
b0011	Read/write access	Read/write access
b0100	UNP	UNP
b0101	Read-only	No access
b0110	Read-only	Read-only
b0111	UNP	UNP
b1xxx	UNP	UNP

The following instructions are supported for backwards compatibility with existing ARM processors with memory protection, and access the standard registers:

```
MRC p15, 0, Rd, c5, c0, 0; read data access permission bits
MRC p15, 0, Rd, c5, c0, 1; read instruction access permission bits
MCR p15, 0, Rd, c5, c0, 0; write data access permission bits
MCR p15, 0, Rd, c5, c0, 1; write instruction access permission bits
```

Table 2-14 shows the data format for these registers.

Table 2-14 Instruction and data access permission bits (standard)

Register bits	Function
[15:14]	Ap7[1:0] bits for area 7
[13:12]	Ap6[1:0] bits for area 6
[11:10]	Ap5[1:0] bits for area 5
[9:8]	Ap4[1:0] bits for area 4
[7:6]	Ap3[1:0] bits for area 3

Table 2-14 Instruction and data access permission bits (standard) (continued)

Register bits	Function
[5:4]	Ap2[1:0] bits for area 2
[3:2]	Ap1[1:0] bits for area 1
[1:0]	Ap0[1:0] bits for area 0

The values of the IApn[1:0] and DApn[1:0] bits define the access permission for each area of memory, n. Table 2-15 shows the encoding.

Table 2-15 Access permission encoding (standard)

I/DApn[1:0]	Access permission	
	Privileged	User
b00	No access	No access
b01	Read/write access	No access
b10	Read/write access	Read-only
b11	Read/write access	Read/write access

Note

On reset, the values of IApn and DApn bits are undefined. However, because on reset the protection unit is disabled, this is as though all areas are set to privileged mode read/write access and User read/write access. Therefore, you must program the access permission registers before you enable the protection unit.

If the access permissions are initially programmed using the extended access permissions (see Table 2-13 on page 2-18), and then reprogrammed using the standard access permissions (see Table 2-15 on page 2-19), the access permissions applied are as if Apn[3:2] are programmed to b00 in Table 2-13 on page 2-18.

2.3.9 Register 6, Protection Region Base and Size Registers

These registers define the protection region base address and size. You can define eight programmable regions using these registers. The values are ignored when the protection unit is disabled, and on reset only the region enable bit for each region is reset to 0. All other bits are undefined. You must program at least one memory region before you enable the protection unit.

The instructions used to access the eight Protection Region Base and Size Registers are listed in Table 2-16.

Table 2-16 Accessing Protection Region Base and Size Registers

ARM instruction	Protection Region Base and Size Register
`MCR/MRC p15, 0, Rd, c6, c7, 0`	Memory region 7
`MCR/MRC p15, 0, Rd, c6, c6, 0`	Memory region 6
`MCR/MRC p15, 0, Rd, c6, c5, 0`	Memory region 5
`MCR/MRC p15, 0, Rd, c6, c4, 0`	Memory region 4
`MCR/MRC p15, 0, Rd, c6, c3, 0`	Memory region 3
`MCR/MRC p15, 0, Rd, c6, c2, 0`	Memory region 2
`MCR/MRC p15, 0, Rd, c6, c1, 0`	Memory region 1
`MCR/MRC p15, 0, Rd, c6, c0, 0`	Memory region 0

Each Protection Region Base and Size Register has the format shown in Table 2-17.

Table 2-17 Protection Region Base and Size Register format

Register bits	Function
[31:12]	Region base address
[5:1]	Region size
[0]	1 = Region enable 0 = Region disable Reset to 0

You must align the region base to a region size boundary, where the region size is defined in its respective protection region register. The behavior is Unpredictable if this is not done.

Region sizes are encoded as shown in Table 2-18.

Table 2-18 Region size encoding

Bit encoding	Region size
b00000 to b01010	Reserved (UNP)
b01011	4KB
b01100	8KB
b01101	16KB
b01110	32KB
b01111	64KB
b10000	128KB
b10001	256KB
b10010	512KB
b10011	1MB
b10100	2MB
b10101	4MB
b10110	8MB
b10111	16MB
b11000	32MB
b11001	64MB
b11010	128MB
b11011	256MB
b11100	512MB
b11101	1GB
b11110	2GB
b11111	4GB

Example base setting

An 8KB size region aligned to an 8KB boundary at `0x00002000` (covering the address range `0x00002000-0x00003FFF`) is programmed as `0x00002019`.

The following instruction is supported for backward compatibility with other ARM processors using a memory protection unit:

```
MRC p15, 0, Rd, c6, CRm, 1; returns protection region register
```

This instruction enables the protection region registers to be read.

You must not write to the protection region base and size registers with `opcode_2` set to 1 because the behavior is Unpredictable.

2.3.10 Register 7, Cache Operations Register

You can use a write to this register to perform the following operations:

- flush instruction cache and data cache
- prefetch an instruction cache line
- wait for interrupt
- drain the write buffer
- clean and flush the data cache.

The ARM946E-S processor uses a subset of the ARM architecture v4 functions (defined in the *ARM Architecture Reference Manual*). Table 2-19 shows the available operations.

Table 2-19 Cache operations

ARM instruction	Function	Value
`MCR p15, 0, Rd, c7, c5, 0`	Flush instruction cache	SBZ[a]
`MCR p15, 0, Rd, c7, c5, 1`	Flush instruction cache single entry	Address
`MCR p15, 0, Rd, c7, c13, 1`	Prefetch instruction cache line	Address
`MCR p15, 0, Rd, c7, c6, 0`	Flush data cache	SBZ[a]
`MCR p15, 0, Rd, c7, c6, 1`	Flush data cache single entry	Address
`MCR p15, 0, Rd, c7, c10, 1`	Clean data cache entry	Address

Table 2-19 Cache operations (continued)

ARM instruction	Function	Value
`MCR p15, 0, Rd, c7, c14, 1`	Clean and flush data cache entry	Address
`MCR p15, 0, Rd, c7, c10, 2`	Clean data cache entry	Index and segment
`MCR p15, 0, Rd, c7, c14, 2`	Clean and flush data cache entry	Index and segment

a. The value transferred in Rd Should Be Zero.

Figure 2-2 shows the data format for index and segment operations.

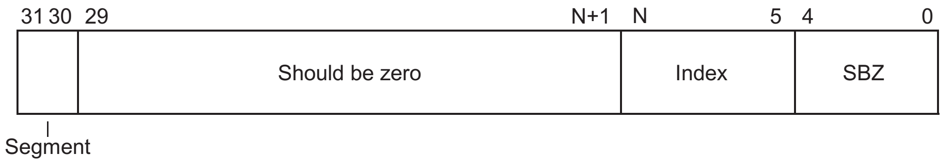


Figure 2-2 Register 7, Index and segment format

The size of the index (N:5) varies depending on the implemented cache size. Table 2-20 shows how the index size changes for the cache sizes supported by the ARM946E-S processor.

Table 2-20 Index fields for supported cache sizes

Cache size	Index
4KB	[9:5]
8KB	[10:5]
16KB	[11:5]
32KB	[12:5]
64KB	[13:5]
128KB	[14:5]
256KB	[15:5]
512KB	[16:5]
1MB	[17:5]

Figure 2-3 shows the data format for the instruction cache prefetch operation.

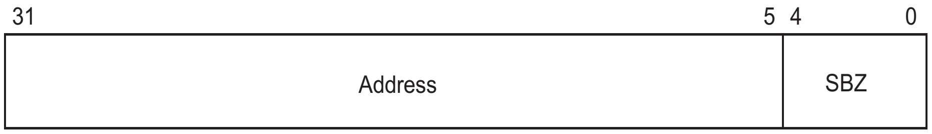


Figure 2-3 Instruction cache address format

Cache clean and flush operations

Cache clean and flush operations can occur during instruction and data linefetches. In such circumstances the linefetch completes before the clean or flush operation is executed.

Drain write buffer

This operation stalls instruction execution until the write buffer is emptied. This 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 is 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
```

This stalls the processor core until any outstanding accesses in the write buffer are completed, that is, until all data is written to external memory.

Wait for interrupt

This operation enables the ARM946E-S processor to enter a low-power standby mode. When you invoke this operation, the processor core is halted and the cache and TCMs are placed in a low-power state 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
```

This is the preferred encoding for new software. For compatibility with existing software, the ARM946E-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 this instruction is executed until either **nFIQ**, **nIRQ**, or **EDBGRQ** are asserted. Also, if the debugger sets the debug request bit in the EmbeddedICE-RT logic control register then this causes the *wait for interrupt* condition to terminate.

In the case of **nFIQ** and **nIRQ**, the processor operation continues regardless of whether the interrupts are enabled or disabled (that is, independent of the I and F bits in the processor CPSR). **DBGEN** must be set (debug enabled) if this operation of **EDBGRQ** or of the debug request bit is required.

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.

The write buffer continues to drain until empty while the wait for interrupt operation is executing.

2.3.11 Register 9, Cache Lockdown Registers

These registers enable you to lock down regions of the cache. To read and write these registers:

```
MCR p15, 0, Rd, c9, c0, 0; write data lockdown control
MRC p15, 0, Rd, c9, c0, 0; read data lockdown control
MCR p15, 0, Rd, c9, c0, 1; write instruction lockdown control
MRC p15, 0, Rd, c9, c0, 1; read instruction lockdown control
```

Table 2-21 shows the format of the register, `Rd`, transferred during this operation.

Table 2-21 Lockdown Register format

Register bits	Function
[31]	Load bit, DL/IL
[30:2]	UNP/SBZ
[1:0]	Cache segment, Dindex, Iindex

Lockdown is described in *Cache lockdown* on page 3-12.

2.3.12 Register 9, Tightly-coupled Memory Region Registers

These registers enable you to modify the visible size of the TCMs.

You can either increase or decrease the size of the TCMs from the physical sizes described in Register 0 (see *Register 0, Tightly-coupled Memory Size Register* on page 2-10). Increasing the visible size of the TCMs above the physical size enables aliasing within the TCM space. This feature is useful for debugging multitasking systems.

There is a memory region register for each of the TCMs:

```
MRC p15, 0, Rd, c9, c1, 0; read data tightly-coupled memory
MCR p15, 0, Rd, c9, c1, 0; write data tightly-coupled memory
MRC p15, 0, Rd, c9, c1, 1; read instruction tightly-coupled memory
MCR p15, 0, Rd, c9, c1, 1; write instruction tightly-coupled memory
```

Table 2-22 shows the format of each TCM region register.

Table 2-22 TCM Region Register format

Register bits	Function
[31:12]	Region base
[5:1]	Area size Minimum size = 4KB Maximum size = 4GB (See Table 2-23).
[0]	SBZ

For a given number of aliases for the physical memory size (set in register 0), the area size is calculated in the following way:

```
Number of required aliases = x (where x is a power of 2)
N = log₂x (or 2ᴺ = x)
Area size = Physical size + N
```

Table 2-23 shows the encodings for the supported TCM area sizes.

Table 2-23 TCM area size encoding

Bit encoding	TCM area size
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
b10010	128MB
b10011	256MB
b10100	512MB
b10101	1GB
b10110	2GB
b10111	4GB

You must align the region base to an area size boundary. The behavior is Unpredictable if this is not done.

The Instruction TCM base address is fixed at `0x00000`. For the Instruction TCM, the region base returns the value `0x00000` when read.

When writing to the Instruction TCM, you must set the region base to `0x00000`. Writes with the region base set to any other value are Unpredictable.

At reset, the region base for both the Instruction and Data TCM region registers are cleared to `0x00000`.

At reset, the area size for the Instruction and Data TCM region registers takes the value defined in the TCM size register (see *Register 0, Tightly-coupled Memory Size Register* on page 2-10).

You must program the Data TCM region registers before you set the Data TCM enable bit (bit 16) in register 1 (see *Register 1, Control Register* on page 2-11). If this is not done, the Data TCM resides at the same location resulting in Unpredictable behavior.

Note

If the Data TCM is located at the same address as the Instruction TCM, then the instruction memory takes precedence for data accesses. If the Data TCM is located at the same address as the Instruction TCM, and the Instruction TCM is in load mode, data accesses read from the Data TCM and write to the Instruction TCM.

2.3.13 Register 13, Trace Process Identifier Register

This register enables you to identify the currently executing process in multi-tasking environments using the real-time trace tools.

The contents of this register are replicated on the **ETMPROCID** pins of the ARM946E-S processor.

The following ARM instructions are used for accessing the Process ID Register:

```
MRC p15, 0, Rd, c13, c0, 1; read process ID register
MCR p15, 0, Rd, c13, c0, 1; write process ID register
```

To support software written for other ARM processors, the following instructions are also supported:

```
MRC p15, 0, Rd, c13, c1, 1; read process ID register
MCR p15, 0, Rd, c13, c1, 1; write process ID register
```

The format of the register, `Rd`, transferred during these operations is shown in Figure 2-4.

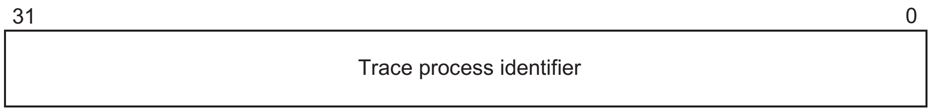


Figure 2-4 Process ID format

2.3.14 Register 15, BIST Control Registers

Register 15 gives you access to the test features included within the ARM946E-S processor. Memory BIST operations are initiated by writes to this register. BIST results and status are evaluated by reading this register. The formats of the TAG BIST Control Register, TCM BIST Control Register, and the Cache RAM BIST Control Register are the same.

Table 2-24 shows the register map for CP15 register 15 BIST-related instructions.

Table 2-24 Register 15, BIST instructions

Register	Read	Write
TAG BIST Control Register	`MRC p15, 0, Rd, c15, c0, 1`	`MCR p15, 0, Rd, c15, c0, 1`
TCM BIST Control Register	`MRC p15, 1, Rd, c15, c0, 1`	`MCR p15, 1, Rd, c15, c0, 1`
Cache RAM BIST Control Register	`MRC p15, 2, Rd, c15, c0, 1`	`MCR p15, 2, Rd, c15, c0, 1`

Table 2-25 shows CP15 register 15 implementation-specific BIST instructions.

Table 2-25 Register 15, implementation-specific BIST instructions

Register	Read	Write
Instruction TAG BIST Address Register	`MRC p15, 0, Rd, c15, c0, 2`	`MCR p15, 0, Rd, c15, c0, 2`
Instruction TAG BIST General Register	`MRC p15, 0, Rd, c15, c0, 3`	`MCR p15, 0, Rd, c15, c0, 3`
Data TAG BIST Address Register	`MRC p15, 0, Rd, c15, c0, 6`	`MCR p15, 0, Rd, c15, c0, 6`
Data TAG BIST General Register	`MRC p15, 0, Rd, c15, c0, 7`	`MCR p15, 0, Rd, c15, c0, 7`
Instruction TCM BIST Address Register	`MRC p15, 1, Rd, c15, c0, 2`	`MCR p15, 1, Rd, c15, c0, 2`
Instruction TCM BIST General Register	`MRC p15, 1, Rd, c15, c0, 3`	`MCR p15, 1, Rd, c15, c0, 3`
Data TCM BIST Address Register	`MRC p15, 1, Rd, c15, c0, 6`	`MCR p15, 1, Rd, c15, c0, 6`
Data TCM BIST General Register	`MRC p15, 1, Rd, c15, c0, 7`	`MCR p15, 1, Rd, c15, c0, 7`

Table 2-25 Register 15, implementation-specific BIST instructions (continued)

Register	Read	Write
Instruction Cache RAM BIST Address Register	`MRC p15, 2, Rd, c15, c0, 2`	`MCR p15, 2, Rd, c15, c0, 2`
Instruction Cache RAM BIST General Register	`MRC p15, 2, Rd, c15, c0, 3`	`MCR p15, 2, Rd, c15, c0, 3`
Data Cache RAM BIST Address Register	`MRC p15, 2, Rd, c15, c0, 6`	`MCR p15, 2, Rd, c15, c0, 6`
Data Cache RAM BIST General Register	`MRC p15, 2, Rd, c15, c0, 7`	`MCR p15, 2, Rd, c15, c0, 7`

Note

It is recommended that you do not write application code that relies on the presence of the BIST Address and General Registers. Support for these registers in future versions of the ARM946E-S processor is not guaranteed.

Table 2-26 shows the format of the BIST Control Register.

Table 2-26 BIST Control Register bit definitions

Register bit	Meaning when written	Meaning when read
[31: 21]	Instruction BIST size	Instruction BIST size
[20]	Reserved (SBZ)	Instruction BIST complete flag
[19]	Reserved (SBZ)	Instruction BIST fail flag
[18]	Instruction BIST enable	Instruction BIST enable
[17]	Instruction BIST pause	Instruction BIST pause
[16]	Instruction BIST run strobe	Instruction BIST running flag
[15: 5]	Data BIST size	Data BIST size
[4]	Reserved (SBZ)	Data BIST complete flag
[3]	Reserved (SBZ)	Data BIST fail flag
[2]	Data BIST enable	Data BIST enable
[1]	Data BIST pause	Data BIST pause
[0]	Data BIST run strobe	Data BIST running flag

Note

The pause and size bits of this register are not supported in all implementations.

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 2-27 shows some examples.

Table 2-27 BIST size encodings 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	1 KB
b000000 01010	10	4 KB
b000000 01111	15	128 KB
b000000 11000 (maximum)	24	64 MB

2.3.15 Register 15, Test State Register

The register is accessed by:

```
MCR p15, 0, Rd, c15, c0, 0; write test state register
MRC p15, 0, Rd, c15, c0, 0; read test state register
```

Table 2-28 shows the bit assignments of the test state access register.

Table 2-28 Test State Register bit assignments

Register bits	Function
[31:13]	Unpredictable (SBZ)
[12]	Disable data cache streaming
[11]	Disable instruction cache streaming

Table 2-28 Test State Register bit assignments (continued)

Register bits	Function
[10]	Disable data cache linefill
[9]	Disable instruction cache linefill
[8:0]	Reserved (SBZ)

Reading the Test State Register returns bits [12:0] in the least significant bits. The value returned in bits [31:13] is Unpredictable. Writing the test state register updates only bits [12:9].

In debug you must be able to execute code without causing linefills to update the caches, primarily to load new code into memory. This means that `STR` instructions, if they hit the cache, must update the memory and the cache, and that for `LDR` instructions or instruction prefetches that miss, a linefill is not performed.

Bits [10:9] when set prevent the respective cache from performing a linefill on a cache miss. The memory mapping, as seen by the ARM9E-S or by the programmer, is unchanged. This improves the performance of single-stepping when in debug.

Bits [12:11] when set prevent the respective cache from streaming data to the ARM9E-S while the linefill is performed to the cache. The linefill still occurs, but the prefetched instruction or load data is returned to the core at the end of a linefill.

2.3.16 Register 15, Cache Debug Index Register

Additional instructions and operations are required to support debug operations within the cache. Table 2-29 shows the instructions for the additional operations.

Table 2-29 Additional operations

Function	Data	Instruction
Write CP15 Cache Debug Index Register	Index and segment	`MCR p15, 3, Rd, c15, c0, 0`
Read CP15 Cache Debug Index Register	Index and segment	`MRC p15, 3, Rd, c15, c0, 0`
Instruction TAG write	Data	`MCR p15, 3, Rd, c15, c1, 0`
Instruction TAG read	Data	`MRC p15, 3, Rd, c15, c1, 0`
Data TAG write	Data	`MCR p15, 3, Rd, c15, c2, 0`
Data TAG read	Data	`MRC p15, 3, Rd, c15, c2, 0`

Table 2-29 Additional operations (continued)

Function	Data	Instruction
Instruction cache write	Data	`MCR p15, 3, Rd, c15, c3, 0`
Instruction cache read	Data	`MRC p15, 3, Rd, c15, c3, 0`
Data cache write	Data	`MCR p15, 3, Rd, c15, c4, 0`
Data cache read	Data	`MRC p15, 3, Rd, c15, c4, 0`

With the Cache Debug Index Register, you can access any location within the instruction or data cache. You must program this register before using any of the TAG or cache read/write operations. The Cache Debug Index Register provides an index into the cache memories.

Figure 2-5 on page 2-33 shows the format of the index and segment data.

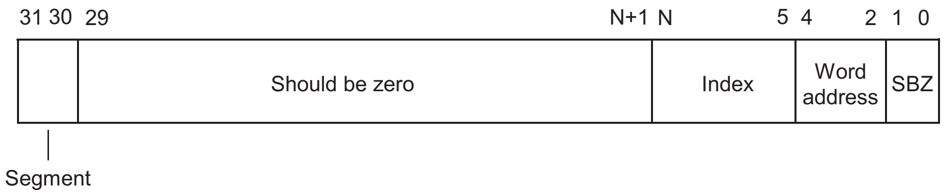


Figure 2-5 Register 15, Index and segment format

The number of bits used in the index field varies depending on the implemented cache size. Table 2-20 on page 2-23 shows how the index address field size changes for the cache sizes supported by the ARM946E-S processor.

——— **Note** ———

For TAG operations, the word address field in the Cache Debug Register is ignored.

———————————

Figure 2-6 shows the data format for the TAG read/write operations.

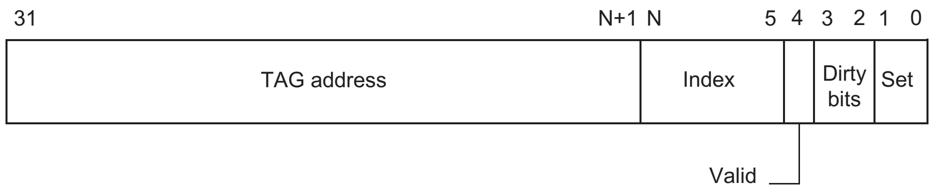


Figure 2-6 Data format TAG read/write operations

The number of bits used in the index and TAG address fields vary depending on the implemented cache size. Table 2-30 shows how the index and TAG address field sizes change for the cache sizes supported by the ARM946E-S processor.

Table 2-30 Index fields for supported cache sizes

Cache size	TAG	Index
4KB	[31:10]	[9:5]
8KB	[31:11]	[10:5]
16KB	[31:12]	[11:5]
32KB	[31:13]	[12:5]
64KB	[31:14]	[13:5]
128KB	[31:15]	[14:5]
256KB	[31:16]	[15:5]
512KB	[31:17]	[16:5]
1MB	[31:18]	[17:5]

2.3.17 Register 15, Trace Control Register

This register enables masking of **ETMFIFOFULL** during interrupts in the ARM946E-S processor. It enables you to determine whether **nIRQ** or **nFIQ** interrupts take priority over **ETMFIFOFULL** to prevent the core being stalled if an interrupt is received when **ETMFIFOFULL** is asserted. Table 2-31 shows the access instructions for register 15.

Table 2-31 Trace Control Register

Register	Read	Write
Trace Control Register	`MRC p15, 1, Rd, c15, c1, 0`	`MCR p15, 1, Rd, c15, c1, 0`

Table 2-32 shows the bit assignments for this register. If bit 1 is 1, **nIRQ** interrupts do not re-enable the ARM946E-S processor if **ETMFIFOFULL** is asserted. If bit 2 is 1, **nFIQ** interrupts do not re-enable the ARM946E-S processor if **ETMFIFOFULL** is asserted. When these bits are set to 0, **ETMFIFOFULL** does not stall the core during interrupts. Bits [2:1] of this register are reset to 0 when **HRESETn** is asserted.

Table 2-32 Trace Control Register bit assignments

Register bits	Content
[31:3]	Reserved (Should Be Zero)
[2]	1 = Mask **nFIQ** interrupts during trace 0 = Do not mask **nFIQ** interrupts during trace
[1]	1 = Mask **nIRQ** interrupts during trace 0 = Do not mask **nIRQ** interrupts during trace
[0]	Reserved (Should Be Zero)

Chapter 3
Caches

To reduce the effective memory access time, the ARM946E-S processor uses a cache controller, an instruction cache, and a data cache. This chapter describes the features and behavior of each of these blocks. It contains the following sections:

3.1 About cache architecture

The ARM946E-S processor incorporates instruction cache and data cache. You can tailor the size of these to suit individual applications. A range of different cache sizes is supported:

- 0KB
- 4KB
- 8KB
- 16KB
- 32KB
- 64KB
- 128KB
- 256KB
- 512KB
- 1MB.

You can select the instruction cache and data cache sizes independently.

The instruction cache and data cache are formed from synchronous SRAM, and have similar architectures. Figure 3-1 on page 3-3 shows an example 8K cache.

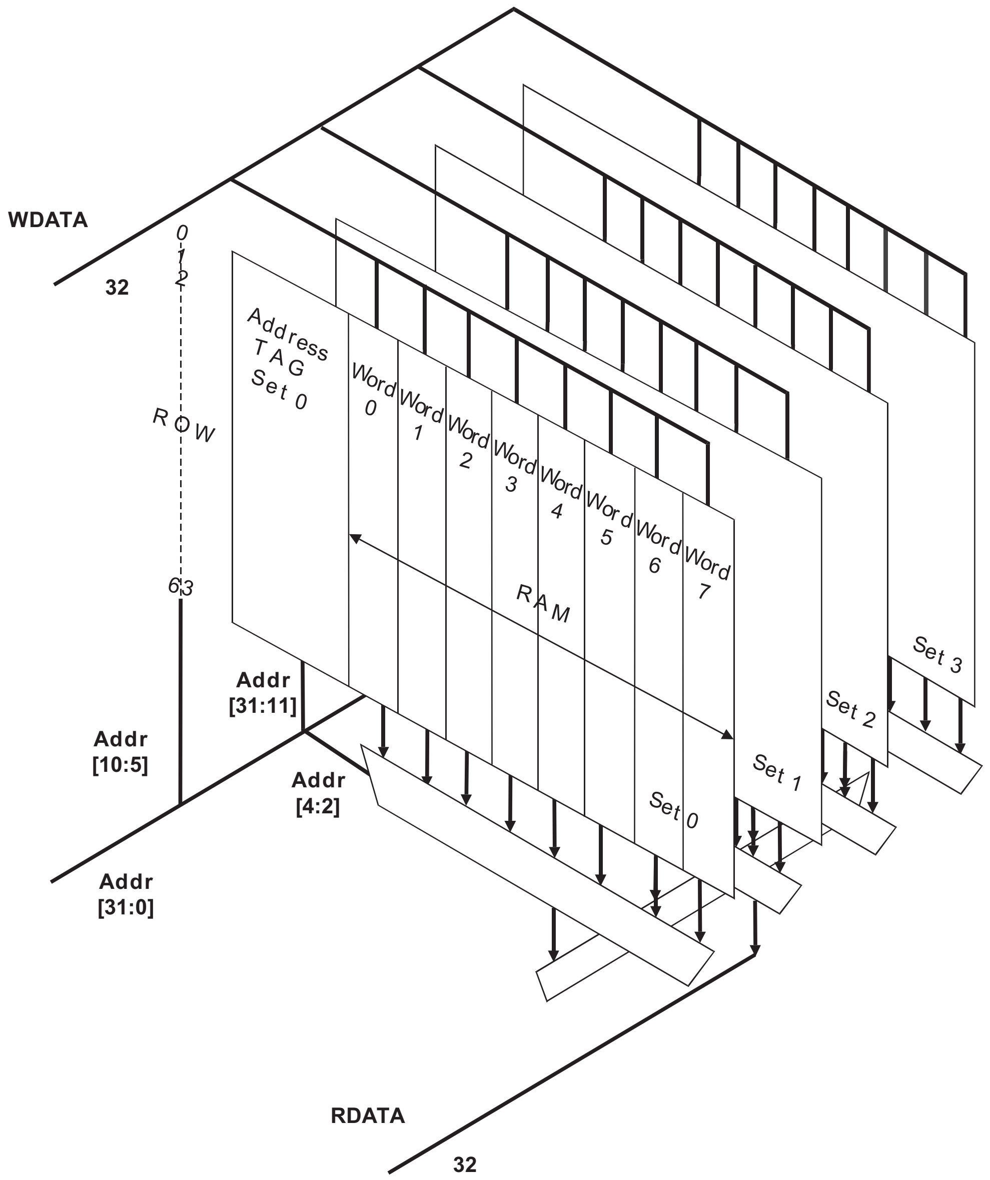


Figure 3-1 Example 8KB cache

The instruction cache and data cache are four-way set associative, with a cache line length of 8 words (32 bytes). Each cache supports single-cycle read access.

Each cache segment consists of a TAG RAM for storing the cache line address and a data RAM for storing the instructions or data.

During a cache access, all TAG RAMs are accessed for the first nonsequential access, and the TAG address is compared with the access address. If a match (or hit) occurs, the data from the segment is selected for return to the ARM9E-S core. If none of the TAGs match (a miss), then external memory must be accessed. If the access is a buffered write then the write buffer is used.

If a read access from a cachable memory region misses, new data is loaded into one of the four segments. This is an *allocate on read-miss* replacement policy. Selection of the segment is performed by a segment counter that can be clocked in a pseudo-random manner, or in a predictable manner based on the replacement algorithm selected.

Critical or frequently accessed instructions or data can be locked into the cache by restricting the range of the replacement counter. You cannot replace locked lines. They remain in the cache until they are unlocked or flushed.

Note

Flushing the entire cache also flushes any locked-down code. If you want to preserve locked-down code, you must flush lines individually, avoiding the locked-down lines.

The access address from the ARM9E-S core can be split into four distinct segments:

- byte address (Addr[1:0])
- word address (Addr[4:2])
- index (cache line)
- address TAG.

Table 3-1 shows how the number of bits in the index and TAG fields change for the cache sizes supported by the ARM946E-S processor.

Table 3-1 TAG and index fields for supported cache sizes

Cache size	Index	TAG
4KB	Addr[9:5]	Addr[31:10]
8KB	Addr[10:5]	Addr[31:11]
16KB	Addr[11:5]	Addr[31:12]
32KB	Addr[12:5]	Addr[31:13]
64KB	Addr[13:5]	Addr[31:14]
128KB	Addr[14:5]	Addr[31:15]

Table 3-1 TAG and index fields for supported cache sizes (continued)

Cache size	Index	TAG
256KB	Addr[15:5]	Addr[31:16]
512KB	Addr[16:5]	Addr[31:17]
1MB	Addr[17:5]	Addr[31:18]

For example, the access address is broken down as shown in Figure 3-2 for a 4KB cache.

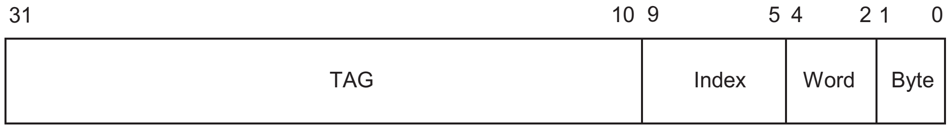


Figure 3-2 Access address for a 4KB cache

Three additional bits are associated with each TAG entry:

Valid bit This is set when the cache line has been written with valid data. Only a valid line can return a hit during a cache lookup. On reset, all the valid bits are cleared.

Dirty bits These are associated with write operations in the data cache and are used to indicate that a cache line contains data that differs from data stored at the address in external memory. One bit is allocated for each half cache line.

Data can only be marked as dirty if it resides in a write-back protection region.

3.2 Instruction cache

The ARM946E-S processor has a four-way set-associative instruction cache. You can choose the size of the instruction cache from any of the supported cache sizes. The instruction cache uses the physical address generated by the processor core. It uses a policy of *allocate on read-miss*, and is always reloaded one cache line (eight words) at a time, through the external interface.

3.2.1 Enabling and disabling the instruction cache

Caution

You must not enable the instruction cache if your implementation is configured with zero size instruction cache. Enabling the instruction cache when no cache is present can lead to Unpredictable behavior.

You can enable the instruction cache by setting bit 12 of the CP15 Control Register. The cache is only enabled if the protection unit is already enabled, or if they are enabled simultaneously. When the instruction cache is enabled, a cachable read-miss places lines in the instruction cache.

You can enable the instruction cache and protection unit simultaneously with a single write to the CP15 Control Register, although you must program at least one protection region before you enable the protection unit. You can lock critical or frequently accessed instructions into the instruction cache.

3.2.2 Instruction cache operation

When enabled, the instruction cache operation is additionally controlled by the *Cachable instruction* (Ci) bit stored in the protection unit. This selectively enables or disables caching for different memory regions. The Ci bit affects instruction cache operation as follows:

Successful cache read

Data is returned to the core only if the Ci bit is 1.

Unsuccessful cache read

If the Ci bit is 1, a linefetch of eight words is performed. The linefetch starts with the requested address aligned to an eight-word boundary (that is, the linefetch starts with word 0). If the Ci bit is 0, a single-word external access is performed to fetch the requested instruction. The cache is not updated.

You can disable the instruction cache by clearing bit 12 of the CP15 Control Register. This prevents all instruction cache look-ups and line fills, and forces all instruction fetches to be performed as single external accesses.

3.2.3 Instruction cache validity

The ARM946E-S processor does not support external memory snooping. Therefore if you write self-modifying code, the instructions in the instruction cache can become incoherent with external memory. Similarly, if you reprogram the protection regions, code might exist in the cache that should be in a noncachable region. In either of these cases you must flush the instruction cache.

You can flush the entire instruction cache by software in one operation, or you can flush individual cache lines by writing to the CP15 Cache Operations Register (register 7). The instruction cache is automatically flushed during reset. The instruction cache never has to be cleaned because its only source of data is from external memory. The ARM9E-S core cannot write to the instruction cache, except using the Cache Debug Index Register.

Flushing the entire cache

As shown in Table 2-19 on page 2-22, you can flush the entire instruction cache using an `MCR` instruction. In this case, the contents of the ARM Register transferred to CP15 should be zero. You can use the following code segment to do this:

```
    MOV    r0, #0                    ; Clear r0
    MCR    p15, 0, r0, c7, c5, 0     ; Flush entire ICache
```

Note

- The use of r0 is arbitrary.
- Flushing the entire cache also flushes any locked-down code. If you want to preserve locked down code, you must flush lines individually, avoiding the locked down lines.

Flushing a single cache line

You can flush single cache lines. To do this, you must specify in `Rd` the address to be flushed from the cache. You can use the following code segment to do this:

```
LDR    r0, =FlushAddress                ; Load r0 with address FlushAddress
MCR    p15, 0, r0, c7, c5, 1            ; Flush single cache line
```

3.3 Data cache

The ARM946E-S processor has a four-way set-associative data cache. You can choose the size of the data cache from any of the supported cache sizes. The data cache uses the physical address generated by the processor core. It uses an *allocate on read-miss* policy, and is always reloaded one cache line (eight words) at a time, through the external interface.

The data cache supports both write-back and write-through modes. For data stores that hit in the data cache, in write-back mode the cache line is updated and the dirty bit associated with the half cache line updated is set. This indicates that the internal version of the data differs from that in external memory. In write-through mode, a store that hits in the data cache causes the cache line to be updated but not marked as dirty, because the data store is also written to the write buffer to keep the external memory consistent. In both write-back and write-through modes, a store that misses in the cache is sent to the write buffer. When a linefetch causes a cache line to be evicted from the data cache, the dirty bit for each half of the victim line is read and, if the half-line contains valid and dirty data, it is written back to the write buffer before the linefill replaces it.

The *Cachable data* (Cd) and *Bufferable data* (Bd) bits control the behavior of the data cache. For this reason the protection unit must be enabled when the data cache is enabled.

3.3.1 Enabling and disabling the data cache

Caution

You must not enable the data cache if your implementation is configured with zero size data cache. Enabling the data cache when no cache is present can lead to Unpredictable behavior.

You can enable the data cache by setting bit 2 of the CP15 Control Register. The cache is only enabled if the protection unit is already enabled, or is enabled simultaneously, although you must program at least one protection region before you enable the protection unit.

You can disable the data cache by clearing bit 2 of the CP15 Control Register.

The data cache is automatically disabled and flushed on reset.

When the data cache is disabled, cache searches are prevented. This marks all data accesses as noncachable, forcing the ARM946E-S processor to perform external accesses. The write buffer control is still decoded from the Bd and Cd bits. The Cd bit is forced to 0 (noncachable).

3.3.2 Operation of the Bd and Cd bits

The Cd bit determines whether data being read must be placed in the data cache and used for subsequent reads. Typically, main memory is marked as cachable to reduce memory access time and therefore increase system performance. It is usual to mark input/output space as noncachable. For example, if a processor is polling a memory-mapped register in input/output space, it is important that the processor is forced to read data direct from the peripheral, and not a copy of initial data held in the data cache.

The Bd and Cd bits affect writes that both hit and miss in the data cache. If the Bd and Cd bits are both 1, the area of memory is marked as write-back, and stores that hit in the data cache only update the cache, not external memory. If the Bd bit is 0 and the Cd bit is 1, the area of memory is marked as write-through, and stores that hit in the data cache update both the cache and external memory.

3.3.3 Data cache operation

When the data cache is enabled, it is searched when the processor performs a load or store. If the cache hits on a load, data is returned from the cache if the Cd bit is 1. If the cache read-misses, the Cd bit is examined. Table 3-2 shows the meaning of the values of the Cd bit.

Table 3-2 Meaning of Cd bit values

Cd bit value	Meaning
1	Cachable data area and protection unit enabled. A linefill of eight words is performed and the data is written into the data cache.
0	A single or multiple external access is performed and the cache is not updated.

Stores that hit in the cache update the cache line if the Cd bit is 1. Stores that miss the cache use the Cd and Bd bits to determine whether the write is buffered. A write miss is not loaded into the cache as a result of that miss.

Load and store multiples are broken up on 4KB boundaries (the minimum protection region size), enabling a protection check to be performed in case the *Load Multiple* (`LDM`) or *Store Multiple* (`STM`) crosses into a region with different protection properties.

3.3.4 Data cache validity

The ARM946E-S processor does not support memory translation so you can always consider the data in the data cache as valid within the context of the ARM946E-S processor. However, if you use external memory translation, and the mappings are changed, the data cache is no longer consistent with external memory, and you must flush it.

The ARM946E-S processor does not support external memory snooping. Any shared data memory space therefore, must not be cachable. Additionally, if you reprogram the data protection regions, data already in the cache might now be in a noncachable region, and you must flush it.

3.3.5 Data cache clean and flush

The data cache has flexible cleaning and flushing utilities that enable the following operations:

- You can invalidate the whole data cache (*flush data cache*) in one operation without writing back dirty data.
- You can invalidate individual lines without writing back any dirty data (*flush data cache single entry*).
- You can perform cleaning on a line-by-line basis. The data is only written back through the write buffer when a dirty line is encountered, and the cleaned line remains in the cache (*clean data cache single entry*). You can clean cache lines using either their index within the data cache, or their address within memory.
- You can clean and flush individual lines in one operation, using either their index within the data cache, or their address within memory.

You perform the cleaning and flushing operations using CP15 register 7, in a similar way to the instruction cache.

The format of `Rd` transferred to CP15 for all register 7 operations is shown in Figure 3-3.

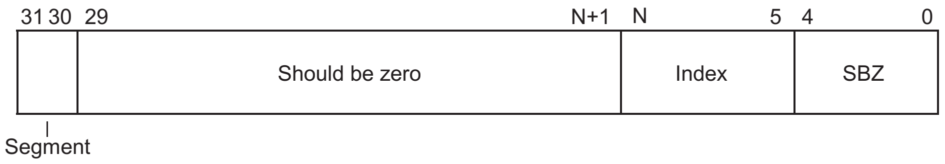


Figure 3-3 Register 7, Rd format

The value of N depends on the cache size, as shown in Table 3-3.

Table 3-3 Calculating index addresses

Cache size	Value of N
4KB	9
8KB	10
16KB	11
32KB	12
64KB	13
128KB	14
256KB	15
512KB	16
1MB	17

The value of N is derived from the following equation:

$$N = \log_2\left(\frac{\text{cache size}}{\text{Number of sets x line length in bytes}}\right) + 4$$

Where the number of sets x the line length in bytes is 128.

It is usual to clean the cache before flushing it, so that external memory is updated with any dirty data. The following code segment shows how you can clean and flush the entire cache (assuming a 4KB data cache):

```
MOV    r1, #0                       ; Initialize segment counter outer_loop
MOV    r0, #0                       ; Initialize line counter inner_loop
ORR    r2, r1, r0                   ; Generate segment and line address
MCR    p15, 0, r2, c7, c14, 2       ; Clean and flush the line
ADD    r0, r0, #0x20                ; Increment to next line
CMP    r0, #0x400                   ; Complete all entries in one segment?
BNE    inner_loop                   ; If not branch back to inner_loop
ADD    r1, r1, #0x40000000          ; Increment segment counter
CMP    r1, #0x0                     ; Complete all segments
BNE    outer_loop                   ; If not branch back to outer_loop
                                    ; End of routine
```

3.4 Cache lockdown

To provide predictable code behavior in embedded systems, a mechanism is provided for locking code into the caches. For example, you can use this feature to hold high-priority interrupt routines where there is a hard real-time constraint, or to hold the coefficients of a DSP filter routine to reduce external bus traffic.

You can lock down a region of the instruction cache or data cache by executing a short software routine, taking note of these requirements:

- the program must be held in a noncachable area of memory
- the cache must be enabled and interrupts must be disabled
- software must ensure that the code or data to be locked down is not already in the cache
- if the caches have been used after the last reset, the software must ensure that the cache in question is cleaned, if appropriate, and then flushed.

You can carry out lockdown in the data cache using CP15 register 9. Instruction cache lockdown uses both CP15 registers 7 and 9.

As described in *About cache architecture* on page 3-2, the ARM946E-S instruction cache and data cache each comprise four segments. You can perform lockdown with a granularity of one segment. Lockdown starts at segment zero, and can continue until three of the four segments are locked.

3.4.1 Locking down the caches

The procedures for locking down a segment in the instruction cache and data cache are slightly different. In both cases you must:

1. Put the cache into lockdown mode by programming register 9.
2. Force a linefill.
3. Lock the corresponding data in the cache.

Data cache lockdown

For the data cache:

1. Write to CP15 register 9, setting DL to 1 (DL is bit 31, the load bit) and Dindex to 0 (Dindex are bits 1:0, the cache segment bits).
2. Initialize the pointer to the first of the words to be locked into the cache.

3. Execute an `LDR` from that location. This forces a linefill from that location and the resulting eight words are captured in the cache.
4. Increment the pointer by 32 (number of bytes in a cache line).
5. Execute an `LDR` from that location. The resulting linefill is captured in the cache.
6. Repeat steps 4 and 5 until all words are loaded in the cache, or one quarter of the cache has been loaded.
7. Write to CP15 register 9, setting DL to 0 and Dindex to 1.

If there is more data to lockdown, at the final step, the DL bit must be left set and the process repeated. The DL bit must only be cleared when all the lockdown data has been loaded. The Dindex bits must be set to the next available segment.

Note

The write to CP15 register 9 must not be executed until the linefill has completed. This is achieved by aligning the `LDR` to the last address of the line.

Instruction cache lockdown

For the instruction cache:

1. Write to CP15 register 9, setting IL to 1 (the load bit) and Iindex to 0 (the cache segment bits).
2. Initialize the pointer to the first of the words to be locked into the cache.
3. Force a linefill from that location by writing to CP15 register 7 (instruction cache preload).
4. Increment the pointer by 32 (number of bytes in a cache line).
5. Force a linefill from that location by writing to CP15 register 7. The resulting linefill is captured in the instruction cache.
6. Repeat steps 4 and 5 until all words are loaded in the cache, or one quarter of the cache has been loaded.
7. Write to CP15 register 9, setting IL to 0 and Iindex to 1.

If there are more instructions to lockdown, at the final step, the IL bit must be left set and the process repeated. The IL bit must only be cleared when all the lockdown instructions have been loaded. The Iindex bits must be set to the next available segment.

The only significant difference between the sequence of operations for the data cache and instruction cache is that an `MCR` instruction must be used to force the linefill in the instruction cache, instead of an `LDR`. The rest of the sequence is the same as for data cache lockdown.

The `MCR` to perform the instruction cache fetch is a CP15 register 7 operation:

```
MCR   p15, 0, Rd, c7, c13, 1
```

Chapter 4
Protection Unit

This chapter describes the ARM946E-S protection unit. It contains the following sections:

4.1 About the protection unit

The protection unit enables you to partition memory and set individual protection attributes for each protection region. You can divide the address space into eight regions of variable size. Figure 4-1 shows a simplified block diagram of the protection unit.

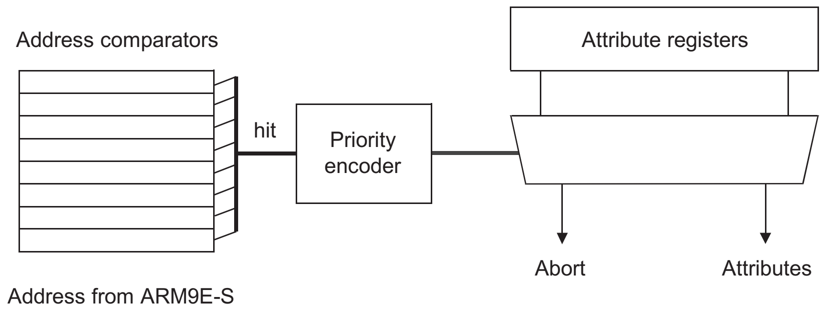


Figure 4-1 Protection unit

The protection unit is programmed using CP15 registers 1, 2, 3, 5, and 6 (see *Accessing CP15 registers* on page 2-6).

4.1.1 Enabling the protection unit

Before the protection unit is enabled, you must program at least one valid protection region. If you do not do this the ARM946E-S processor can enter a state that is recoverable only by reset.

Setting bit 0 of the CP15 register 1, the Control Register, enables the protection unit.

When the protection unit is disabled, all instruction fetches are noncachable and all data accesses are noncachable and nonbufferable.

4.2 Memory regions

You can partition the address space into a maximum of eight regions. Each region is specified by the following:

- region base address
- region size
- cache and write buffer configuration
- read and write access permissions.

The ARM architecture uses constants known as *inline literals* to perform address calculations. These constants are automatically generated by the assembler and compiler and are stored inline with the instruction code. To ensure correct operation, you must define an area of memory, from where code is to be executed, that enables both data and instruction accesses.

The base address and size properties are programmed using CP15 register 6. Table 4-1 shows the format.

Table 4-1 Protection Register format

Register bits	Function
[31:12]	Region base address
[11:6]	Unused
[5:1]	Region size
[0]	Region enable Reset to disable (0).

4.2.1 Region base address

The base address defines the start of the memory region. You must align this to a region-sized boundary. For example, if a region size of 8KB is programmed for a given region, the base address must be a multiple of 8KB.

Note

If the region is not aligned correctly, this results in Unpredictable behavior.

4.2.2 Region size

The region size is specified as a five-bit value, encoding a range of values from 4KB to 4GB. Table 4-2 shows the encoding.

Table 4-2 Region size encoding

Bit encoding	Area size
b00000 to b01010	Reserved
b01011	4KB
b01100	8KB
b01101	16KB
b01110	32KB
b01111	64KB
b10000	128KB
b10001	256KB
b10010	512KB
b10011	1MB
b10100	2MB
b10101	4MB
b10110	8MB
b10111	16MB
b11000	32MB
b11001	64MB
b11010	128MB
b11011	256MB
b11100	512MB
b11101	1GB
b11110	2GB
b11111	4GB

Note

Any value less than b01011 programmed in CP15 register 6 bits [5:1] results in Unpredictable behavior.

4.2.3 Partition attributes

Each region has a number of attributes associated with it. These control how a memory access is performed when the processor core issues an address that falls within a given region. The attributes are:

- cachable
- bufferable (for data regions only)
- read/write permissions.

You specify this information by programming CP15 registers 2, 3, and 5 (see Chapter 2 *Programmer's Model*). If an access fails its protection check (for example, if a User mode application attempts to access a *Privileged mode access only* region), a memory abort occurs. The processor enters the abort exception mode, branching to the Data Abort or Prefetch Abort vector accordingly.

The cachable and bufferable bits in CP15 registers 2 and 3 are used together to select one of four cache and write buffer configurations. These are described in Chapter 6 *Bus Interface Unit and Write Buffer*, and specifically in *The write buffer* on page 6-13.

4.3 Overlapping regions

You can program the protection unit with two or more overlapping regions. When overlapping regions are programmed, a fixed priority scheme is applied to determine the overlapping region attribute that is applied to the memory access (attributes for region 7 take highest priority, those for region 0 take lowest priority). For example:

Region 2	Is programmed to be 4KB in size, starting from address `0x3000` with DApn[3:0] set to b0010. (Privileged mode full access, User mode read only.)
Region 1	Is programmed to be 16KB in size, starting from address `0x0000` with DApn[3:0] set to b0001. (Privileged mode access only.)

When the processor performs a data write to address `0x3010` while in User mode, the address falls into both region 1 and region 2, as shown in Figure 4-2. Because there is a clash, the attributes associated with region 2 are applied. Because you are only enabled to perform reads from this region, a Data Abort occurs.

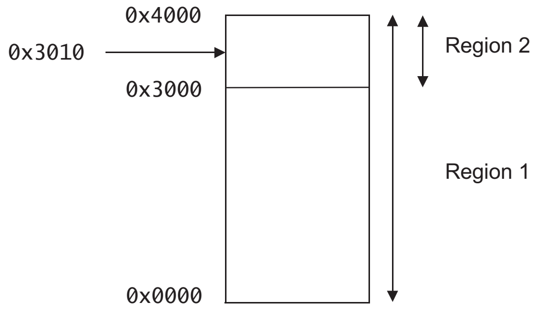


Figure 4-2 Overlapping memory regions

4.3.1 Background regions

Overlapping regions increase the flexibility of how the eight regions can be mapped onto physical memory devices in the system. You can also use the overlapping properties to specify a background region. For example, you might have a number of physical memory areas sparsely distributed across the 4GB address space. If a programming error occurs therefore, it might be possible for the processor to issue an address that does not fall into any defined region.

If the address issued by the processor falls outside any of the defined regions, the ARM946E-S protection unit is hard-wired to abort the access. You can override this behavior by programming region 0 to be a 4GB background region. In this way, if the address does not fall into any of the other seven regions, the access is controlled by the attributes you have specified for region 0.

Chapter 5
Tightly-Coupled Memory Interface

This chapter describes the *Tightly-Coupled Memory* (TCM) interface in the ARM946E-S processor. It contains the following sections:

For details of the ARM9E-S interface signals referenced in this chapter, see the *ARM9E-S Technical Reference Manual*.

5.1 ARM946E-S TCM interface description

The instruction and data *Tightly-Coupled Memories* (TCMs) are placed outside the ARM946E-S processor boundary. This enables greater flexibility in the memory attached to the ARM946E-S processor. The memories used must support single-cycle accesses from the ARM946E-S processor. They must be capable of returning data to the ARM9E-S core in a single cycle. This requirement applies to both the Instruction TCM and Data TCM. They are normally realized using synchronous SRAM.The Instruction TCM and Data TCM can both be of any size from 0 bytes to 1MB, although to ease implementation the size must be an integer power of two. The minimum size for a TCM when present is 4KB. The Instruction TCM and Data TCM can have different sizes.

To enable the Instruction TCM to be initialized, and for access to literal tables during execution, the data interface of the ARM9E-S core must be able to access the Instruction TCM. This means that the ARM946E-S processor must multiplex the instruction and data addresses before entering the Instruction TCM. It also means that the instruction data is routed to both the instruction and data interfaces of the core. See *Instruction TCM accesses* on page 5-9 for details of this data and address multiplexing.

Figure 5-1 shows a typical TCM read cycle. The enable signal, **TCMEn**, is either **ITCMEn** or **DTCMEn**, depending on whether instruction or data memory is being accessed. The TCM interface signals are described in *TCM interface signals* on page B-4.

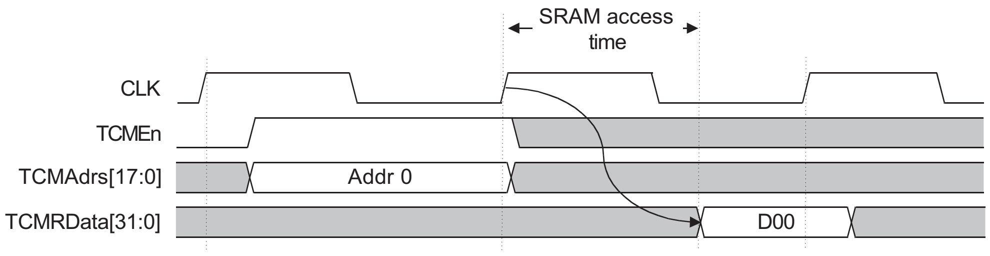


Figure 5-1 TCM read cycle

The Instruction TCM is located at address `0x00000000` in the memory map. This simplifies the implementation of the design by removing the requirement for complex address comparators on both the instruction and data interfaces of the ARM9E-S core to generate the chip select logic for the Instruction TCM.

5.2 Using CP15 Control Register

When out of reset, the behavior of the TCM is controlled by the state of the CP15 Control Register.

5.2.1 Enabling the Instruction TCM

You can enable the Instruction TCM by setting bit 18 of the CP15 Control Register. You must use *read-modify-write* to access this register to preserve the contents of the bits not being modified. See *Register 1, Control Register* on page 2-11 for details of how to read and write the CP15 Control Register. When you have enabled the Instruction TCM, all future ARM9E-S instruction fetches and data accesses to the Instruction TCM address space cause the Instruction TCM to be accessed. Enabling the Instruction TCM greatly increases the performance of the ARM946E-S processor because the majority of accesses to it can be performed with no stall cycles. Accessing the AHB however, can cause several stall cycles for each access.You must take care to ensure that the Instruction TCM is appropriately initialized before it is enabled and used to supply instructions to the ARM9E-S core. If the core tries to execute instructions from uninitialized Instruction TCM, the behavior is Unpredictable.

5.2.2 Disabling the Instruction TCM

You can disable the Instruction TCM by clearing bit 18 of the CP15 Control Register. See *Register 1, Control Register* on page 2-11 for details of how to read and write the CP15 Control Register. When you have disabled the Instruction TCM, all future ARM9E-S instruction fetches access the AHB.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.

5.2.3 Defining the physical and visible size of the Instruction TCM

You can determine the physical size of the Instruction TCM by using CP15 register 0. See *Register 0, Tightly-coupled Memory Size Register* on page 2-10 for more details.

You can determine the visible size of the Instruction TCM by using CP15 register 9. See *Register 9, Tightly-coupled Memory Region Registers* on page 2-26 for more details.

5.2.4 Initializing the Instruction TCM

You must initialize the Instruction TCM with the required code image before execution from the Instruction TCM.

You can initialize the Instruction TCM by writing to the memory from the ARM9E-S core data interface.

The Instruction TCM load mode enables this to be done in an efficient manner. Using the load mode enables you to copy from an address in the data cache or external memory into the same address within the Instruction TCM.

The Instruction TCM load mode bit of CP15 register 1 inhibits reads from the Instruction TCM, forcing reads from addresses that are within the Instruction TCM address range to access either main memory, or the data cache. Writes to addresses that are within the Instruction TCM range are not affected by the Instruction TCM load mode bit. The procedure for initializing the Instruction TCM using the load TCM mode is:

1. Enable the Instruction TCM and Instruction load mode.
2. Load ARM registers from main memory, data cache, or Data TCM.
3. Store ARM registers into Instruction TCM.
4. Increment address pointers and repeat load/store steps until the code image has been copied.

A suggested assembler code sequence for this procedure is:

```
  MOV R0, #0                ; Initialize pointer
  LDR R1, =ImageTop         ; Define end of code image
  MRC p15, 0, R2, c1, c0, 0 ; Read Control Register
  ORR R2, R2, #&C0000
  MCR p15, 0, R2, c1, c0, 0 ; Enable Instruction TCM and Load Mode
CopyLoop
  LDMIA R0, {R2 - R9}       ; Load 8 registers from main memory
  STMIA R0!, {R2 - R9}      ; Store 8 regs into Instruction TCM
  CMP R1, R0                ; Check if limit reached
  BGT CopyLoop              ; Repeat if more to do
```

SWP and SWPB operations to the Instruction TCM when it is in load mode have Unpredictable results. This is because the read accesses external memory or the data cache, and the write updates the Instruction TCM.

SWP and SWPB operations must not be performed to addresses in the Instruction TCM space when it is in load mode.

5.2.5 Enabling the Data TCM

You can enable the Data TCM by setting bit 16 of the CP15 Control Register. See *CP15 register map summary* on page 2-4 for details of how to read and write this register. When you have enabled the Data TCM all future read and write accesses to the Data TCM address space cause the Data TCM to be accessed.

5.2.6 Disabling the Data TCM

You can disable the Data TCM by clearing bit 16 of the CP15 Control Register. When you have disabled the Data TCM all future reads and writes to the Data TCM address space access the AHB. Read and write accesses to Instruction TCM address space either use the Instruction TCM or access the AHB depending on whether Instruction TCM is enabled or not.

5.2.7 Defining the physical and visible size of the Data TCM

You can determine the physical size of the Data TCM by using CP15 register 0. See *Register 0, Tightly-coupled Memory Size Register* on page 2-10 for more details.

You can determine the visible size of the Data TCM by using CP15 register 9. See *Register 9, Tightly-coupled Memory Region Registers* on page 2-26 for more details.

Using CP15 register 9, you must ensure the base address of the Data TCM is a value other than `0x0`.

5.2.8 Initializing the Data TCM

You must initialize the Data TCM with the required data image before use.

You can initialize the Data TCM by writing to the memory from the ARM9E-S core data interface.

The Data TCM load mode enables this to be done in an efficient manner. Using the load mode enables you to copy from an address in the data cache or external memory into the same address within the Data TCM.

The Data TCM load mode bit of CP15 register 1 inhibits reads from the Data TCM, forcing reads from addresses that are within the Data TCM address range to access either main memory or the data cache. Writes to addresses that are within the Data TCM range are not affected by the Data TCM load mode bit.

To initialize the Data TCM using the load mode:

1. Enable the Data TCM and Data TCM load mode.
2. Load ARM registers from main memory or data cache.
3. Store ARM registers into data RAM.
4. Increment address pointers and repeat the load and store steps until the data image has been copied.

A suggested assembler code sequence for this procedure is:

```
  LDR R0, #ImageStart            ; Initialize pointer
  LDR R1, =ImageTop              ; Define end of data space
  MRC p15, 0, R2, c1, c0, 0      ; Read Control Register
  ORR R2, R2, #&30000
  MCR p15, 0, R2, c1, c0, 0      ; Enable Data TCM and Load Mode
CopyLoop
  LDMIA R0, {R2 - R9}            ; Load 8 registers from main memory
  STMIA R0!, {R2 - R9}           ; Store 8 regs into Data TCM
  CMP R1, R0                     ; Check if limit reached
  BGT CopyLoop                   ; Repeat if more to do
```

`SWP` and `SWPB` operations to the Data TCM while it is in load mode have Unpredictable results. This is because the read accesses external memory or the data cache, and the write updates the Data TCM.

`SWP` and `SWPB` operations must not be performed to addresses in the Data TCM space while it is in load mode.

5.3 Enabling the Instruction TCM during soft reset

Following a soft reset, you can use the Instruction TCM for the reset vector. This is achieved by using the **INITRAM** signal. If asserted this signal enables the Instruction TCM at reset. The address space allocated for the Instruction TCM defaults to the physical size of the Instruction TCM. To use the reset vector in the Instruction TCM, the memory contents must be preserved during reset. The **VINITHI** signal must be de-asserted so that the reset vector is located at address `0x00000000`.The **INITRAM** signal does not affect the Data TCM, which is disabled at reset.

Note

If **HRESETn** is asserted asynchronously during a soft reset, the firmware designer must ensure that all writes to the TCM have been completed and the TCM interfaces are idling at the time of **HRESETn** assertion. This is necessary for the following reasons:

- to ensure there are no pending writes in the pipeline which would be lost in the case of a reset
- to prevent any loss or corruption of TCM contents by an existing write on the TCM interface.

This can be done by executing a drain write buffer instruction, then entering Standby WFI mode before asserting **HRESETn**. Contact ARM Limited for more details.

5.4 Data TCM accesses

Accesses to the Data TCM do not incur stall cycles unless a write to the Data TCM is completing. Figure 5-2 shows this access.

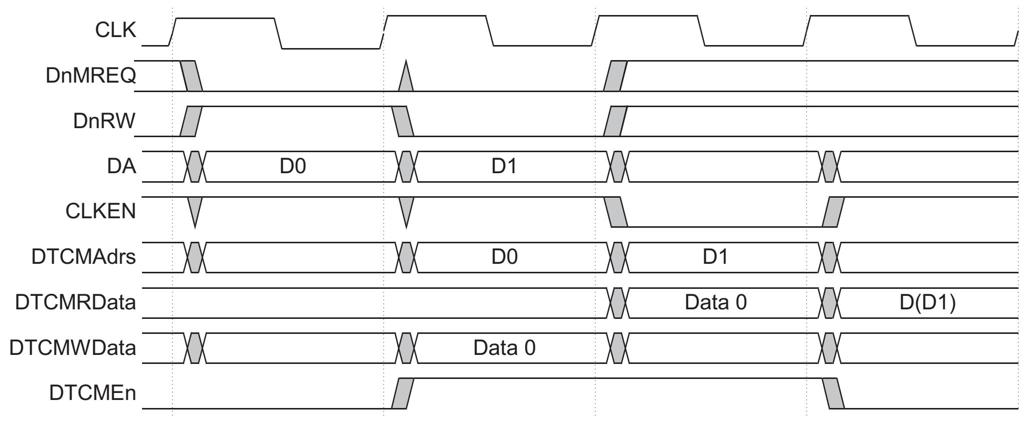


Figure 5-2 Data write followed by data read of Data TCM

5.5 Instruction TCM accesses

The Instruction TCM provides deterministic behavior for time-critical operations, and is located at address `0x00000000` within the processor memory map.The Instruction TCM is implemented using single port synchronous compiled memory.

The protection unit does not have to be enabled for the Instruction TCM to be used.

If the protection unit is enabled then the access permissions programmed into the protection unit are applied to accesses to the Instruction TCM.

The Instruction TCM can be accessed for either instruction fetches or data accesses (read and write) from the ARM946E-S processor.

5.5.1 Instruction accesses to Instruction TCM

Instruction accesses to the Instruction TCM are single-cycle read accesses. No stall cycles are required for instruction accesses to the Instruction TCM unless there is a data access completing.

5.5.2 Data accesses to Instruction TCM

Data accesses to the Instruction TCM can either be reads or writes.

Data access to the Instruction TCM can introduce stall cycles to the ARM946E-S processor.

5.5.3 Stall cycles for Instruction TCM accesses

Simultaneous instruction fetch and data reads of the Instruction TCM incur a single stall cycle. This is because the Instruction TCM is a single port memory, which can only return a single word of memory per clock cycle. This is shown in Figure 5-3 on page 5-10.

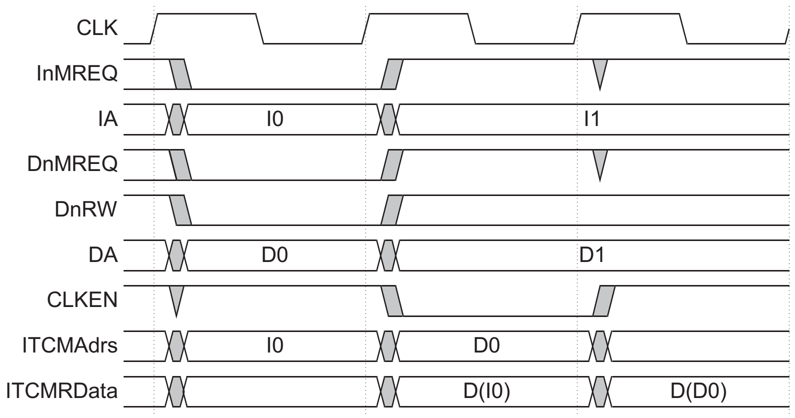


Figure 5-3 Simultaneous instruction fetch and data read of Instruction TCM

A data write to the Instruction TCM followed by a data read from the Instruction TCM incurs a single stall cycle. This is because the memory requires that the write address is pipelined to be in-line with the write data. The read address cannot then be applied until the next cycle, so requiring the stall. Figure 5-4 shows this sequence.

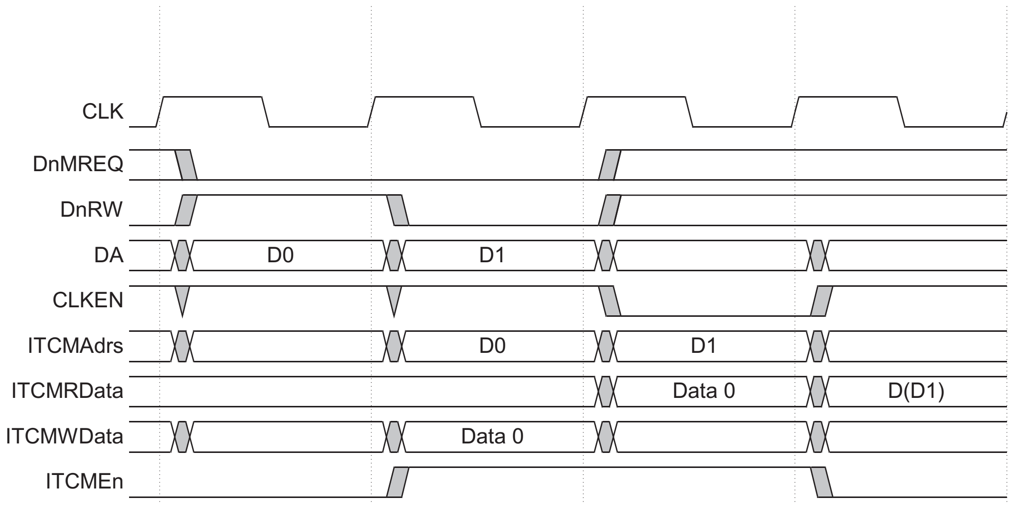


Figure 5-4 Data write followed by data read of Instruction TCM

Similarly, a data write operation followed by an instruction fetch incurs a stall cycle, as shown in Figure 5-5 on page 5-11.

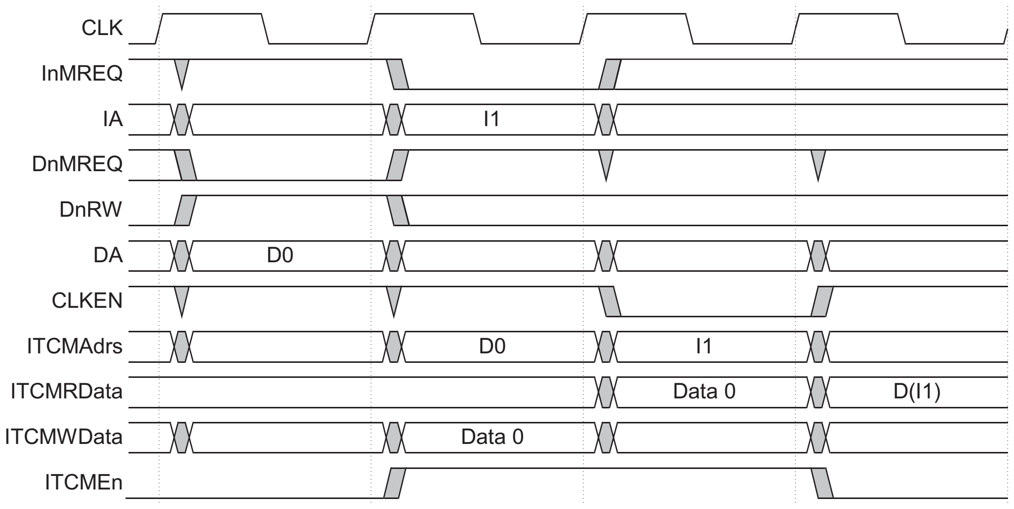


Figure 5-5 Data write followed by instruction fetch of Instruction TCM

A data read followed by an instruction fetch also requires a stall cycle. This stall is incurred as a result of the multiplexor switching being controlled by registered versions of the ARM9E-S data memory interface. The stall is therefore inserted for the data read cycle rather than the instruction read. Figure 5-6 shows the sequence.

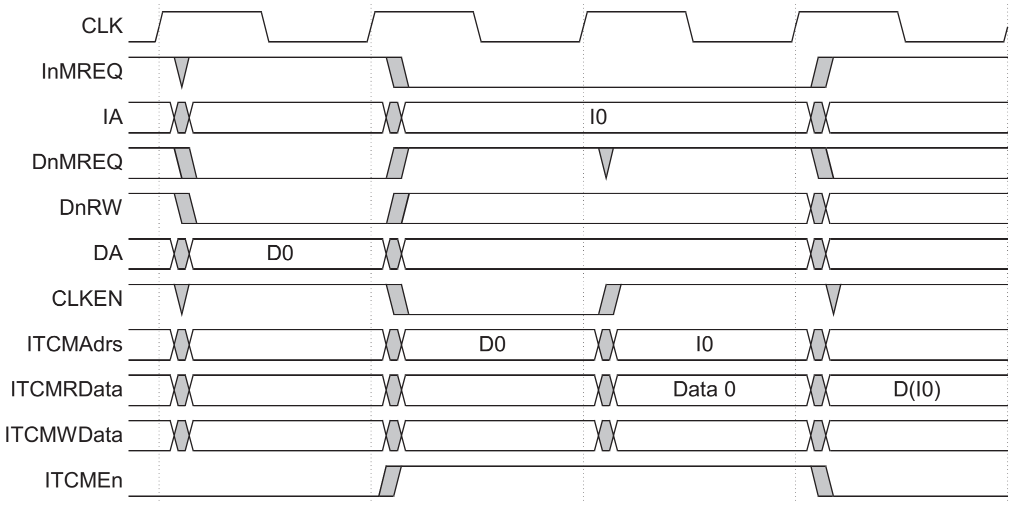


Figure 5-6 Data read followed by instruction fetch

Simultaneous instruction fetch and data write incurs a single stall cycle because of the pipelining of the data access to the data address. Figure 5-7 shows the sequence.

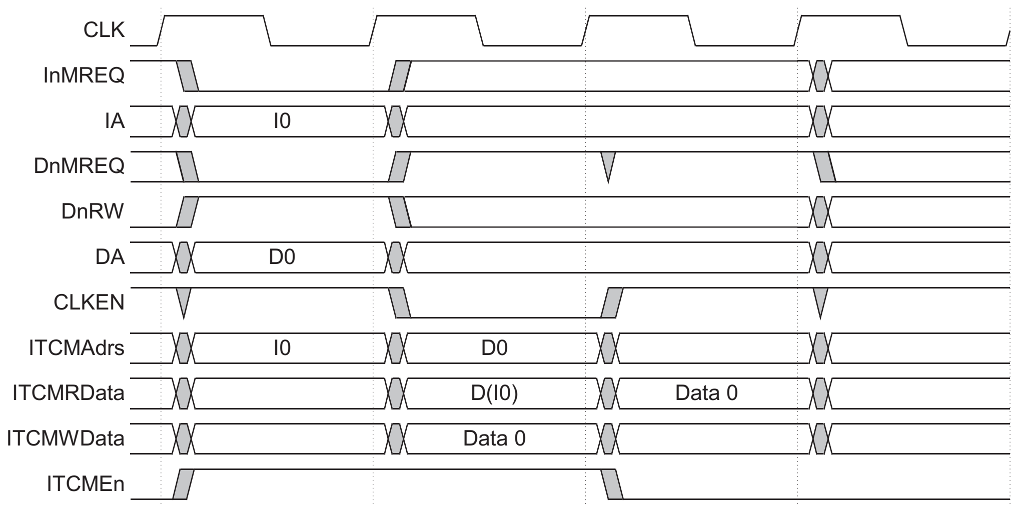


Figure 5-7 Simultaneous instruction fetch and data write

A data write followed by a simultaneous instruction fetch and data read incurs two stall cycles. The first stall is caused by the write still being active when the instruction fetch begins. The second stall is caused by the two reads required. This is shown in Figure 5-8 on page 5-13.

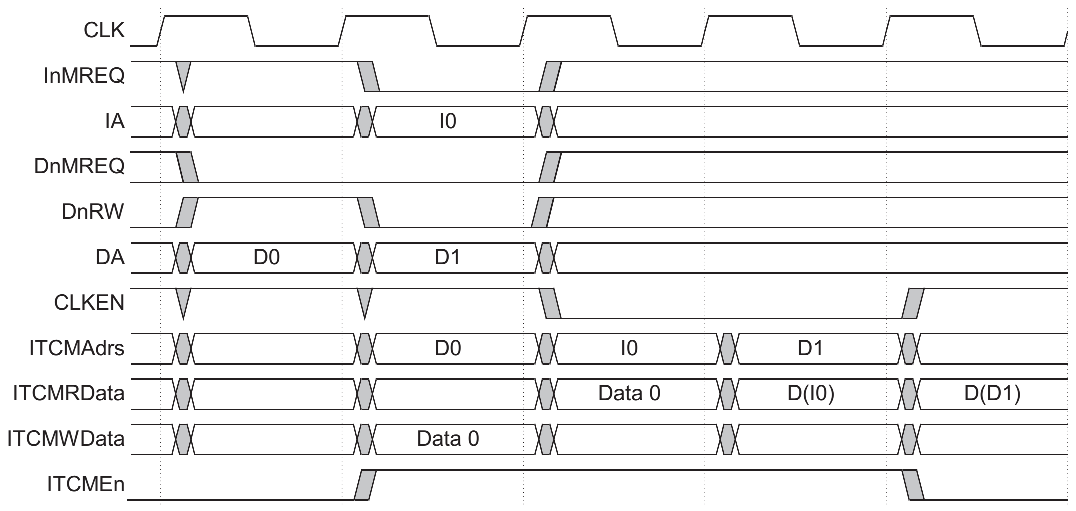


Figure 5-8 Data write followed by simultaneous instruction fetch and data read

Chapter 6
Bus Interface Unit and Write Buffer

This chapter describes the ARM946E-S *Bus Interface Unit* (BIU) and write buffer. It contains the following sections:

6.1 About the BIU and write buffer

The ARM946E-S processor supports the *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 high-performance synthesizable designs, including:

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

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

The ARM946E-S BIU implements a fully-compliant AHB bus master interface and incorporates a write buffer to increase system performance. The BIU is the link between the ARM9E-S core with the caches and *Tightly-Coupled Memory* (TCM) and the external AHB memory. The AHB memory must be accessed for cache linefills and for initializing the TCMs, and to access code and data that are not within the cachable or TCM address regions.

When an AHB access is performed, the BIU and system controller handshake to ensure that the ARM9E-S core is stalled until the access has been performed. If you are using the write buffer, you might be able to enable the core to continue program execution. The BIU controls the write buffer and related stall behavior.

6.2 AHB bus master interface

The ARM946E-S processor implements a fully compliant AHB bus master interface as defined in the *AMBA Rev 2.0 Specification*. See this document for a detailed description of the AHB protocol.

Note

In all timing diagrams in this section, it is assumed that **HCLK** is the same frequency as **CLK**.

6.2.1 About AHB

The AHB architecture is based on separate cycles for address and data. The address and control 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 for the next transfer are driven out. This leads to a fully pipelined address architecture.

When an access is in its data cycle, a slave can extend an access by driving the **HREADY** signal LOW. This stretches the current data cycle, and therefore the pipelined address and control for the next transfer is also stretched. This provides a system where all AHB masters and slaves sample **HREADY** on the rising edge of **HCLK** to determine whether an access has completed and a new address can be sampled or driven out.

6.2.2 Transfer type

Table 6-1 shows the transfer types that can be generated by the ARM946E-S processor from the **HTRANS[1:0]** signal.

Table 6-1 AHB transfer types

Transfer type	HTRANS[1:0]	Description
IDLE	b00	Indicates that no data transfer is required. The IDLE transfer is used when a bus master is granted the bus, but does not want to perform a data transfer. Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer must be ignored by the slave.
BUSY	b01	The BUSY transfer enables bus masters to insert idle cycles in the middle of bursts of transfers. This transfer indicates that the bus master is continuing with a burst of transfers, but the next transfer cannot take place immediately. When a master uses the BUSY transfer the address and control signals must reflect the next transfer in the burst. The transfer must be ignored by the slave. Slaves must always provide a zero wait state OKAY response, in the same way that they respond to IDLE transfers. Examples of where the ARM946E-S uses BUSY cycles are: • during debug and coprocessor operations to uncachable areas of memory • LDM accesses to uncachable areas of memory depending on the start address of the burst.
NONSEQ	b10	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of one and therefore the transfer type is NONSEQUENTIAL.
SEQ	b11	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes).

——— **Note** ———

BUSY transfers are inserted between certain sequences of NONSEQ and SEQ transfers. Examples of transfers that can cause BUSY transfers include multiple data reads during debug and coprocessor operations to uncachable areas of memory. LDM accesses to uncachable areas of memory might also cause BUSY transfers depending on the start address of the burst.

System designers must ensure that any AHB peripherals can handle BUSY transfers as defined in the *AMBA Specification.*

6.2.3 Burst sizes

The ARM946E-S processor supports the burst types shown in Table 6-2.

Table 6-2 Supported burst types

Burst type	HBURST[2:0] encoding	Use
SINGLE	b000	Single writes (`STR/STRH/STRB`) Uncached single reads Uncached instruction fetches
INCR	b001	Store multiple (`STM`) Uncached burst reads (`LDM`)
INCR4	b011	Dirty half-cache line Write-Back
INCR8	b101	Dirty cache line Write-Back Cache linefetches

Incrementing bursts have an address increment of four, that is, word increment.

6.2.4 Linefetch transfers

The ARM946E-S processor is optimized to run with the instruction cache and data cache enabled. If a memory request (either instruction or data) to a cachable area misses in the cache the ARM946E-S processor performs a linefetch.

Figure 6-1 shows a linefetch transfer.

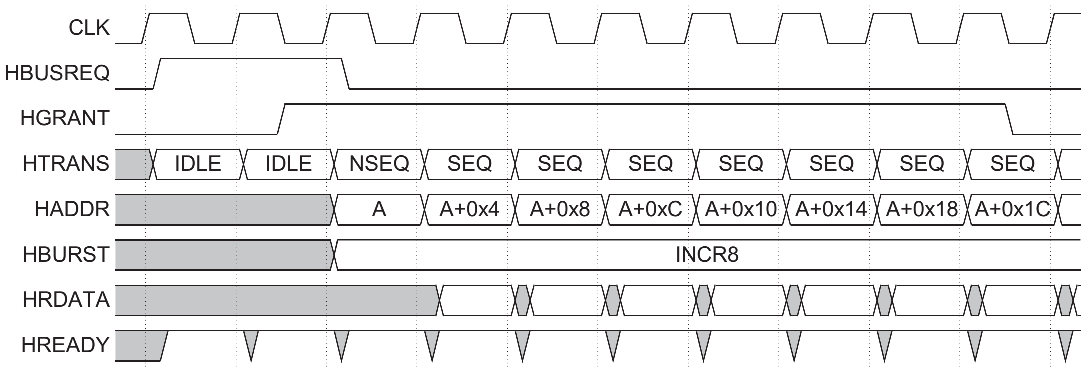


Figure 6-1 Linefetch transfer

A linefetch is a fixed length burst of eight words. The start address of a linefetch is aligned to an eight-word boundary. The ARM946E-S processor asserts the bus request **HBUSREQ** until the arbiter grants the AHB bus (**HGRANT** asserted). The bus request is then negated. This enables optimum system performance because the arbiter can accurately predict the end of the defined length burst.

6.2.5 Back to back linefetches

The ARM946E-S processor supports streaming of data and instructions (core execution is advanced during the linefetch). To enable for cache look-ups when crossing a cache line boundary the ARM946E-S processor must insert IDLE cycles onto the AHB bus. Figure 6-2 shows the effect of this.

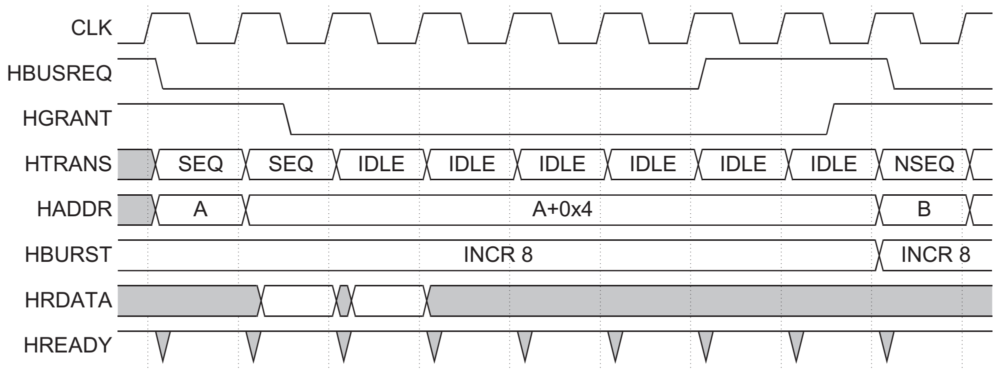


Figure 6-2 Back-to-back line fetches

6.2.6 Uncached transfers

If a memory request is made to an uncachable region, or the ARM946E-S cache is not enabled, the memory requests are serviced by the AHB interface. Sequential instruction fetches are treated as nonsequential reads.

Figure 6-3 on page 6-7 shows uncached instruction fetches. Nonsequential uncached data operations exhibit similar bus timings.

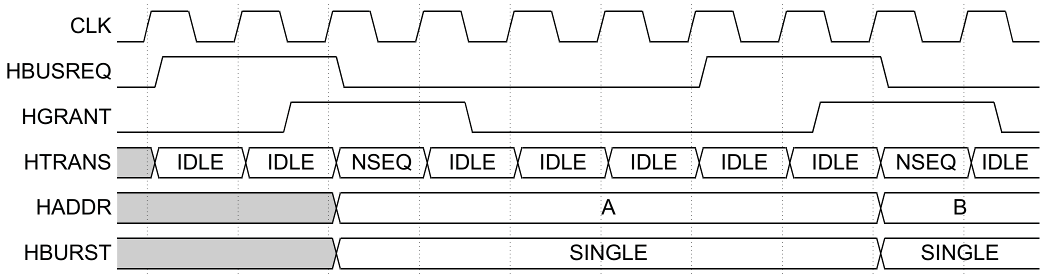


Figure 6-3 Nonsequential uncached accesses

6.2.7 Burst accesses

Uncached burst operations (`STM`/`LDM` instructions) are performed as incrementing bursts of undefined length on the AHB.

Figure 6-4 shows a data burst followed by an uncached instruction fetch.

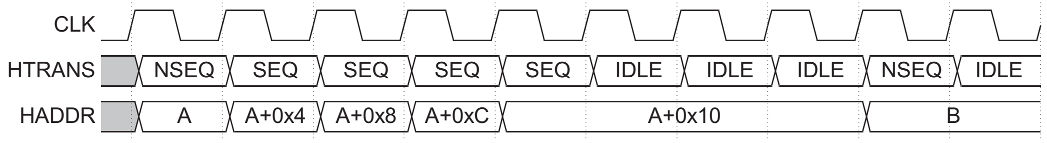


Figure 6-4 Data burst followed by instruction fetch

6.2.8 Bursts crossing 1KB boundary

The AHB specification requires that bursts must not continue across a 1KB boundary. Linefetches and cache line Write-Backs cannot cross a 1KB boundary because the start address is aligned to either a four or eight-word boundary, and the burst length is fixed.

Uncached data bursts can cross a 1KB boundary. Figure 6-5 shows an example of this. The burst is restarted by inserting a nonsequential transfer as the boundary is crossed.

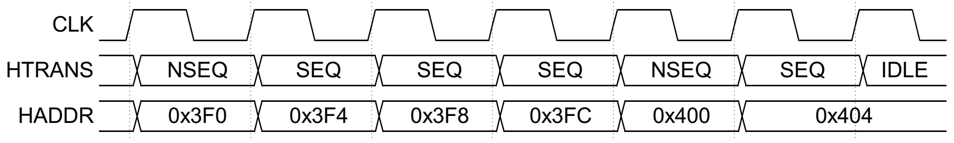


Figure 6-5 Crossing a 1KB boundary

6.2.9 Uncached LDC operations

Coprocessor loads to its registers from memory are shown in Figure 6-6. **DnMREQ**, **DMORE**, **CLKEN**, and **RDATA** are internal ARM946E-S signals. See the ARM9E-S *Technical Reference Manual* for more information about these signals. The sequence assumes that the ARM946E-S processor has already been granted bus ownership.

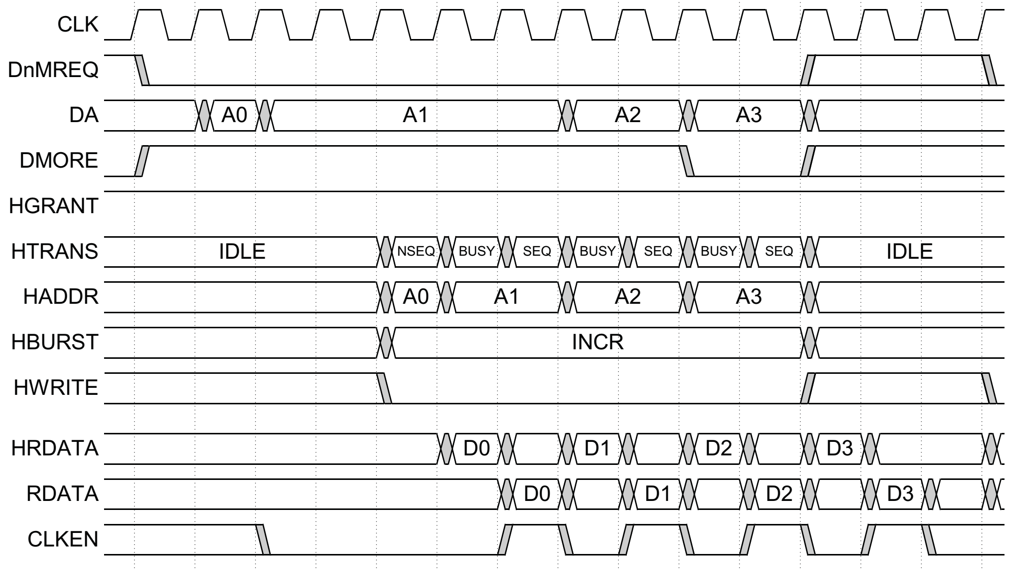


Figure 6-6 Uncached LDC sequence

6.3 Noncached Thumb instruction fetches

Thumb instruction fetches are performed as 32-bit accesses on the AHB interface. To minimize bus loading, AHB transfers are only performed for nonsequential addresses and for sequential addresses that cross a word boundary. The word returned from main memory is latched so that both halfwords are available for the processor core.

6.4 AHB clocking

The ARM946E-S processor design uses a single rising-edge clock **CLK** to time all internal activity. In many systems in which the ARM946E-S processor is embedded, you might prefer to run the AHB at a lower rate. To support this requirement, the ARM946E-S processor requires a clock enable, **HCLKEN**, to time AHB transfers.

The **HCLKEN** input is driven HIGH around a rising edge of the ARM946E-S processor **CLK** to indicate that this rising-edge is also a rising-edge of **HCLK** so must be synchronous to the ARM946E-S processor **CLK**.

When the ARM9E-S is running from Tightly-Coupled Memory (TCM) or performing writes using the write buffer, the ARM946E-S processor **HCLKEN** and **HREADY** inputs are not used to generate the **SYSCLKEN** core stall signal. The core is only stalled by TCM stall cycles or if the write buffer overflows. This means that the ARM9E-S is executing instructions at the faster **CLK** rate and is effectively decoupled from the **HCLK** domain AHB system.

If, however, you want to perform an AHB read access or unbuffered write, the core is stalled until the AHB transfer has completed. When the AHB system is being clocked by the lower rate **HCLK**, **HCLKEN** is examined to detect when to drive out the AHB address and control to start an AHB transfer. **HCLKEN** is then required to detect the following rising edges of **HCLK** so that the BIU knows the access has completed.

If the slave being accessed at the **HCLK** rate has a multi-cycle response, the **HREADY** input to the ARM946E-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 has completed. When this is the case, the ARM9E-S core is enabled by reasserting **SYSCLKEN**.

Note

When an AHB access is required, the core is stalled until the next **HCLKEN** pulse is received, before it can start the access, and then until the access has completed. This stall before the start of the access is a synchronization penalty and the worst case can be expressed in **CLK** cycles as the **HCLK** to **CLK** ratio minus 1.

6.4.1 CLK to HCLK skew

The ARM946E-S processor drives out the AHB address on the rising edge of **CLK** when the **HCLKEN** input is HIGH. The AHB outputs therefore have output hold and delay values relative to **CLK**. However, these outputs are used in the AHB system where transfers are timed using **HCLK**. Similarly, inputs to the ARM946E-S processor are timed relative to **HCLK** but are sampled within the ARM946E-S processor with

CLK. This leads to hold time issues, from **CLK** to **HCLK** on outputs, and from **HCLK** to **CLK** on inputs. To minimize this effect you must minimize the skew between **HCLK** and **CLK**.

Figure 6-7 shows the AHB clock relationships.

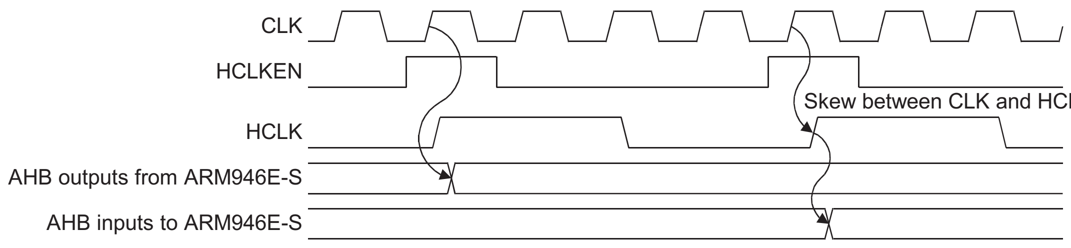


Figure 6-7 AHB clock relationships

Clock tree insertion at top level

Considering the skew issue in more detail, the ARM946E-S processor requires a clock tree to be 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 therefore relative to **CLK**, at the bottom of the inserted clock tree and subject to the clock tree insertion delay. To maximize performance, when the ARM946E-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 ARM946E-S processor. You can achieve this using a clock tree insertion tool, if the clock tree is inserted for the ARM946E-S processor and the embedded system at the same time (top level insertion).

Figure 6-8 on page 6-12 shows an example of an AHB slave connected to the ARM946E-S processor.

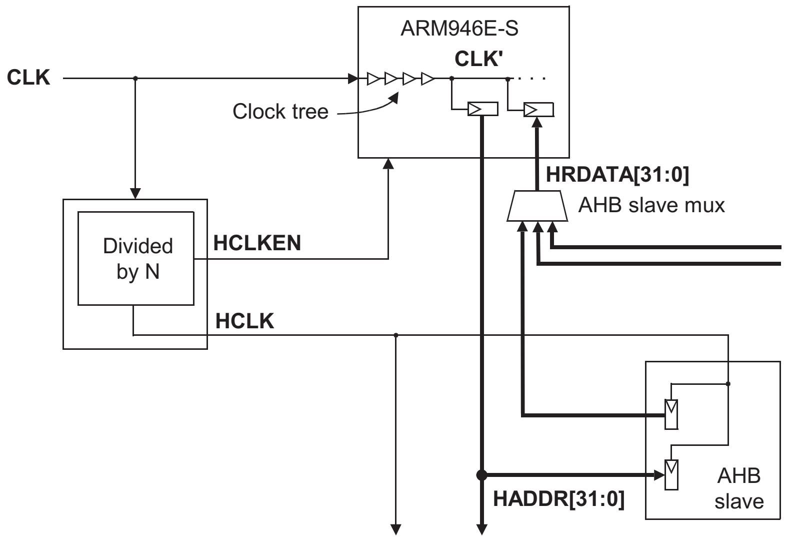


Figure 6-8 ARM946E-S CLK to AHB HCLK sampling

In Figure 6-8, the slave peripheral has an input setup and hold, and an output hold and valid time relative to **HCLK**. The ARM946E-S processor has an input setup and hold, and an output hold and valid time relative to **CLK'**, the clock at the bottom of the clock tree. You can use clock tree insertion to position **HCLK** to match **CLK'** for optimal performance.

Hierarchical clock tree insertion

If you perform clock tree insertion on the ARM946E-S processor before it is embedded, you can add buffers 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 ARM946E-S processor input setup and hold requirements. Because the ARM946E-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.

6.5 The write buffer

The ARM946E-S processor provides a write buffer to improve system performance. The write buffer has a 16-entry FIFO. Each entry can be either address or data. The type of entry is determined by the setting of an address/data flag. Each address entry is tagged with the size of transfer, as indicated by the ARM9E-S core (byte, halfword, or word).

Write buffer behavior is controlled by the protection region attributes of the store being performed and the data cache and protection unit enable status. This control is represented by the data *Cachable bit* (Cd) and the write *Buffer control bit* (Bd) from the protection unit. These control bits are generated as follows:

Cd bit	This is generated from the cachable attribute of the protection region AND the data cache enable AND the protection unit enable.
Bd bit	This is generated from the bufferable attribute for the protection region AND the protection unit enable.

All accesses are initially noncachable and nonbufferable until you have programmed and enabled the protection unit. Therefore, you cannot use the write buffer while the protection unit is disabled.

On reset, all entries in the write buffer are invalidated.

6.5.1 Write buffer operation

The write buffer is used when the data cache hits and/or misses, depending on the mode of operation. Table 6-3 shows how the Cd and Bd bits control the behavior of the write buffer.

Table 6-3 Data write modes

Cd	Bd	Access mode
0	0	NCNB (noncachable, nonbufferable)
0	1	NCB (noncachable, bufferable)
1	0	WT (write-through)
1	1	WB (write-back)

NCNB	Data reads and writes are not cached, and can be externally aborted. Writes are not buffered, so the processor is stalled until the external access is performed. NCNB reads bypass the write buffer.

NCB	Data reads and writes are not cached. Writes are buffered, and so cannot be externally aborted. Reads can be externally aborted. Reads cause the write buffer to drain. If the data cache hits for this type of access, there has been a programming error. data cache hits are ignored and the data cache line is not updated for a read. Swap instructions operating on data in an NCB region are made to perform NCNB type accesses and are *not* buffered.
WT	Searches the data cache for reads and writes. Reads that miss in the data cache cause a line fill. Reads that hit in the data cache do not perform an external access. All writes are buffered, regardless of whether they hit or miss in the data cache. Writes that hit in the data cache update the cache but do not mark the cache line as dirty, because the write is also sent to the write buffer. Writes cannot be externally aborted. Data cache linefills cause the write buffer to drain before the linefill starts.
WB	Searches the data cache for reads and writes. Reads that miss in the data cache cause a line fill. Reads that hit in the data cache do not perform an external access. Writes that miss in the data cache are buffered. Writes that hit in the data cache update the cache line, mark it as dirty, and do not send the data to the write buffer. Data cache write-backs are buffered. Writes (write-miss and write-back) cannot be externally aborted. Data cache linefills cause the write buffer to drain before the linefill starts.

6.5.2 Enabling and disabling the write buffer

You cannot directly enable or disable the write buffer. However, you can prevent the write buffer being used by setting the properties of a memory region to be NCNB, or by disabling the protection unit.

6.5.3 Self-modifying code

Instruction fetches and NCNB reads bypass the write buffer. If you write self-modifying code to a bufferable or cachable region, then it is essential that you drain the write buffer before fetching instructions from these addresses.

Chapter 7
Coprocessor Interface

This chapter describes the ARM946E-S pipelined coprocessor interface. It contains the following sections:

7.1 About the coprocessor interface

The ARM946E-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 ARM946E-S processor to the coprocessor has been pipelined by a single clock cycle as shown in Figure 7-1.

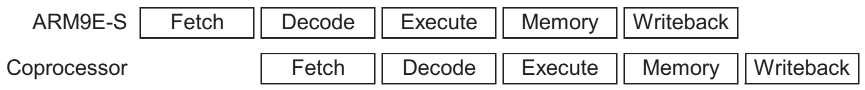


Figure 7-1 Pipeline stages

This ensures that ARM946E-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.

7.2 Coprocessor interface signals

Table 7-1 describes the ARM946E-S coprocessor interface signals.

Table 7-1 Coprocessor interface signals

Name	Direction with respect to ARM946E-S processor	Description
CPCLKEN Coprocessor clock enable	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] Coprocessor instruction data	Output	The 32-bit coprocessor instruction bus over which instructions are transferred to the coprocessor pipeline follower.
CPDOUT[31:0] Coprocessor read data	Output	The 32-bit coprocessor read data bus for transferring data to the coprocessor.
CPDIN[31:0] Coprocessor write data	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]
Coprocessor handshake decode
	Input	The handshake signals from the Decode stage of the coprocessors pipeline follower. Indicates:
		b10 = ABSENT
		b00 = WAIT
		b01 = GO
		b11 = LAST.

CHSEX[1:0]
Coprocessor handshake execute
	Input	The handshake signals from the Execute stage of the coprocessors pipeline follower. Indicates:
		b10 = ABSENT
		b00 = WAIT
		b01 = GO
		b11 = LAST.

Table 7-1 Coprocessor interface signals (continued)

Name	Direction with respect to ARM946E-S processor	Description
CPTBIT Coprocessor instruction Thumb bit	Output	When HIGH indicates that the ARM946E-S processor is in Thumb state. When LOW indicates that the ARM946E-S processor is in ARM state. Sampled by the coprocessor pipeline follower.
nCPMREQ Not coprocessor instruction request	Output	When LOW on the rising edge of **CLK** and **CPCLKEN** is HIGH, the instruction on **CPINSTR** must enter the coprocessor pipeline.
nCPTRANS Not coprocessor memory translate	Output	When LOW indicates that the ARM946E-S processor is in User mode. When HIGH indicates that the ARM946E-S processor is in privileged mode. Sampled by the coprocessor pipeline follower.

7.2.1 Synchronizing the external coprocessor pipeline

A coprocessor connected to the ARM946E-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 ARM946E-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 ARM946E-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 ARM946E-S processor without modification. The data path of the coprocessor differs however, because of the ARM946E-S pipelined output data **CPDOUT[31:0]**.

7.2.2 External coprocessor clocking

The *Coprocessor Data Processing* (`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 is stalled (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 is stalled 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 is only stalled when the current instruction leaves Execute and new instructions are required from the ARM946E-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 7-10
- *MCR/MRC* on page 7-12
- *CDP* on page 7-14.

7.2.3 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/STC` instruction, the coprocessor instruction drives the handshake signals with GO when two or more words still have to be transferred. When only one more word is to 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` instruction. Also in this cycle, **DnMREQ** is driven LOW, indicating to the ARM946E-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 7-6 on page 7-10.

LAST An `LDC` or `STC` instruction 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 penultimate cycle LAST (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.

7.2.4 Coprocessor handshake encoding

Table 7-2 shows how the handshake signals **CHSDE[1:0]** and **CHSEX[1:0]** are encoded.

Table 7-2 Handshake encoding

[1:0]	Meaning
b10	ABSENT
b00	WAIT
b01	GO
b11	LAST

Note

If an external coprocessor is not attached in the ARM946E-S embedded system, the **CHSDE[1:0]** and **CHSEX[1:0]** handshake inputs must be tied off to indicate ABSENT.

7.2.5 Multiple external coprocessors

Figure 7-2 shows an example where VFP9 and two other coprocessors are connected to the ARM946E-S processor using the coprocessor interface logic block.

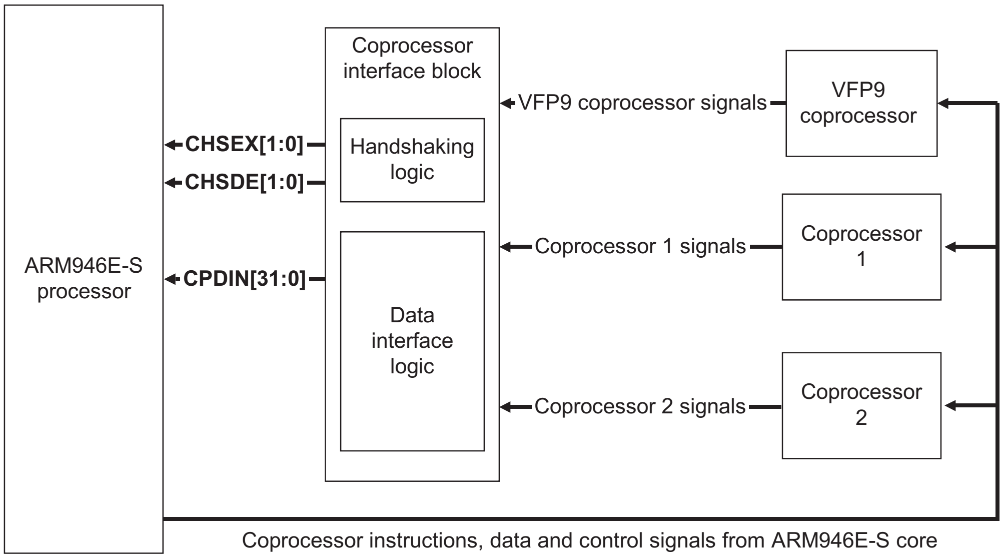


Figure 7-2 Connecting multiple coprocessors

The handshaking signals from the coprocessors can be combined by ANDing bit 1, and ORing bit 0.

In the case of coprocessors that have handshaking signals **CHSDECP1**, **CHSEXCP1**, **CHSDECP2**, **CHSEXCP2**, and **CHSDEVFP9**, **CHSEXVFP9** use:

CHSDE[1] = **CHSDECP1[1]** AND **CHSDECP2[1]** AND **CHSDEVFP9[1]**

CHSDE[0] = **CHSDECP1[0]** OR **CHSDECP2[0]** OR CHSDEVFP9[0]

CHSEX[1] = **CHSEXCP1[1]** AND **CHSEXCP2[1]** AND CHSEXVFP9[1]

CHSEX[0] = **CHSEXCP1[0]** OR **CHSEXCP2[0]** OR CHSEXVFP9[0]

Figure 7-3 shows example components of the handshaking logic in the coprocessor interface logic block.

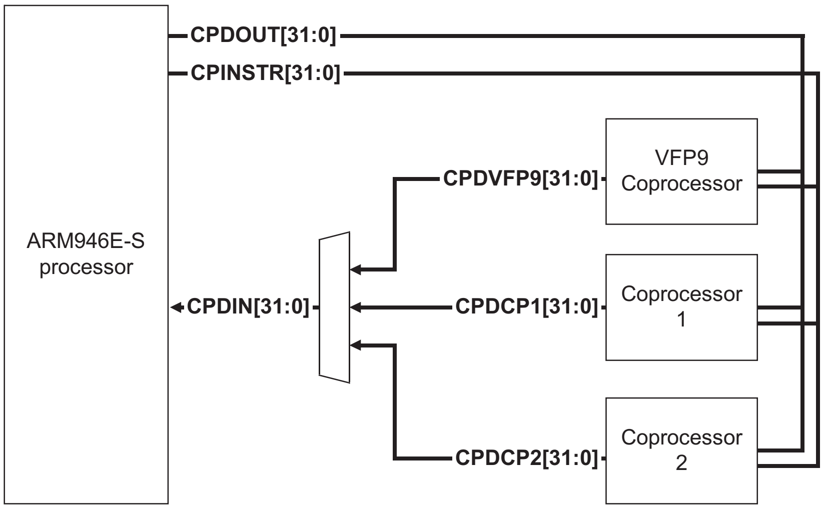


Figure 7-3 Example handshake logic blocks

For connecting to the **CPDIN[31:0]** signals, there are two options for interfacing the coprocessor data buses to the ARM946E-S processor:

- The coprocessor drives its data bus to logic 0 when not selected. This enables a simple OR connection scheme as shown in Figure 7-4 on page 7-9. This is the recommended method of coprocessor data bus interfacing.

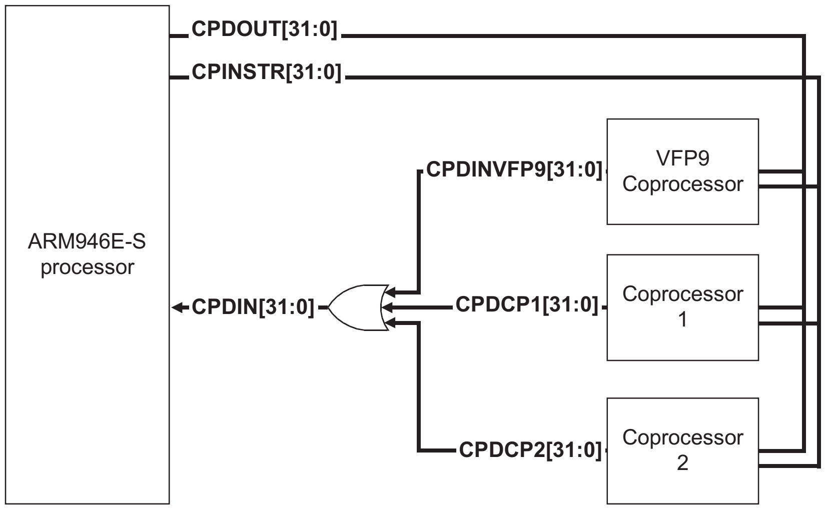


Figure 7-4 Driving the coprocessors data buses to logic 0

- Multiplexing the coprocessor data bus, as shown in Figure 7-5. For coprocessors that do not drive data buses to logic 0 a multiplexor circuit is required. Multiplexor control is determined by the coprocessor decoding the coprocessor number field Bits [11:7] in the `MRC/STC` instruction in the correct pipeline stage.

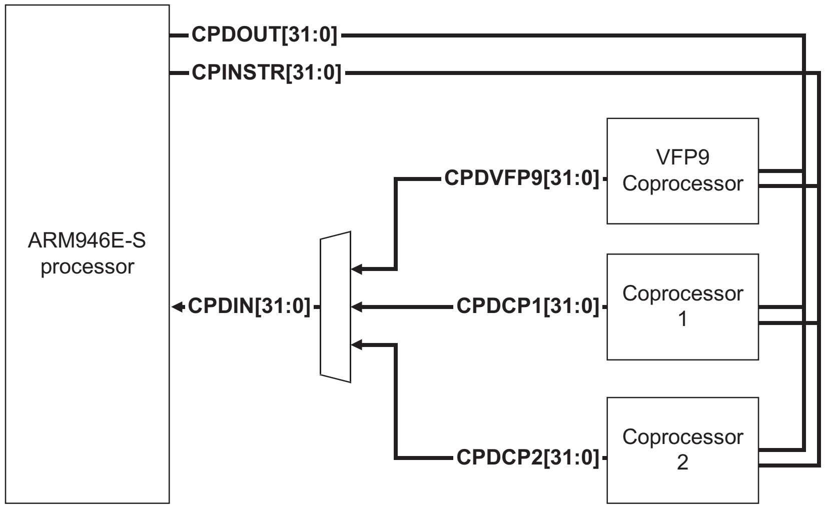


Figure 7-5 Multiplexing the coprocessors data buses

7.3 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 ARM946E-S processor, the memory can be either cache, *Tightly-Coupled Memory* (TCM) or AHB depending on the address range of the access and the protection unit settings.

Figure 7-6 shows the cycle timing for these operations.

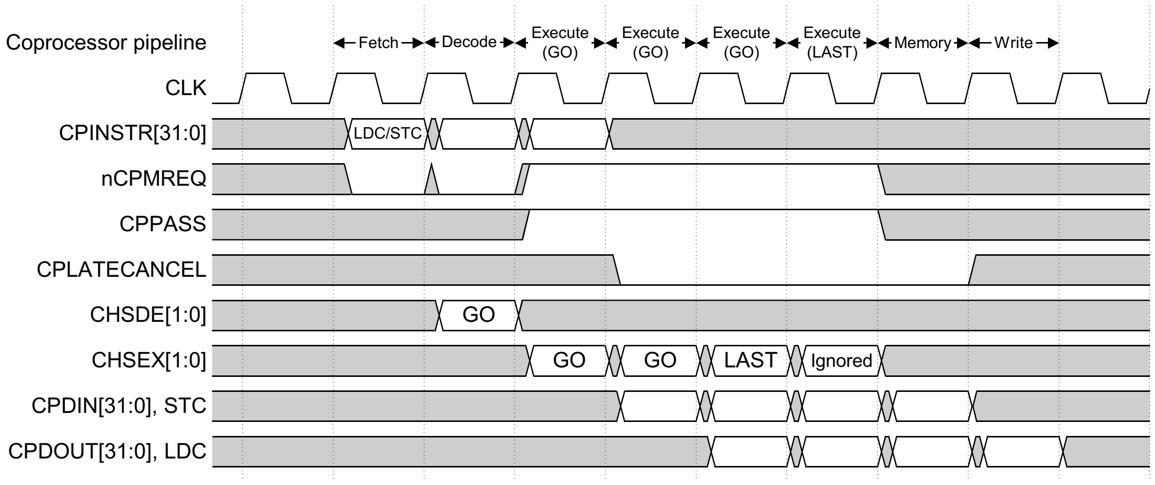


Figure 7-6 LDC/STC cycle timing

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**. This 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 is taking 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 is stalled, and the coprocessor pipeline must not advance.

During the Execute stage, the condition codes are compared with the flags to determine whether the instruction really executes or not. 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 busy-waits, **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 any coprocessor instruction. 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 is entering 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]**.

7.4 MCR/MRC

These cycles look very similar to `STC`/`LDC`. Figure 7-7 shows an example with a busy-wait state. 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 has now been issued to the Execute stage. If the condition codes pass, 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).

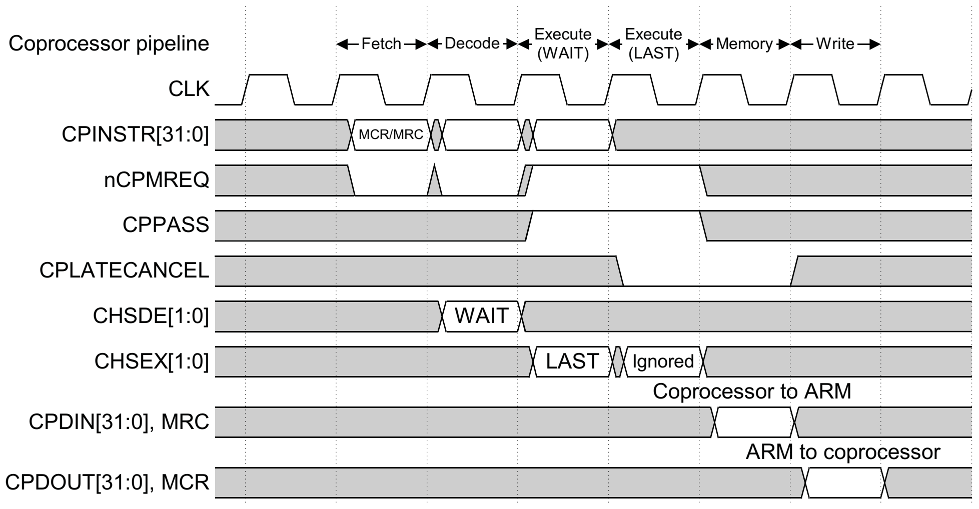


Figure 7-7 MCR/MRC transfer timing with busy-wait

For any successive Execute cycles the **CHSEX[1:0]** handshake bus is examined. When the LAST condition is observed, the instruction is committed. In the case of a `MCR`, the **CPDOUT[31:0]** bus is driven with the registered data. In the case of a `MRC`, **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.

7.5 Interlocked MCR

If the data for a 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 from 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 7-8 shows an example of an interlocked MCR that also has a busy-wait state.

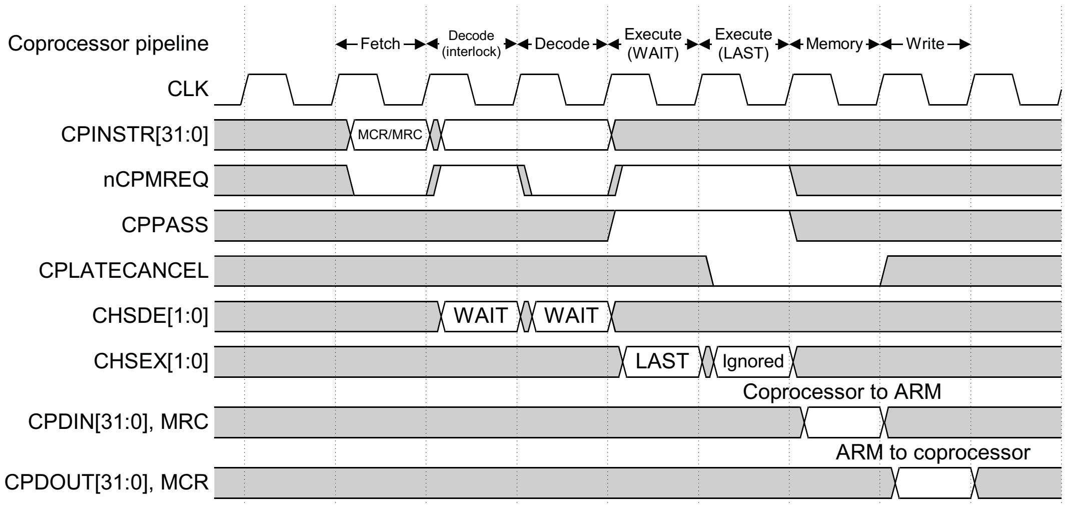


Figure 7-8 Interlocked MCR/MRC timing with busy-wait

7.6 CDP

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

- if the instruction really 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 7-9 shows a canceled CDP because of the previous instruction causing a Data Abort.

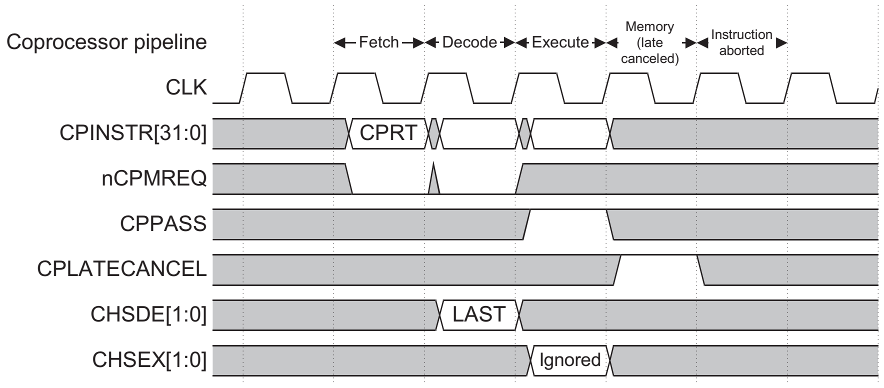


Figure 7-9 Late canceled CDP

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 and no state changes are made to the coprocessor.

7.7 Privileged instructions

The coprocessor restricts certain instructions for use in privileged modes only. To do this, the coprocessor tracks the **nCPTRANS** output. Figure 7-10 shows how **nCPTRANS** changes after a mode change.

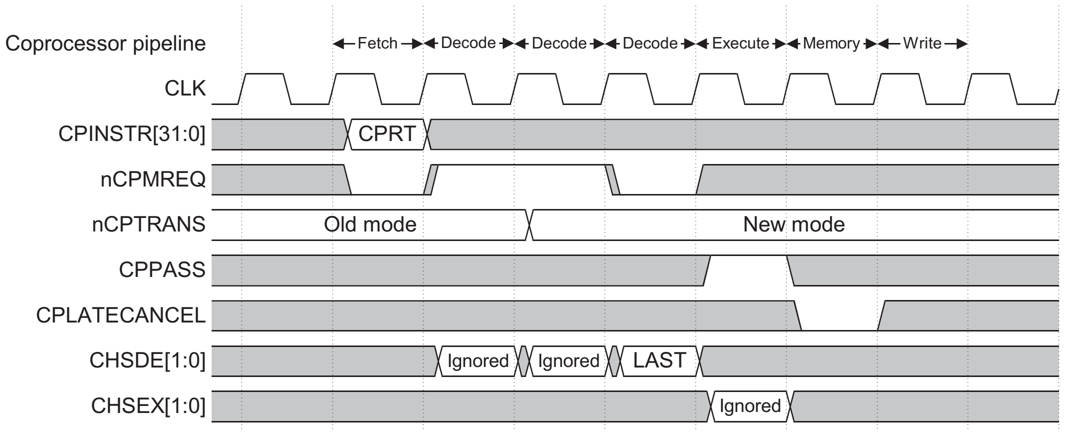


Figure 7-10 Privileged instructions

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

7.8 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 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 7-11 shows a busy-waited coprocessor instruction being abandoned because of an interrupt. **CPLATECANCEL** is also asserted as a result of the Execute interruption.

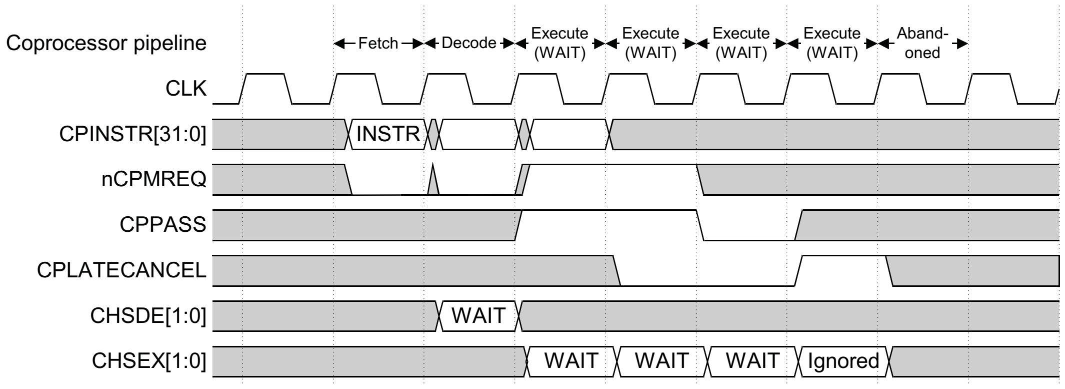


Figure 7-11 Busy-waiting and interrupts

Chapter 8
ETM Interface

This chapter describes the ARM946E-S *Embedded Trace Macrocell* (ETM) interface. It contains the following sections:

8.1 About the ETM interface

The ARM946E-S processor supports the connection of an optional external *Embedded Trace Macrocell* (ETM) to provide real-time tracing of ARM946E-S instructions and data in an embedded system.

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 ARM946E-S processor from the ETM interface registers, as shown in Figure 8-1.

The ETM interface outputs are pipelined by a single clock cycle to provide early output timing and to isolate any ETM input load from the critical ARM946E-S processor signals. The latency of the pipelined outputs does not affect ETM trace behavior, because all outputs are delayed by the same amount.

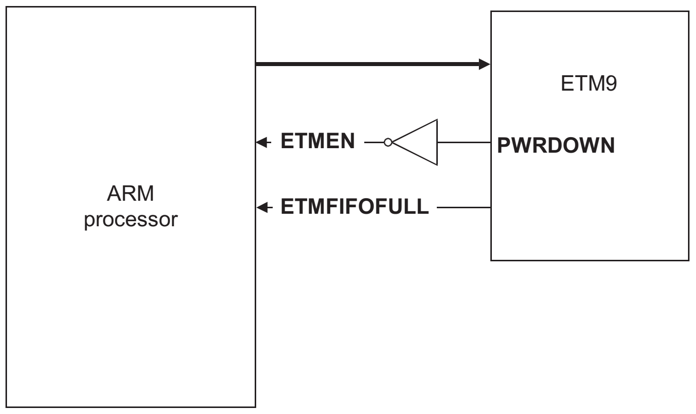


Figure 8-1 ARM946E-S ETM interface

8.2 Enabling the ETM interface

The ETM interface on the ARM946E-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.

It is recommended that the **ETMEN** input is connected 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 ARM946E-S design, the **ETMEN** input must be tied LOW to save power.

8.3 ARM946E-S trace support features

The ARM946E-S processor includes the following trace support features:

- *ETMFIFOFULL*
- *Register 15, Trace Control Register*
- *Register 13, Trace Process Identifier Register*.

8.3.1 ETMFIFOFULL

The signal, **ETMFIFOFULL**, is an input to the ARM946E-S processor driven by the ETM9. Whenever the programmed upper watermark of the ETM FIFO is reached, **ETMFIFOFULL** is asserted. The ARM946E-S processor uses **ETMFIFOFULL** to stall the ARM9E-S core, preventing trace loss. The ARM9E-S core remains stalled until **ETMFIFOFULL** is deasserted.

The ARM946E-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 current instruction is either an `LDM` or an `STM`, the FIFO might have to accept up to 16 words after the assertion of **ETMFIFOFULL**.

Note

Using **ETMFIFOFULL** to stall the ARM946E-S processor affects real-time operating performance.

8.3.2 Register 15, Trace Control Register

The Trace Control Register enables **nIRQ** and **nFIQ** interrupt priority over **ETMFIFOFULL** to be programmed. The operation of this register is described in *Register 15, Trace Control Register* on page 2-35.

8.3.3 Register 13, Trace Process Identifier Register

The ARM946E-S processor contains a Trace Process Identifier Register that enables real-time trace tools to identify the currently executing process in multi-tasking environments. The operation of this register is described in *Register 13, Trace Process Identifier Register* on page 2-28.

Chapter 9
Debug Support

This chapter describes the ARM946E-S debug interface. It contains the following sections:

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

9.1 About the debug interface

Debug support is implemented using the ARM9E-S core embedded within the ARM946E-S processor. The ARM946E-S processor 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 ARM946E-S processor contains hardware extensions for advanced debugging features. These make it easier to develop application software, operating systems, and the hardware itself. The ARM9E-S core supports two debug modes:

- *Halt mode*
- *Monitor mode*.

9.1.1 Halt mode

The debug extensions enable you to force the core to be stopped and placed in debug state by:

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

In debug state, the core and ARM946E-S processor memory system are effectively stopped, and isolated from the rest of the system. This is known as *halt mode* operation and enables you to examine the internal state of the ARM9E-S core, ARM946E-S processor, and external AHB state, while all other system activity continues as normal. When debug has been completed, the ARM9E-S restores the core and system state, and resumes program execution.

The examination of the internal state of the ARM946E-S processor uses a JTAG-style interface, that enables you to serially insert instructions into the instruction pipeline. This exports the contents of the ARM9E-S core registers. The exported data is serially shifted out without affecting the rest of the system.

9.1.2 Monitor mode

The ARM9E-S also supports monitor mode where on a breakpoint or watchpoint, an internal Instruction Abort or Data Abort is generated.

When used in conjunction with a debug monitor program activated by the abort exception entry, you can debug the ARM946E-S processor while the execution of critical interrupt service routines continues.

The debug monitor program typically communicates with the debug host over the ARM946E-S debug communication channel. Real-time debug is described in *Monitor mode debugging* on page 9-33.

9.1.3 Debug clocks

You must synchronize the system and test clocks externally to the ARM946E-S processor. The ARM Multi-ICE debug agent directly supports one or more cores within an ASIC design. To synchronize off-chip debug clocking with the ARM946E-S processor you must use a three-stage synchronizer. The off-chip device (for example, Multi-ICE) issues a **TCK** signal, and waits for the **RTCK** (Returned **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 9-1 shows this synchronization.

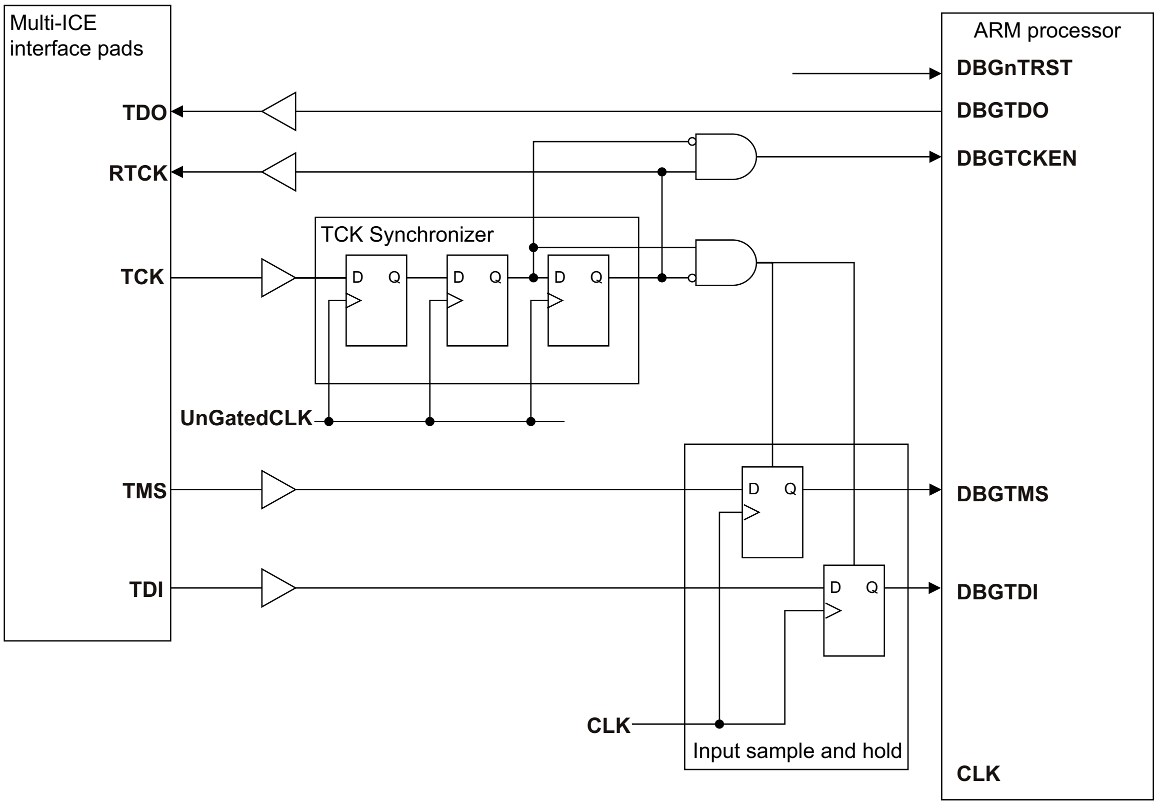


Figure 9-1 Clock synchronization

Note

In Figure 9-1 on page 9-3, **CLK** and **UnGatedClk** must have the same clock frequency.

9.2 Debug systems

The ARM946E-S processor forms one component of a debug system that interfaces from the high-level debugging performed by the user to the low-level interface supported by the ARM946E-S processor. Figure 9-2 shows a typical debug system.

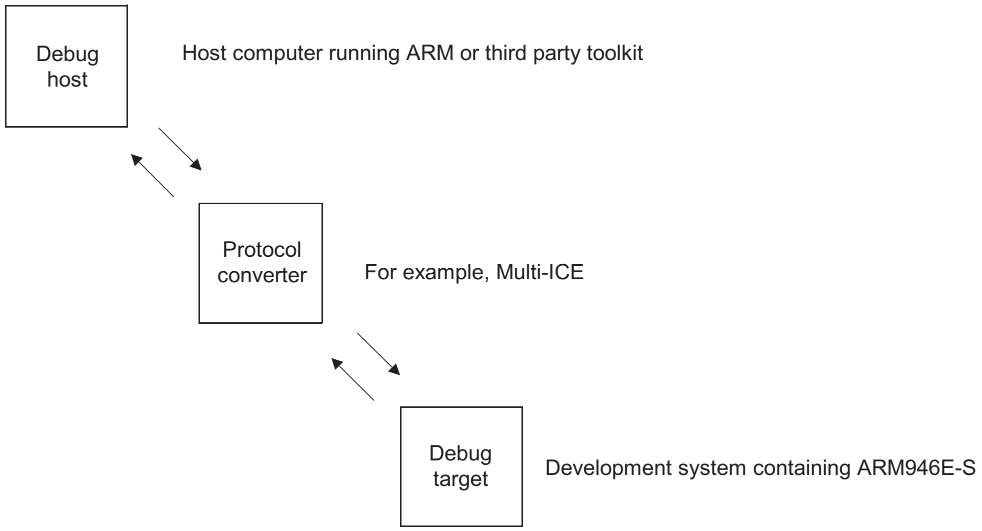

Debug host
Host computer running ARM or third party toolkit
Protocol converter
For example, Multi-ICE
Debug target
Development system containing ARM946E-S

Figure 9-2 Typical debug system

A debug system typically has three parts:

- *The debug host*
- *The protocol converter* on page 9-6
- *ARM946E-S debug target* on page 9-6.

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

9.2.1 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.

9.2.2 The protocol converter

An interface, such as a parallel port, connects the debug host to the ARM946E-S processor development system. The messages broadcast over this connection *must be converted* to the interface signals of the ARM946E-S processor. The protocol converter performs the conversion.

9.2.3 ARM946E-S debug target

The ARM9E-S core within the ARM946E-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.

Figure 9-3 on page 9-7 shows the following major blocks of the ARM9E-S:

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 *Overview of EmbeddedICE-RT* on page 9-26.

TAP controller

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

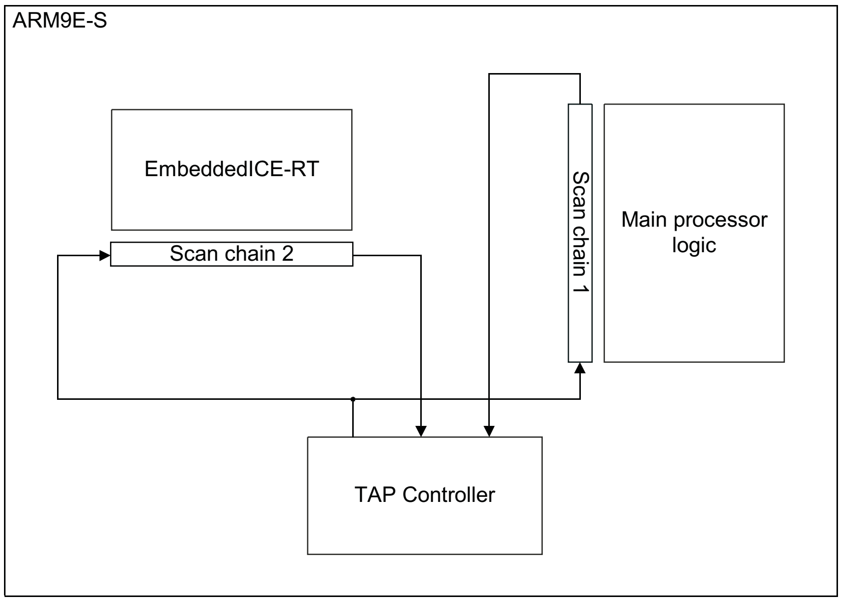


Figure 9-3 ARM9E-S block diagram

The ARM9E-S debug model is extended within the ARM946E-S processor by the addition of scan chain 15. This is used for debug access to the CP15 register bank, to enable you to configure the system state within the ARM946E-S processor while in debug state, for instance to enable or disable the TCM before performing a debug load or store.

9.3 The JTAG state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure 9-4 shows the state transitions that occur in the TAP controller, with the state names and their numbers. State numbers are output from the ARM946E-S processor on **DBGTAPSM[3:0]**.

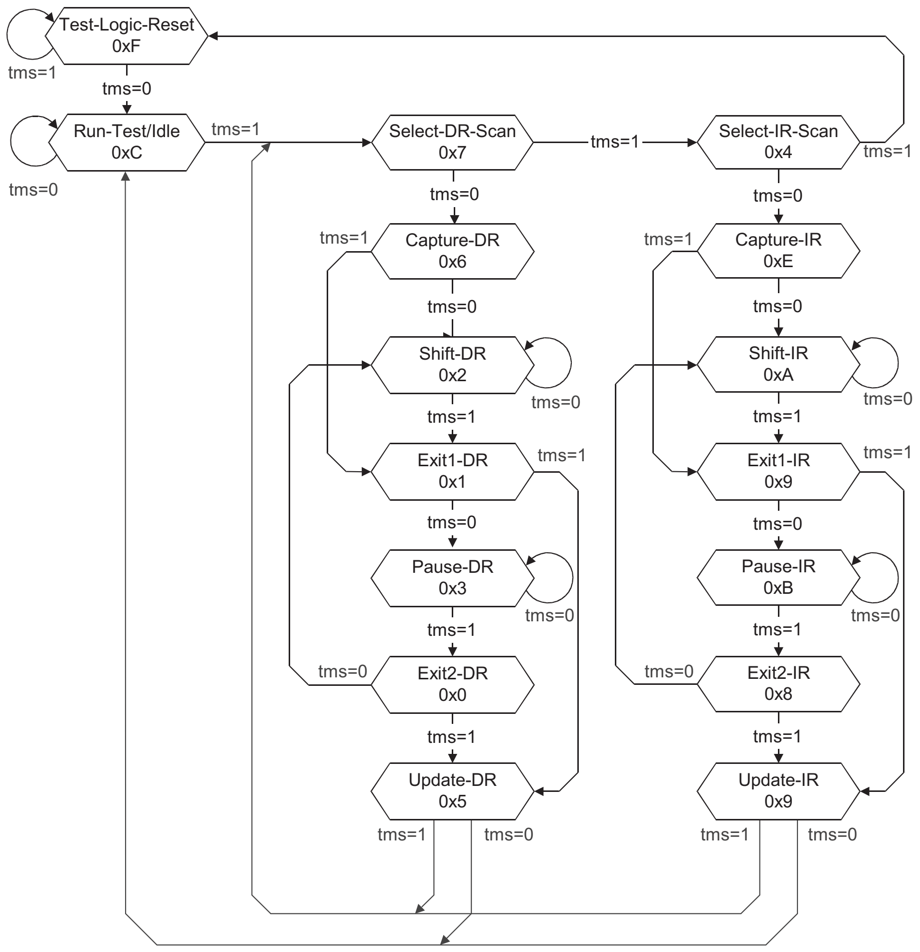


Figure 9-4 TAP controller state transitions[1]

1. From IEEE Std. 1149.1-2001. Copyright 2001 IEEE. All rights reserved.

9.3.1 Reset

The JTAG interface includes a state-machine controller (the TAP controller). To force the TAP controller into the correct state after first applying power to the device you must apply a reset pulse to the **DBGnTRST** signal, or you must cycle the JTAG state machine through the TEST-LOGIC-RESET state. If you do not intend using the JTAG interface, you can tie the **DBGnTRST** input permanently LOW.

Note

A clock on **TCK** is not necessary to reset the device.

The action of reset is as follows:

1. Forces exit from debug state. The boundary scan chain cells do *not* intercept any of the signals passing between the external system and the core.
2. The IDCODE instruction is selected. If the TAP controller is put into the SHIFT-DR state and **TCK** is pulsed, the contents of the ID Register are clocked out of **TDO**.

9.3.2 Instruction Register

The Instruction Register is four bits in length. There is no parity bit. The fixed value loaded into the Instruction Register during the CAPTURE-IR controller state is b0001.

9.3.3 Public instructions

Table 9-1 lists the public instructions that are supported.

Table 9-1 Public instructions

Instruction	Binary code
EXTEST	b0000
SCAN_N	b0010
INTEST	b1100
IDCODE	b1110
BYPASS	b1111
SAMPLE/PRELOAD	b0011
RESTART	b0100

In this section it is assumed that **TDI** and **TMS** are sampled on the rising edge of **TCK** and all output transitions on **TDO** occur as a result of the falling edge of **TCK**.

EXTEST (b0000)

The selected scan chain is placed in test mode by the EXTEST instruction. The EXTEST instruction connects the selected scan chain between **TDI** and **TDO**.

When the Instruction Register is loaded with the EXTEST instruction, all the scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, inputs from the system logic and outputs from the output scan cells to the system are captured by the scan cells.

In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain on **TDO**, while new test data is shifted in on the **TDI** input. This data is applied immediately to the system logic and system pins.

SCAN_N (b0010)

This instruction connects the Scan Path Select Register between **TDI** and **TDO**.

During the CAPTURE-DR state, the fixed value b10000 is loaded into the register.

During the SHIFT-DR state, the ID number of the desired scan path is shifted into the Scan Path Select Register.

In the UPDATE-DR state, the scan register of the selected scan chain is connected between **TDI** and **TDO**, and remains connected until a subsequent SCAN_N instruction is issued. On reset, scan chain 3 is selected by default. The Scan Path Select Register is five bits long in this implementation, although no finite length is specified.

INTEST (b1100)

The selected scan chain is placed in test mode by the INTEST instruction. The INTEST instruction connects the selected scan chain between **TDI** and **TDO**.

When the Instruction Register is loaded with the INTEST instruction, all the scan cells are placed in their test mode of operation.

In the CAPTURE-DR state, the value of the data applied from the core logic to the output scan cells, and the value of the data applied from the system logic to the input scan cells is captured.

In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain on the **TDO** signal pin, while new test data is shifted in on the **TDI** signal pin.

IDCODE (b1110)

The IDCODE instruction connects the Device Identification Register (or ID Register) between **TDI** and **TDO**. The ID Register is a 32-bit register that enables the manufacturer, part number, and version of a component to be determined through the TAP. The ID Register is loaded from the **TAPID[31:0]** input bus. This must be tied to a constant value that represents the unique JTAG IDCODE for the device.

When the Instruction Register is loaded with the IDCODE instruction, all the scan cells are placed in their normal (system) mode of operation.

In the CAPTURE-DR state, the device identification code is captured by the ID Register.

In the SHIFT-DR state, the previously captured device identification code is shifted out of the ID Register on the **TDO** signal pin, while data is shifted in on the **TDI** signal pin into the ID Register.

In the UPDATE-DR state, the ID Register is unaffected.

BYPASS (b1111)

The BYPASS instruction connects a 1-bit shift register (the Bypass Register) between **TDI** and **TDO**.

When the BYPASS instruction is loaded into the Instruction Register, all the scan cells are placed in their normal (system) mode of operation. This instruction has no effect on the system pins.

In the CAPTURE-DR state, a 0 is captured by the bypass register.

In the SHIFT-DR state, test data is shifted into the Bypass Register on **TDI** and out on **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a 0.

The Bypass Register is not affected in the UPDATE-DR state.

Note

All unused instruction codes default to the BYPASS instruction.

SAMPLE/PRELOAD (b0011)

When the Instruction Register is loaded with the SAMPLE/PRELOAD instruction, all the scan cells of the selected scan chain are placed in the normal mode of operation.

In the CAPTURE-DR state, a snapshot of the signals of the boundary scan is taken on the rising edge of **TCK**. Normal system operation is unaffected.

In the SHIFT-DR state, the sampled test data is shifted out of the boundary scan on the **TDO** signal pin, while new data is shifted in on the **TDI** signal pin to preload the boundary scan parallel input latch. This data is not applied to the system logic or system pins while the SAMPLE/PRELOAD instruction is active.

You must use this instruction to preload the Boundary Scan Register with known data prior to selecting INTEST or EXTEST instructions.

RESTART (b0100)

This instruction restarts the processor on exit from debug state. The RESTART instruction connects the Bypass Register between **TDI** and **TDO** and the TAP controller behaves as if the BYPASS instruction is loaded. The processor resynchronizes back to the memory system when the RUN-TEST/IDLE state is entered.

9.4 Scan chains

The ARM946E-S processor supports 32 scan chains. Three scan chains are used inside the ARM946E-S processor. These enable testing, debugging, and programming of the EmbeddedICE-RT watchpoint units.

Table 9-2 shows the supported scan chains.

Table 9-2 ARM946E-S scan chain allocations

Scan chain number	Function
16-31	Unassigned
15	Control coprocessor
4-14	Reserved
3	External boundary scan
2	EmbeddedICE-RT logic programming
1	Debug
0	Reserved

9.4.1 Scan chain 1

This scan chain is primarily used for debugging and provides access to the core instruction and data buses. These are arranged as shown in Table 9-3.

Table 9-3 Scan chain 1 bits

Bits	Function
[66:35]	Data values
[34:32]	Control bits
[31:0]	Instruction values

The three control bits are:

- SYSSPEED
- WPTANDBKPT
- a reserved bit.

While debugging, the value placed in the SYSSPEED control bit determines if the ARM9E-S core executes the instruction at system speed.

After the ARM946E-S processor has entered debug state, the first time SYSSPEED is captured and scanned out tells the debugger whether the core has entered debug state because of a breakpoint (SYSSPEED clear) or a watchpoint (SYSSPEED set). A watchpoint and a breakpoint can occur simultaneously. When a watchpoint condition occurs, the WPTANDBKPT bit must be examined by the debugger to determine whether the instruction currently in the Execute stage of the pipeline is breakpointed. If it is, WPTANDBKPT is set, otherwise it is clear.

9.4.2 Scan chain 2

Scan chain 2 enables access to the EmbeddedICE-RT Logic Registers. Table 9-4 shows the order of the scan chain from the **DBGTDI** input to the **DBGTDO** output.

Table 9-4 Scan chain 2 bits

Bits	Function
[37]	Read = 0, write = 1
[36:32]	Register address
[31:0]	Data value

No action occurs during CAPTURE-DR.

During SHIFT-DR, a data value is shifted into the serial register. Bits 36:32 specify the address of the EmbeddedICE-RT Register to be accessed.

During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read, 1 = write).

9.4.3 Scan chain 15

Scan chain 15 enables debug access to the CP15 register bank and enables the cache to be interrogated. Scan chain 15 is 39 bits long.

Table 9-5 shows the order of scan chain 15 from the **DBGTDI** input to the **DBGTDO** output.

Table 9-5 Scan chain 15 bits

Bits	Contents
[38]	Read = 0, write = 1
[37:32]	CP15 register address
[31:0]	CP15 data value

Table 9-6 shows the mapping of the CP15 Register address field of scan chain 15 to CP15 Registers.

Table 9-6 Mapping of scan chain 15 address field to CP15 registers

Address [37]	[36:33]	[32]	Register number	Register name	Register type
0	b0000	0	C0.ID	ID Register	Read
0	b0000	1	C0.C	Cache type	Read
0	b0001	0	C1	Control	Read/write
0	b0010	0	C2.D	Data cachable bits	Read/write
0	b0010	1	C2.I	Instruction cachable bits	Read/write
0	b0011	0	C3	Write buffer control	Read/write
0	b0100	0	C0.M	Tightly-coupled memory size	Read
0	b0101	0	C5.D	Data space access permissions	Read/write
0	b0101	1	C5.I	Instruction address access permissions	Read/write
1	<CRm>[a]	0	C6.[7:0]	Memory region protection	Read/write
0	b0111	0	C7.FD	Flush data cache	Read/write
0	b0111	1	C7.FI	Flush instruction cache	Read/write
0	b1110	0	C7.FD.s	Flush data cache single (uses C15.C.Ind)	Read/write
0	b1110	1	C7.FI.s	Flush instruction cache single (uses C15.C.Ind)	Read/write
1	b1010	1	C7.CD.s	Clean data cache single (uses C15.C.Ind)	Read/write

Table 9-6 Mapping of scan chain 15 address field to CP15 registers (continued)

Address [37]	[36:33]	[32]	Register number	Register name	Register type
0	b1001	0	C9.D	Data cache lock-down	Read/write
0	b1001	1	C9.I	Instruction cache lock-down	Read/write
1	b1000	1	C9.Dram	Data TCM size/location	Read/write
1	b1001	1	C9.Iram	Instruction TCM size/location	Read/write
0	b1101	1	C13.TPID	Trace process identifier	Read/write
0	b1111	0	C15.State	Test state	Read/write
0	b1111	1	C15.TAG	TAG BIST control	Read/write
1	b1111	1	C15.RAM	Cache RAM BIST control	Read/write
1	b1101	0	C15.C.Ind	Cache index (address/segment)	Read/write
0	b1010	0	C15.DC	Data cache read/write (uses C15.C.Ind)	Read/write
0	b1010	1	C15.IC	Instruction cache read/write (uses C15.C.Ind)	Read/write
0	b1011	0	C15.DT	Data tag read/write (uses C15.C.Ind)	Read/write
0	b1011	1	C15.IT	Instruction tag read/write (uses C15.C.Ind)	Read/write
1	b1110	1	C15.Mem	TCM BIST control	Read/write

a. For CP15 register 6, CRm corresponds to the region number (b0000 to b0111).

In the SHIFT-DR state of the TAP state machine, the read/write bit, the register address and the register value for writing, are shifted in.

For a write, the register value is updated when the UPDATE-DR state is reached.

For reading, return to SHIFT-DR through CAPTURE-DR to shift out the register value.

9.4.4 Scan Chain Debug Status Register

In situations where the AHB clock frequency is significantly less than the debugger clock frequency, cache maintenance operations initialized by the debug scan chain (scan chain 15) might not be registered by the ARM946E-S processor.

This situation can be prevented by providing status information to the debugger. Cache maintenance operations (cache flush and cache clean) are read/write accesses. By reading back from the same scan chain register address that initiated the maintenance operation, a status bit is returned to the debugger. If the bit is set, the operation has been completed and the debug sequence can continue. If the bit is cleared, the requested operation has not been completed.

The status bit is implemented for the debug scan chain operations shown in Table 9-7.

Table 9-7 Status bit mapping of scan chain 15 address field to CP15 registers

Address [37]	[36:33]	[32]	Register number	Register name	Register type
0	b0111	0	C7.FD	Flush data cache	Read/write
0	b0111	1	C7.FI	Flush instruction cache	Read/write
0	b1110	0	C7.FD.s	Flush data cache single (uses C15.C.Ind)	Read/write
0	b1110	1	C7.FI.s	Flush instruction cache single (uses C15.C.Ind)	Read/write
1	b1010	1	C7.CD.s	Clean data cache single (uses C15.C.Ind)	Read/write
0	b1011	1	C15.IT	Instruction tag read/write (uses C15.C.Ind)	Read/write
1	b1110	1	C15.Mem	TCM BIST control	Read/write

Table 9-7 shows the complete list of operations that can be initiated from the debug scan chain.

Table 9-8 shows the status bit associated with each cache maintenance operation.

Table 9-8 Correlation between status bits and cache operations

Status bits	Cache maintenance operation
[31:19]	Unpredictable
[18]	Flush instruction cache busy
[17]	Flush instruction cache single busy
[16:11]	Unpredictable
[10]	Flush data cache busy
[9]	Flush data cache single busy
[8]	Unpredictable
[7]	Clean data cache single busy
[6:0]	Unpredictable

9.5 Debug access to the caches

It is desirable for the debugger to examine the contents of the instruction and data caches during debug operations. This is achieved in two steps:

1. The debugger determines if valid addresses are stored in the cache and forms TAG addresses from the TAG contents and the TAG index.
2. The debugger uses the generated addresses to either access main memory, or to read individual entries using the CP15 scan chain.

9.5.1 Debug access to the caches, Step 1

This is done by reading the instruction cache and data cache TAG arrays using scan chain 15. The debugger must do this for each entry set within the cache. Figure 9-5 shows the format of the data returned.

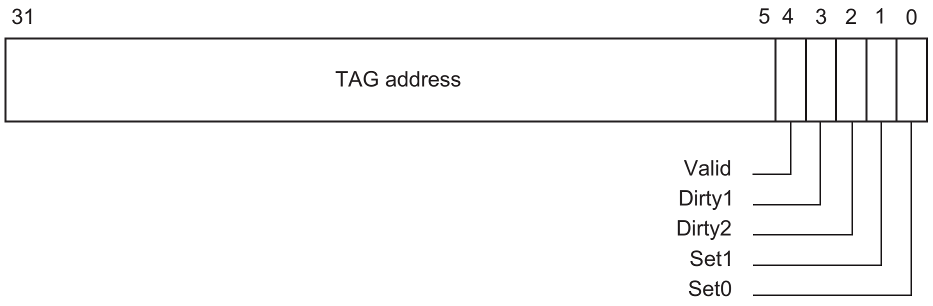


Figure 9-5 TAG address format

The TAG address is formed from the TAG contents and the TAG index used to interrogate the TAG. This ensures that the data format returned is consistent regardless of cache size.

9.5.2 Debug access to the caches, Step 2

Reading individual entries using the CP15 scan chain can be useful where an entry has been marked as dirty, because this indicates that there is an inconsistency between the cache contents and main memory.

For the data cache, the debugger can execute system speed accesses that hit in the cache and, therefore, return the cache contents. Writes to the data cache from the processor core by this method result in the dirty bits being set for write-back regions, and main memory is updated for write-through regions.

If the CP15 scan chain is used for updating the data cache, only the cache contents are updated. Writes are not made to main memory. For this method you must first program the Index/Set Register with the required cache index, set, and word values. Figure 9-6 shows the format of the Cache Index Register.

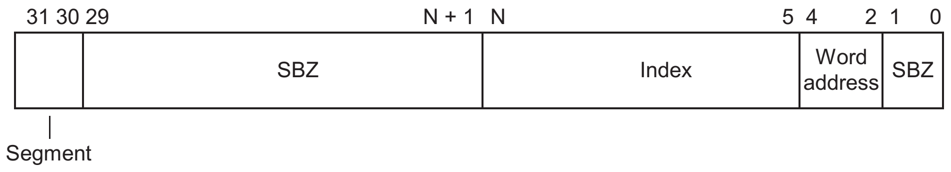


Figure 9-6 Cache Index Register format

Note

Although 27 bits are specified for the TAG address, only those bits required for the TAG implemented are used.

The Cache Index Register is also used for writing to the instruction cache. This is useful for setting software breakpoints within code already in the cache. This means that you do not have to flush the cache and reload the entry.

Note

There is no mechanism for detecting that the instruction cache has been updated in this way. The debugger must restore the original cache contents after executing the breakpoint.

9.6 Debug interface signals

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

- **DBGIEBKPT**, **DBGDEWPT**, and **EDBGRQ** are system requests for the ARM946E-S processor to enter debug state
- **DBGACK** flag is used by the ARM946E-S processor to inform the system that it is in debug state.

9.6.1 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, you must assert the breakpoint signal by the end of the same cycle, as shown in Figure 9-7.

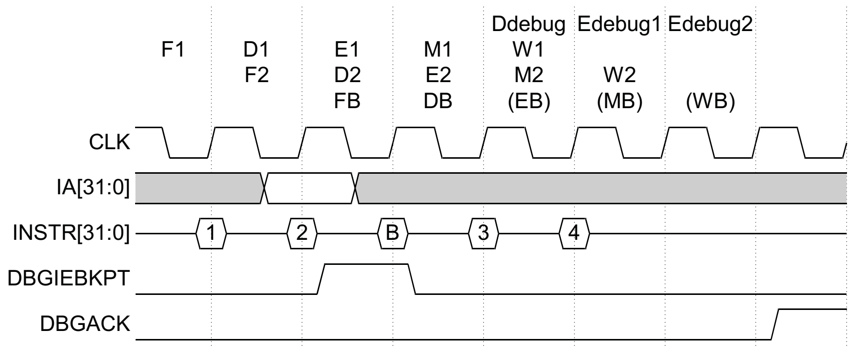


Figure 9-7 Breakpoint timing

You can build external logic, such as additional breakpoint comparators, to extend the breakpoint functionality of the EmbeddedICE-RT logic. The output from the external logic must be applied to the **DBGIEBKPT** input. This signal is logically 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 enabled 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.

9.6.2 Breakpoints and exceptions

A breakpointed instruction can have a Prefetch Abort associated with it. If so, the Prefetch Abort takes priority and the breakpoint is ignored. This is because, if there is a Prefetch Abort, instruction data might be invalid, the breakpoint might have been data-dependent, and as the data might be incorrect, the breakpoint might have been triggered incorrectly.

`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.

On an instruction boundary, if there is a breakpointed instruction and an interrupt (**nIRQ** or **nFIQ**), the interrupt is taken and the breakpointed instruction is discarded. When the interrupt has been serviced, the execution flow is returned to the original program. Where the previously breakpointed instruction 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 has entered halt mode debug state, it is important that additional interrupts do not affect the instructions executed. For this reason, as soon as the processor enters halt mode, interrupts are disabled, although the state of the I and F bits in the *Program Status Register (PSR)* are not affected

9.6.3 Watchpoints

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

You can build external logic, such as external watchpoint comparators, to extend the functionality of the EmbeddedICE-RT logic. The output of the external logic must be applied to the **DBGDEWPT** input. This signal is logically 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 can be implemented.

After a watchpointed access, the next instruction in the processor pipeline is always enabled to complete execution. Where 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 9-8 on page 9-22.

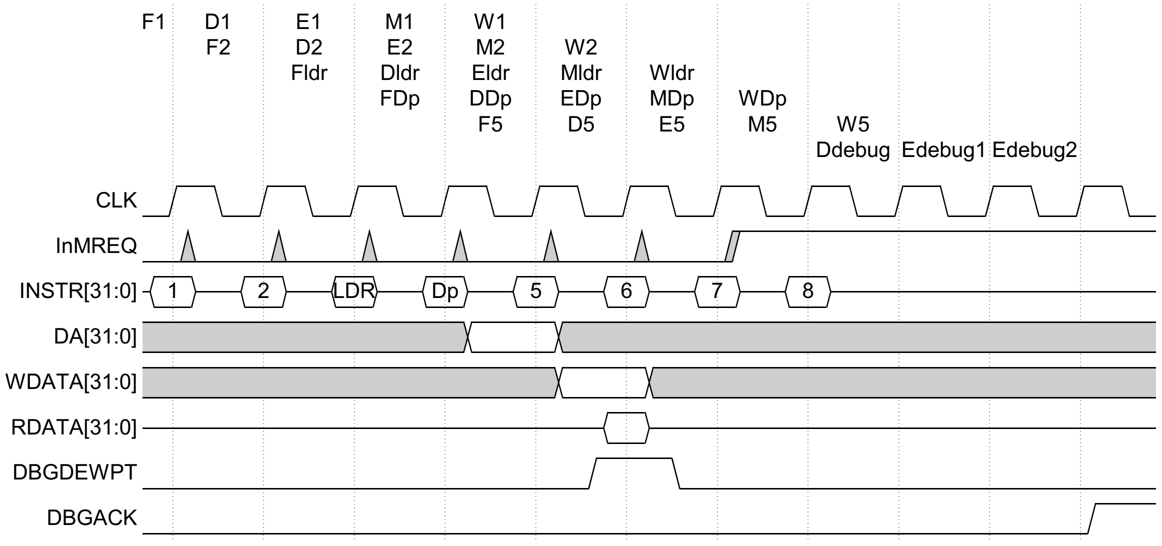


Figure 9-8 Watchpoint entry with data processing instruction

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 that has not yet been executed.

The instruction following the instruction that generated the watchpoint might modify the *Program Counter (PC)*. If this happens, you cannot determine the instruction that caused the watchpoint. Figure 9-9 on page 9-23 shows the timing for debug entry after a watchpoint, where the next instruction is a branch.

When the processor has entered debug state, you can interrogate the ARM9E-S core 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.

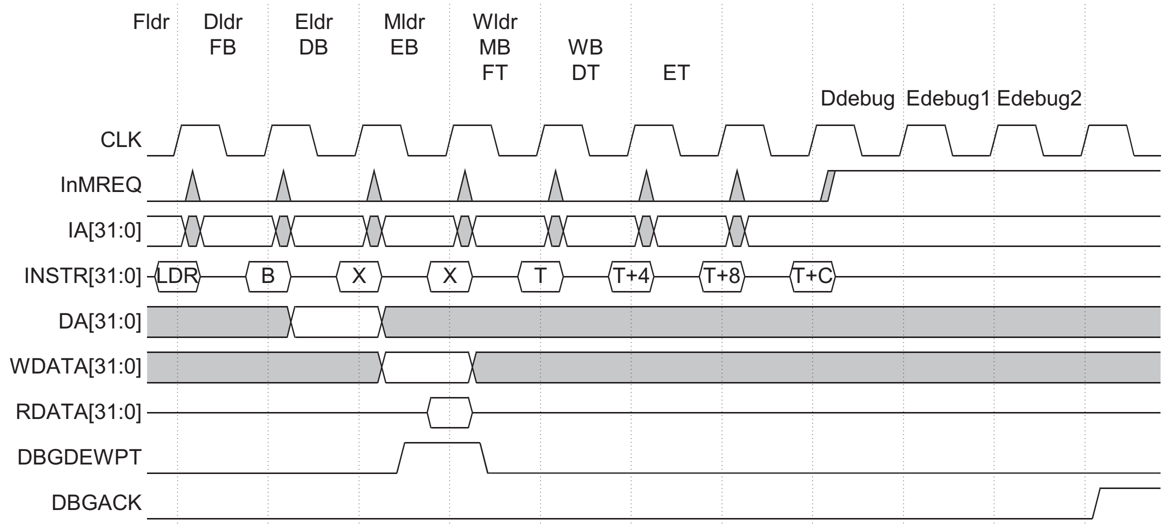


Figure 9-9 Watchpoint entry with branch

9.6.4 Watchpoints and exceptions

If a watchpointed data access also causes an abort, the watchpoint condition is registered and the exception entry sequence performed, and then the processor enters debug state.

If there is an interrupt pending, the ARM9E-S core enables the exception entry sequence to occur and then enters debug state.

9.6.5 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 execution stage of the pipeline has completely finished executing (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.

Note

If **EDBGRQ** is asserted while the processor is operating in monitor mode, the processor enters debug state as if operating in halt mode.

9.6.6 Actions of the ARM9E-S in debug state

When the ARM9E-S core is in debug state, both memory interfaces indicate internal cycles. This ensures that the tightly-coupled memory within the ARM946E-S processor, and the AHB interface, are both at a steady state, enabling 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. This is because if the system applies a reset (**HRESETn** is driven LOW) to the ARM946E-S processor, the state of the ARM9E-S core changes without the knowledge of the debugger.

9.7 Determining the core and system state

When the ARM946E-S processor is in debug state, you can examine the core and system state by forcing load or store multiple instructions into the instruction pipeline.

Before you can examine the core and system state, the debugger must determine whether the processor entered debug from Thumb state or ARM state, by examining bit 4 of the EmbeddedICE-RT Debug Status Register. When bit 4 is set, the core has entered debug from Thumb state.

9.8 Overview of EmbeddedICE-RT

The ARM9E-S EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM9E-S core within the ARM946E-S processor.

EmbeddedICE-RT is programmed serially using the ARM9E-S TAP controller. Figure 9-10 shows the relationship between the core, EmbeddedICE-RT, and the TAP controller, showing only the signals that are pertinent to EmbeddedICE-RT.

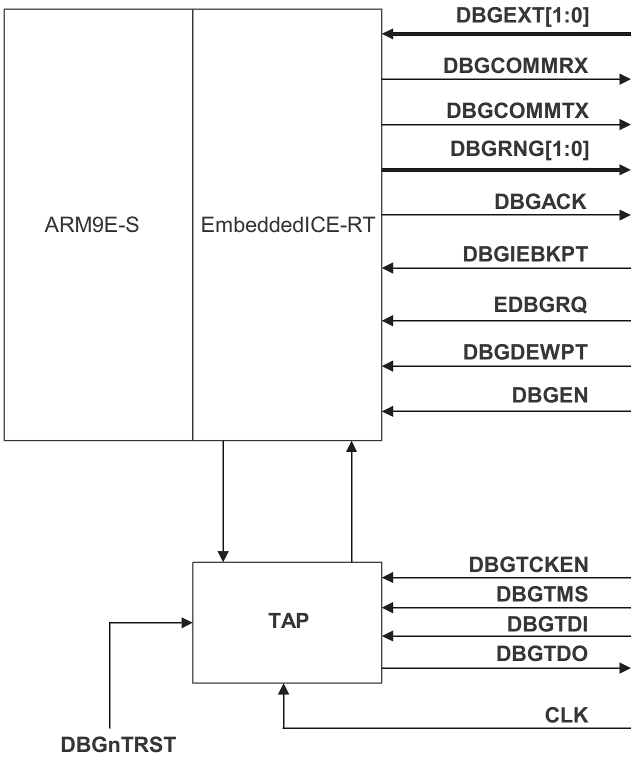


Figure 9-10 The ARM9E-S, TAP controller, and EmbeddedICE-RT

The EmbeddedICE-RT logic comprises:

- two real-time watchpoint units
- two independent registers:
  - the Debug Control Register
  - the Debug Status Register
- debug communication 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

You can mask bits so that their values do not affect the comparison.

You can configure each watchpoint unit to be either a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). Watchpoints and breakpoints can be data-dependent in halt mode debug.

9.9 Disabling EmbeddedICE-RT

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

——— **Caution** ————

Hard wiring the **DBGEN** input LOW *permanently* disables debug access.

————————————

When **DBGEN** is LOW, it inhibits **DBGDEWPT**, **DBGIEBKPT**, and **EDBGRQ** to the core, and **DBGACK** from the ARM946E-S processor is always LOW.

9.10 The debug communication channel

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

The communication channel comprises:

- a 32-bit Communication Data Read Register
- a 32-bit Wide Communication Data Write Register
- a 6-bit wide Communication Channel Status 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 communication channel registers, the processor can access one bit of the 32-bit debug status register for use in the real-time debug configuration.

9.10.1 Debug Communication Channel Registers

CP14 contains four registers. These have the register allocations shown in Table 9-9.

Table 9-9 Coprocessor 14 register map

Register name	Register number	Notes
Communication channel status	C0	Read-only
Communication channel data read	C1	For reads
Communication channel data write	C1	For writes
Debug status	C2	Read/write

9.10.2 Debug Communication Channel Status Register

The Debug Communication Channel Status Register is read-only. It controls synchronized handshaking between the processor and the debugger. Figure 9-11 on page 9-30 shows the Debug Communication Channel Status Register.

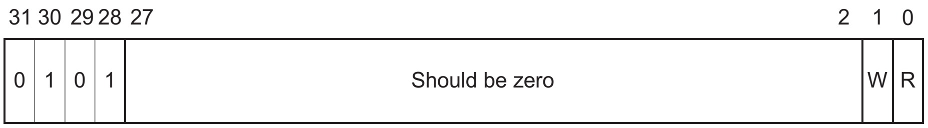


Figure 9-11 Debug Communication Channel Status Register

Each register bit functions as follows:

Bits [31:28]	Contain a fixed pattern that denotes the EmbeddedICE-RT version number (in this case b0101).
Bits [27:2]	Are reserved.
Bit 1	Denotes whether the communication channel data write register is available (from the point of view of the processor). If, from the point of view of the processor, the communication channel data write register is free (W is 0), new data can be written. If the register is not free (W is 1), the processor must poll until W is 0. From the point of view of the debugger, when W is 1, some new data has been written that can then be scanned out.
Bit 0	Denotes whether there is new data in the communication channel data read register. If R is 1, there is new data that can be read using an `MRC` instruction. From the point of view of the debugger, if R is 0, the communication channel data read register is free, and new data can be placed there through the scan chain. If R is 1, this denotes that data previously placed there through the scan chain has not been collected by the processor, and so the debugger must wait.

From the point of view of the debugger, the registers are accessed using the scan chain in the usual way. From the point of view of the processor, these registers are accessed using coprocessor register transfer instructions as follows:

```
MRC p14, 0, Rd, c0, c0
```

This returns the debug communication channel status register into `Rd`.

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

This writes the value in Rn to the communication channel data write register.

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

This returns the debug data read register into `Rd`.

You are advised to access this data using `SWI` instructions when in Thumb state because the Thumb instruction set does not contain coprocessor instructions.

9.10.3 Communications using the communication channel

You can send and receive messages using the communication channel.

Sending a message to the debugger

When the processor has to send a message to the debugger, it must check the communication channel data write register is free for use by reading the W bit of the Debug Communication Channel Status Register:

- If the W bit is clear, the Communication Channel Data Write Register is clear.
- If the W bit is set, previously written data has not been 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 can be sent by a register transfer to coprocessor 14. While the data transfer occurs from the processor to the Communication Channel Data Write Register, the W bit is set in the Debug Communication Channel Status Register.

The debugger sees both the R and W bits when it polls the Debug Communication Channel Status Register through the JTAG interface. When the debugger sees that the W bit is set, it can read the Communication Channel Data Write Register, and scan the data out. The action of reading this data register clears the Debug Communication Channel Status 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 Communication Channel Status Register:

- if the R bit is clear, the Communication 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 has not yet been collected, so the debugger must wait.

When the Communication Channel Data Read Register is free, data can be written to it using the JTAG interface. This sets the R bit in the Debug Communication Channel Status Register.

The processor polls the Debug Communication Channel Status Register. If the R bit is set, there is data that can be read using an `MRC` instruction to coprocessor 14. Reading the Communication Channel Data Register clears the R bit in the Debug Communication Channel Status Register. When the debugger polls this register and sees that the R bit is clear, the data has been taken, and the process can now be repeated.

9.11 Monitor mode debugging

The ARM9E-S within ARM946E-S processor contains logic that enables you to debug 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. When this bit is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the ARM to enter abort mode, taking the Prefetch Abort or Data Abort vectors respectively. You must be aware of a number of restrictions when the ARM is configured for monitor mode debugging:

- Breakpoints/watchpoints cannot be data-dependent. No support is provided for the range and chain functionality. Breakpoints/watchpoints can only be based on:
  - instruction/data addresses
  - external watchpoint conditioner (**DBGEXT**)
  - User/Privileged mode access
  - read/write access (watchpoints)
  - access size (breakpoints).
- The single-step hardware is not enabled.
- External breakpoints/watchpoints are not supported.
- You can use the vector catching hardware, but must not configure it to catch the Prefetch or Data Abort exceptions.
- If **EDBGRQ** is asserted while the processor is operating in monitor mode, the processor enters debug state as if operating in halt mode.

The fact that an abort has been generated by the monitor mode is recorded in the Abort Status Register in coprocessor 14 (see *Scan Chain Debug Status Register* on page 9-16).

The monitor mode enable bit does not put the ARM946E-S processor into debug state. For this reason, it is necessary to change the contents of the watchpoint registers while external memory accesses are taking place, rather than changing them when in debug state where the core is halted.

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) you must:

1. Disable the watchpoint unit by setting EmbeddedICE-RT disable, bit 5 in the Debug Control Register.
2. Change the other registers.
3. Re-enable the watchpoint unit by clearing the EmbeddedICE-RT disable bit in the Debug Control Register.

9.11.1 Debug in depth

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

Chapter 10
Test Support

This chapter describes the test methodology used for the ARM946E-S processor synthesized logic and memory. It contains the following sections:

10.1 About the ARM946E-S processor test methodology

To achieve a high level of fault coverage, you can use scan insertion and ATPG techniques on the ARM9E-S core and ARM946E-S processor control logic as part of the synthesis flow. You can use *Built-In Self Test* (BIST) to provide high fault coverage of the compiled RAMs (cache and TCM).

10.2 Scan insertion and ATPG

This technique is covered in detail in the *ARM946E-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.

You can use *Automatic Test Pattern Generation* (ATPG) tools to create the necessary scan patterns to test the logic, after the scan insertion has been performed. With this technique you can achieve very high fault coverage for the standard cell combinatorial logic, typically in the 95-99% range.

Scan insertion does have an impact on the area and performance of the synthesized design, because of the larger scan register elements and the serial routing between them. However, to minimize these effects, the scan insertion is performed early in the synthesis cycle and the design re-optimized with the scan elements in place.

10.2.1 ARM946E-S INTEST wrapper

In addition to the auto-inserted scan chains, the ARM946E-S processor optionally includes a dual-purpose INTEST scan chain wrapper. This facilitates ATPG and provides an additional method for activating BIST of the compiled RAM.

ATPG

You can use the INTEST scan chain to enable an ATPG tool to access the ARM946E-S processor top-level inputs and outputs in an embedded design. This wrapper adds a scan source for each ARM946E-S processor input and a capture cell for each output. The ATPG tools use this scan chain in addition to the ones created by scan insertion, to test the logic from a given input pin to any register that it connects to, and from any registers whose outputs end up at a pin.

Note

The order of this scan chain is predetermined and must be maintained through synthesis and place and route of the macrocell.

BIST activation

To enable the BIST hardware to be activated by scan means, the INTEST wrapper has a second operational mode. When the **SERIALEN** input is true, serialized `MCR` instructions to initiate BIST operation are scanned in through this scan chain. The instructions target the CP15 BIST Register. After a predetermined number of clock cycles (depending on the size of the test), the appropriate `MRC` instruction is scanned in to read the BIST Control Register to check the test result. The INTEST wrapper enables the full range of BIST operations to be applied as detailed in *BIST of memory arrays* on page 10-5. The flow for generating the serialized patterns from ARM assembler source is supplied with the ARM946E-S implementation scripts.

TESTMODE

This signal is used to prevent the cache from being inadvertently flushed when scan patterns are shifted through the scan chains. It must only be asserted during scan test of the ARM946E-S processor.

10.3 BIST of memory arrays

Caution

Code for running the BIST must not be placed in the Instruction TCM or in a cacheable location, because this can cause invalid or dirty data to be introduced into program execution. Also, caches must be flushed after running the BIST.

Adding a simple memory test controller enables you to perform an exhaustive test of the memory arrays. You can activate BIST operation using an `MCR` to the CP15 BIST Control Register.

When you perform a BIST operation on compiled RAM, the functional enable for all RAMs is automatically disabled, forcing all memory accesses to all TCM and cache address ranges to go to the AHB. This enables you to run BIST operations in the background (for instance the Instruction TCM can be have BIST applied, while code is executed over the AHB).

Serial scan access to the CP15 BIST operations is also provided for production test purposes, using a special mode of operation of the INTEST wrapper. See *ARM946E-S INTEST wrapper* on page 10-3.

You can also perform limited BIST in debug state by using scan chain 15 to access the CP15 BIST Control Register. This is not necessarily recommended as the BIST operation corrupts the contents of the TCM being tested.

You can achieve full programmer control over the BIST mechanism through five registers that are mapped to CP15 register 15 address space. For details of the `MCR/MRC` instructions used to access these registers, see *Register 15, BIST Control Registers* on page 2-29.

10.3.1 BIST algorithm

The BIST test algorithm is a 6N test. Figure 10-1 on page 10-7 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. 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. The inverse pattern is checked, a fixed value is written into each memory address to be tested. The pattern is then checked. If the pattern match fails on either check, 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.

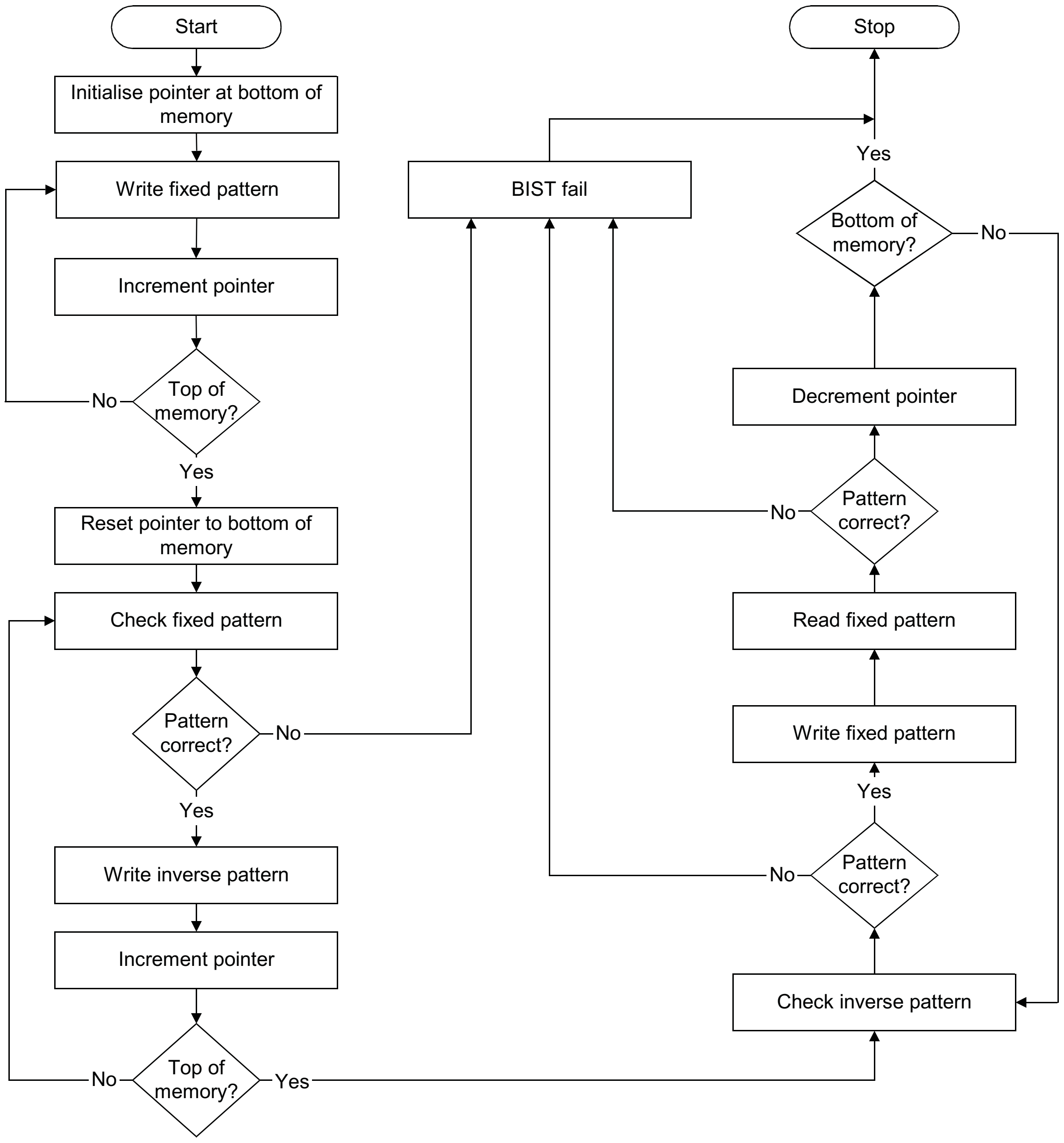


Figure 10-1 Test flow for BIST

10.3.2 BIST Control Register

The CP15 register 15 *BIST Control Register* controls the operation of the compiled RAM memory BIST. Before initiating a BIST operation, an `MCR` is first performed to the BIST Control Register to set up the size of the test and enable the RAM to be tested. An additional `MCR` is required to initiate the test.

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

- running
- paused
- failed
- completed.

When the test result has been read you can return the memory to functional operation. You must first clear the BIST enable by writing to the BIST Control Register. You must then re-enable the memory array by writing to CP15 register 1.

Note

Clearing the functional memory array enable when BIST is enabled prevents you from trying to run from cache or TCM following a BIST operation, without having first flushed the cache memory and reprogrammed the RAM. This is necessary because the BIST algorithm corrupts all tested memory locations.

10.3.3 BIST Address and General Registers

The BIST Control Register enables you to perform standard BIST operations on each RAM block and to optionally specify the size of the test. Additional registers are required, however, to provide the following functionality:

- testing of the BIST hardware
- changing the seed data for BIST
- providing a nonzero starting address for BIST
- peek and poke of the RAM
- returning an address location for a failed BIST.

This additional functionality is most useful for debugging faulty silicon during production test. The exception to this is the start address for a BIST operation. It is possible to perform periodic BIST operations on RAM during the execution of a program rather than in one go. This requires a start address that is incremented by the size of the test each time a test is activated.

Note

It is recommended that you do not write application code that relies on the presence of the BIST Address and General Registers. ARM Limited. does not guarantee to support these registers in future versions of the ARM946E-S processor.

Table 10-1 and Table 10-2 show how the registers are used. The pause bits from the BIST Control Register provide extra decode of these registers.

Table 10-1 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 seed data
IBIST General Register	1	IBIST peek data	IBIST poke data

Table 10-2 Data BIST Address and General Registers

BIST Register	DBIST 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 seed data
DBIST General Register	1	DBIST peek data	DBIST poke data

10.3.4 Pause modes

It is recommended that you use the following production test sequence for the compiled RAM:

1. Test each RAM using a full test.
2. Test the BIST hardware for each RAM.

To enable testing of the BIST hardware, it is necessary to deliberately corrupt data in the RAM. This can be done by the ATPG tool if it recognizes the RAM parameters. Alternatively a pause mechanism enables you to halt the BIST test, enabling you to corrupt data within the RAM. The sequence for this is:

1. Use an `MCR` instruction to write the address for the location to be corrupted to the relevant BIST Address Register.
2. Use an `MCR` instruction to write the corrupted data to the BIST General Register.

You can restart the test using an `MCR` instruction to the BIST Control Register and then check to see that the corrupted data causes the test to fail. You can read the address at which the BIST operation failed 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 described in *Auto pause*.

Note

It is recommended that you do not write application code that relies on the presence of the BIST pause mode. ARM Limited does not guarantee to support this feature in future versions of the ARM946E-S processor.

Auto pause

If you set the pause bit in the BIST control register before you activate the test, the test runs in auto pause mode, where the BIST operation pauses at a predetermined point in the BIST algorithm. The test pauses after the first pass through RAM.

You can poll the BIST Control Register to detect when a test has paused (the running flag is clear). You can then corrupt the data, as described in *Pause modes* on page 10-9, before you restart the BIST test.

Note

Auto pause only operates after the first pass of the BIST operation.

10.3.5 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 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 flag is set, and the fail flag set. If the test completes without failures, the BIST running flag is cleared and the completion flag is set. If the test is paused using auto pause, 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.

Appendix A
AC Parameters

This appendix lists the AC timing parameters for the ARM946E-S processor. It contains the following section:

- *Timing diagrams* on page A-2.

A.1 Timing diagrams

The timing diagrams in this section are:

- *Clock, reset, and AHB enable timing*
- *AHB bus request and grant related timing* on page A-3
- *AHB bus master timing* on page A-4
- *Coprocessor interface timing* on page A-5
- *Debug interface timing* on page A-7
- *JTAG interface timing* on page A-9
- *DBGSDOUT to DBGTDO timing* on page A-10
- *Exception and configuration timing* on page A-11
- *TCM interface timing* on page A-12
- *ETM interface timing* on page A-13.

Each timing diagram is followed by a table showing timing parameters. All figures are expressed as percentages of the **CLK** period at maximum operating frequency.

Note

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 therefore require a positive hold relative to the top-level clock input. The amount of hold required is equivalent to the internal clock skew.

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

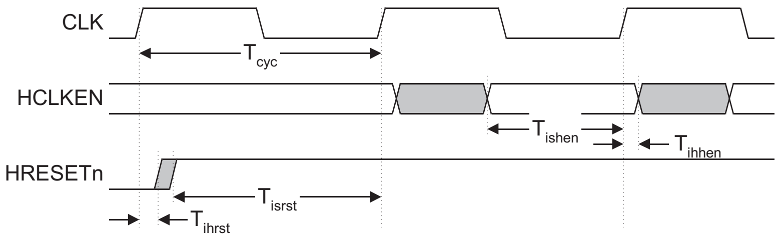


Figure A-1 Clock, reset, and AHB enable timing

Table A-1 shows the timing parameter definitions for clock, reset, and AHB enable.

Table A-1 Timing parameter definitions for clock, reset, and AHB enable

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

Figure A-2 shows the AHB bus request and grant related timing parameters.

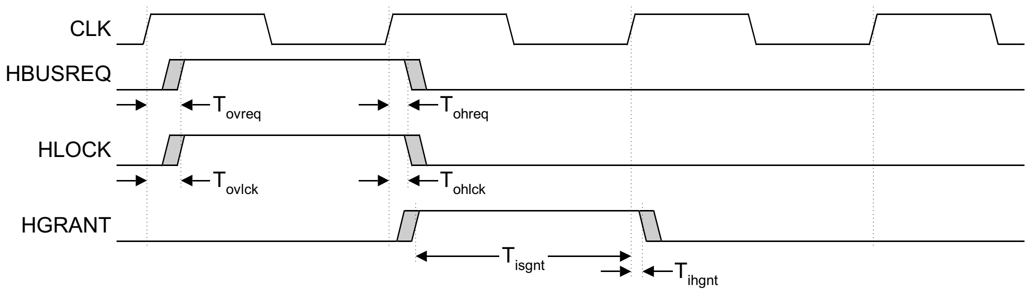


Figure A-2 AHB bus request and grant related timing

Table A-2 shows the parameter definitions for AHB bus request and grant timing.

Table A-2 Parameter definitions for AHB bus request and grant timing

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%
T_{ohlck}	**HLOCK** hold time from rising **CLK**	0%	-
T_{isgnt}	**HGRANT** input setup to rising **CLK**	50%	-
T_{ihgnt}	**HGRANT** input hold from rising **CLK**	0%	-

Figure A-3 shows the AHB bus master timing parameters.

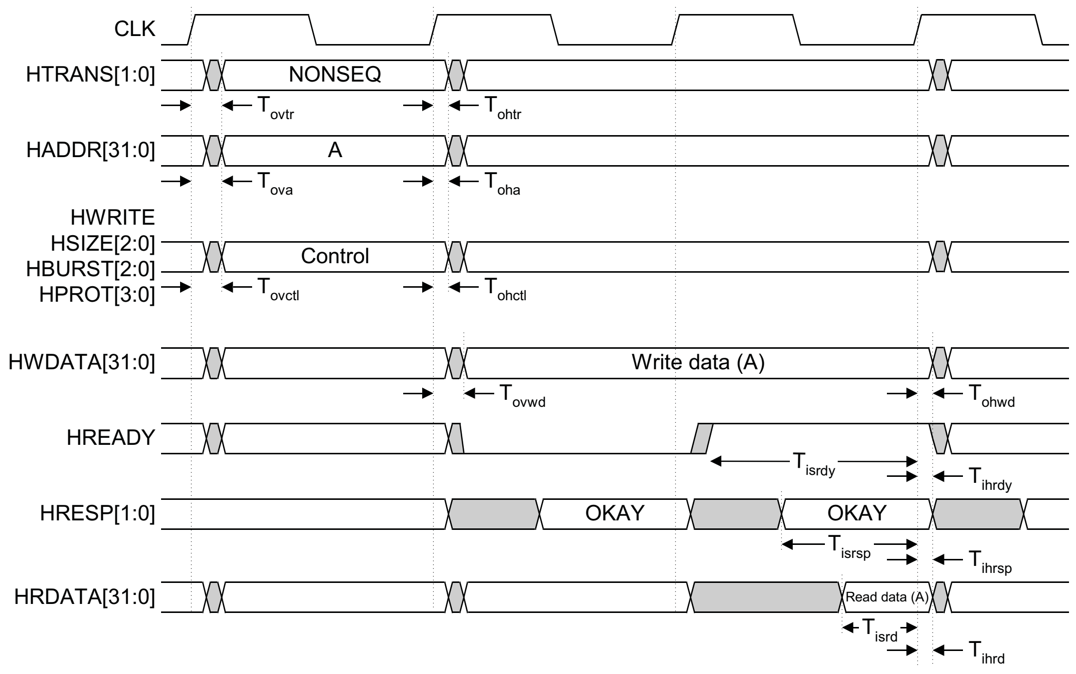


Figure A-3 AHB bus master timing

Table A-3 shows the parameter definitions for AHB bus master timing.

Table A-3 Parameter definitions for AHB bus master timing

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%

Table A-3 Parameter definitions for AHB bus master timing (continued)

Symbol	Parameter	Min	Max
T_{ohwd}	**HWDATA[31:0]** hold time from rising **CLK**	0%	-
T_{isrdy}	**HREADY** input setup to rising **CLK**	50%	-
T_{ihrdy}	**HREADY** input hold from rising **CLK**	0%	-
T_{isrsp}	**HRESP[1:0]** input setup to rising **CLK**	50%	-
T_{ihrsp}	**HRESP[1:0]** input hold from rising **CLK**	0%	-
T_{isrd}	**HRDATA[31:0]** input setup to rising **CLK**	40%	-
T_{ihrd}	**HRDATA[31:0]** input hold from rising **CLK**	0%	-

Figure A-4 shows the coprocessor interface timing parameters.

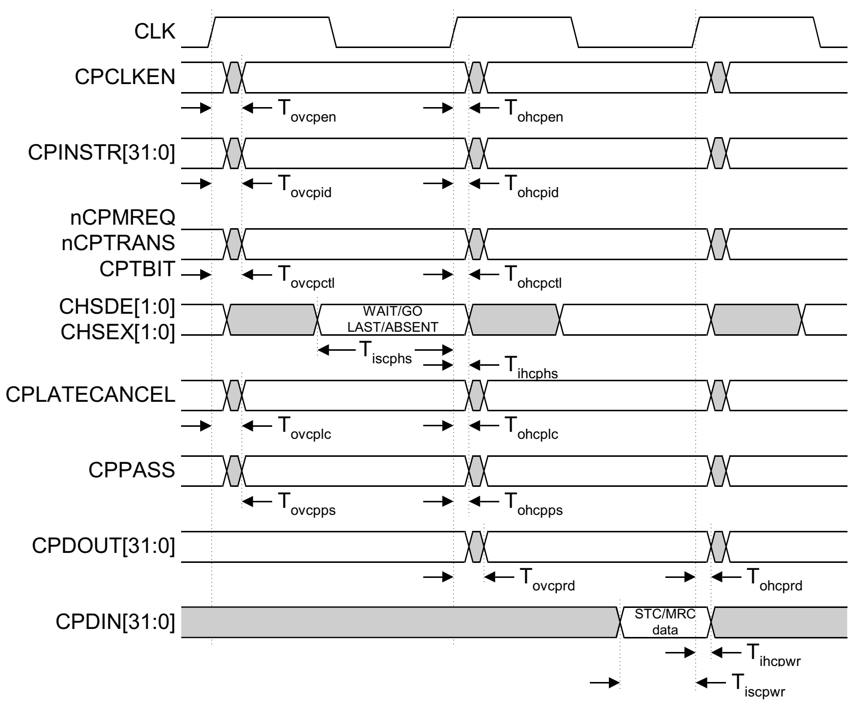


Figure A-4 Coprocessor interface timing

Table A-4 shows the parameter definitions for coprocessor interface timing.

Table A-4 Parameter definitions for coprocessor interface timing

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 to rising **CLK**	50%	-
T_{ihcphs}	Coprocessor handshake input hold 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 to rising **CLK**	50%	-
T_{ihcpwr}	**CPDIN[31:0]** input hold from rising **CLK**	0%	-

Figure A-5 on page A-7 shows the debug interface timing parameters.

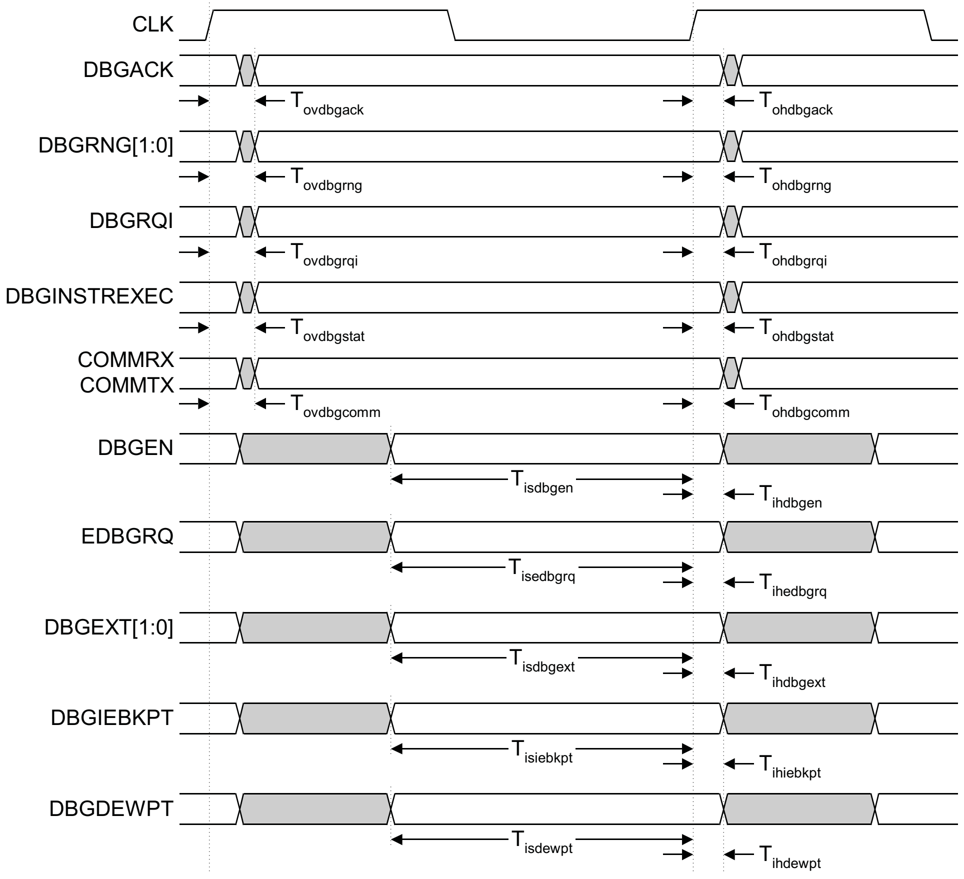


Figure A-5 Debug interface timing

Table A-5 shows the parameter definitions for debug interface timing.

Table A-5 Parameter definitions for debug interface timing

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%	-

Table A-5 Parameter definitions for debug interface timing (continued)

Symbol	Parameter	Min	Max
$T_{ovdbgrqi}$	Rising **CLK** to **DBGRQI** valid	-	45%
$T_{ohdbgrqi}$	**DBGRQI** hold time from rising **CLK**	0%	-
$T_{ovdbgstat}$	Rising **CLK** to **DBGINSTREXEC** valid	-	30%
$T_{ohdbgstat}$	**CLK** hold time from rising **DBGINSTREXEC**	0%	-
$T_{ovdbgcomm}$	Rising **CLK** to communication channel outputs valid	-	60%
$T_{ohdbgcomm}$	Communication channel outputs hold time from rising **CLK**	0%	-
$T_{isdbgen}$	**DBGEN** input setup to rising **CLK**	35%	-
$T_{ihdbgen}$	**DBGEN** input hold from rising **CLK**	0%	-
$T_{isedbgrq}$	**EDBRQ** input setup to rising **CLK**	20%	-
$T_{ihedbgrq}$	**EDBRQ** input hold from rising **CLK**	0%	-
$T_{isdbgext}$	**DBGEXT** input setup to rising **CLK**	15%	-
$T_{ihdbgext}$	**DBGEXT** input hold from rising **CLK**	0%	-
$T_{isiebkpt}$	**DBGIEBKPT** input setup to rising **CLK**	50%	-
$T_{ihiebkpt}$	**DBGIEBKPT** input hold from rising **CLK**	0%	-
$T_{isdewpt}$	**DBGDEWPT** input setup to rising **CLK**	50%	-
$T_{ihdewpt}$	**DBGDEWPT** input hold from rising **CLK**	0%	-

Figure A-6 on page A-9 shows the JTAG interface timing parameters.

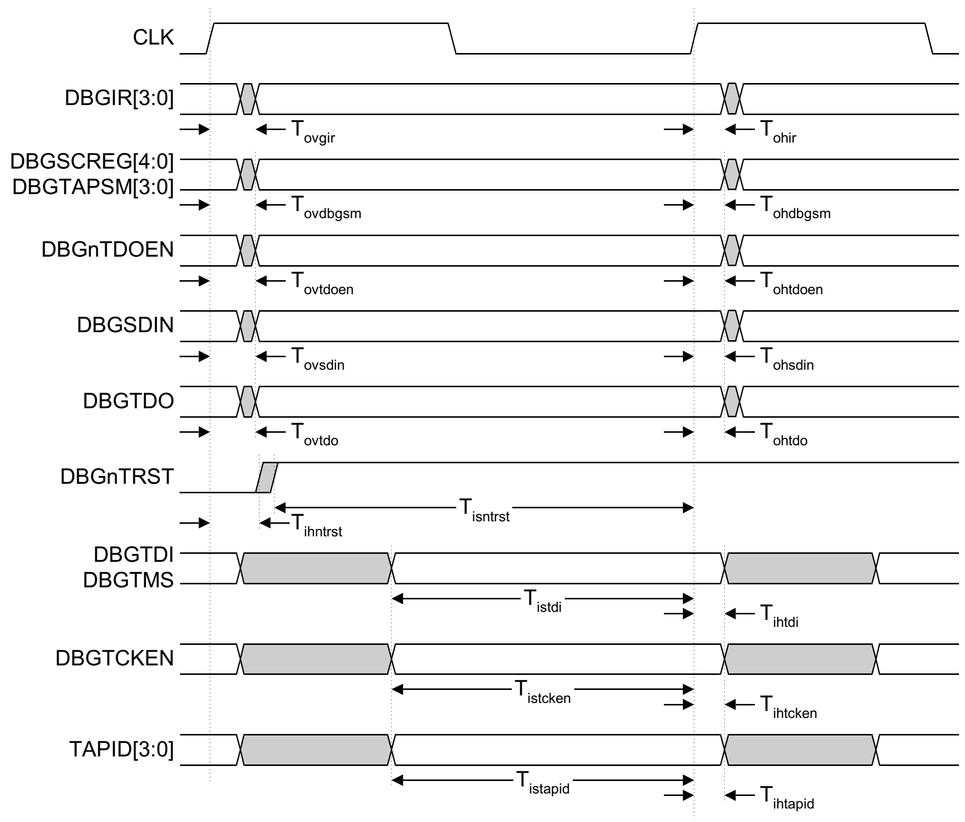


Figure A-6 JTAG interface timing

Table A-6 shows the parameter definitions for JTAG interface timing.

Table A-6 Parameter definitions for JTAG interface timing

Symbol	Parameter	Min	Max
T_{ovir}	Rising **CLK** to **DBGIR[3:0]** valid	-	25%
T_{ohir}	**DBGIR[3:0]** hold time from rising **CLK**	0%	-
$T_{ovdbgsm}$	Rising **CLK** to debug state valid	-	30%
$T_{ohdbgsm}$	Debug state hold time from rising **CLK**	0%	-
$T_{ovtdoen}$	Rising **CLK** to **DBGnTDOEN** valid	-	40%
$T_{ohtdoen}$	**DBGnTDOEN** hold time from rising **CLK**	0%	-

Table A-6 Parameter definitions for JTAG interface timing (continued)

Symbol	Parameter	Min	Max
T_{ovsdin}	Rising **CLK** to **DBGSDIN** valid	-	20%
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** de-asserted input setup to rising **CLK**	25%	-
$T_{ihntrst}$	**DBGnTRST** input hold from rising **CLK**	0%	-
T_{istdi}	Tap state control input setup to rising **CLK**	25%	-
T_{ihtdi}	Tap state control input hold from rising **CLK**	0%	-
$T_{istcken}$	**DBGTCKEN** input setup to rising **CLK**	50%	-
$T_{ihtcken}$	**DBGTCKEN** input hold from rising **CLK**	0%	-
$T_{istapid}$	**TAPID[3:0]** input setup to rising **CLK**	35%	-
$T_{ihtapid}$	**TAPID[3:0]** input hold from rising **CLK**	0%	-

A combinatorial path timing parameter exists from the **DBGSDOUT** input to **DBGTDO** output, as shown in Figure A-7.

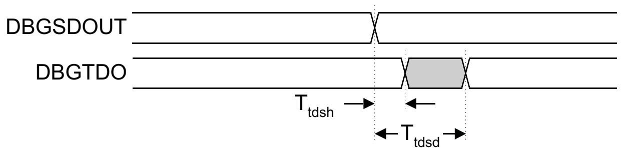


Figure A-7 DBGSDOUT to DBGTDO timing

Table A-7 shows the parameter definitions for **DBGSDOUT** to **DBGTDO** timing.

Table A-7 Parameter definitions for DBGSDOUT to DBGTDO timing

Symbol	Parameter	Min	Max
T_{tdsd}	**DBGTDO** delay from **DBGSDOUTBS** changing	-	30%
T_{tdsh}	**DBGTDO** hold time from **DBGSDOUTBS** changing	0%	-

Figure A-8 shows the exception and configuration timing parameters.

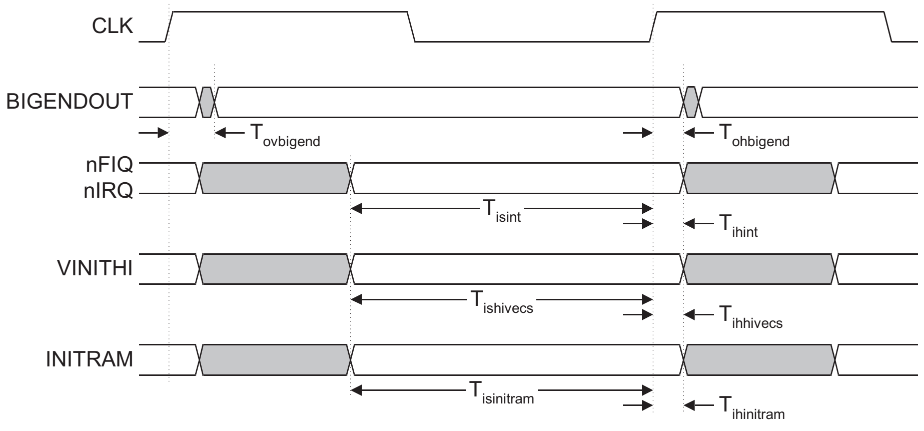


Figure A-8 Exception and configuration timing

Table A-8 shows the parameter definitions for exception and configuration timing.

Table A-8 Parameter definitions for exception and configuration timing

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 to rising **CLK**	15%	-
T_{ihint}	Interrupt input hold from rising **CLK**	0%	-
$T_{ishivecs}$	**VINITHI** input setup to rising **CLK**	90%	-
$T_{ihhivecs}$	**VINITHI** input hold from rising **CLK**	0%	-
$T_{isinitram}$	**INITRAM** input setup to rising **CLK**	90%	-
$T_{ihinitram}$	**INITRAM** input hold from rising **CLK**	0%	-

Note

The **VINTHI** and **INITRAM** signals are specified as 90% of the cycle because it is for input configuration during reset and can be considered static.

Figure A-9 on page A-12 shows the TCM interface timing parameters.

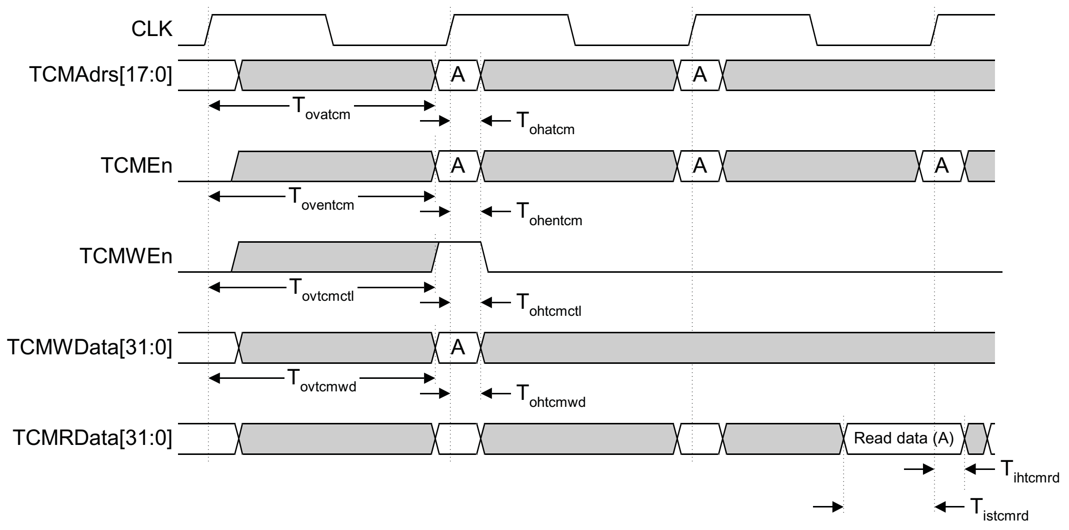


Figure A-9 TCM interface timing

Table A-9 shows the parameter definitions for TCM interface timing.

Table A-9 Parameter definitions for TCM interface timing

Symbol	Parameter	Min	Max
T_{ovatcm}	Rising **CLK** to **TCMAdrs[17:0]** valid	-	90%
T_{ohatcm}	**TCMAdrs[17:0]** hold time from rising **CLK**	0%	-
$T_{oventcm}$	Rising **CLK** to **TCMEn** valid	-	90%
$T_{ohentcm}$	**TCMEn** hold time from rising **CLK**	0%	-
$T_{ovtcmctl}$	Rising **CLK** to TCM control signals valid	-	90%
$T_{ohtcmctl}$	TCM control signals hold time from rising **CLK**	0%	
$T_{istcmrd}$	**TCMRData[31:0]** input setup to rising **CLK**	30%	-
$T_{ihtcmrd}$	**TCMRData[31:0]** input hold from rising **CLK**	0%	-
$T_{ovtcmwd}$	Rising **CLK** to **TCMWData[31:0]** valid	-	90%
$T_{ohtcmwd}$	**TCMWData[31:0]** hold time from rising **CLK**	0%	-

Figure A-10 on page A-13 shows the ETM interface timing parameters.

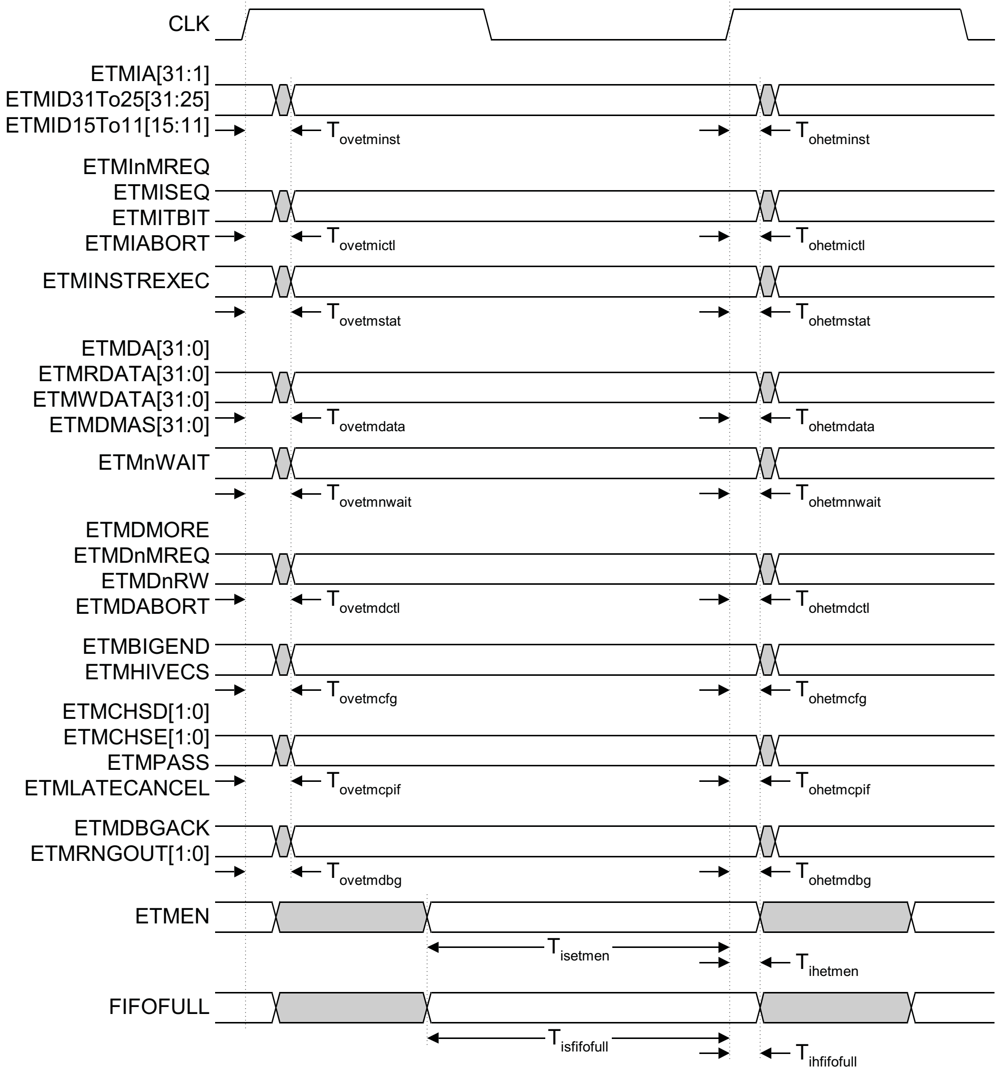


Figure A-10 ETM interface timing

Table A-10 shows the parameter definitions for ETM interface timing.

Table A-10 Parameter definitions for ETM interface timing

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 **INSTREXEC** valid	-	30%
$T_{ohetmstat}$	**INSTREXEC** 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 **nWAIT** valid	-	30%
$T_{ohetmnwait}$	**nWAIT** 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%	-
$T_{isetmen}$	**EN** input setup to rising **CLK**	50%	-
$T_{ihetmen}$	**EN** input hold from rising **CLK**	0%	-
$T_{isfifofull}$	**ETMFIFOFULL** input setup to rising **CLK**	50%	-
$T_{ihfifofull}$	**ETMFIFOFULL** input hold from rising **CLK**	0%	-

Appendix B
Signal Descriptions

This appendix describes the interfaces to the ARM946E-S processor. It contains the following sections:

B.1 Signal properties and requirements

To ensure ease of integration of the ARM946E-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 ARM946E-S processor 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, because bidirectional signals are not used.

Note

You must use external logic to synchronize asynchronous signals, for example, interrupt sources, before applying them to the ARM946E-S processor.

B.2 Clock interface signals

Table B-1 shows the ARM946E-S clock interface signals.

Table B-1 Clock interface signals

Name	Direction	Description
CLK System clock	Input	This clock times all operations in the ARM946E-S processor design. All outputs change from the rising edge and all inputs are sampled on the rising edge. The clock can be stretched in either phase. Using the **HCLKEN** signal, this clock also times AHB operations. Using 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 that the ARM946E-S processor is embedded in. 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 using the JTAG interface. When HIGH on the rising edge of **CLK** the debug logic can advance.
GateTheCLK	Output	Clock control signal for Wait For Interrupt. When asserted, the **CLK** input can be stopped to minimize power.[a]
UnGatedCLK	Input	Free-running clock that is only used to wake-up the processor from the power-saving mode.

a. When **CLK** is disabled, generating a debug request within the ARM946E-S processor does not re-enable the core.

B.3 TCM interface signals

Table B-2 shows the ARM946E-S TCM interface signals.

Table B-2 TCM interface signals

Signal	Direction	Description
DTCMAdrs[17:0]	Output	Data TCM address. This is a word address.
DTCMWData[31:0]	Output	Write data to the TCM.
DTCMRData[31:0]	Input	Read data from the TCM.
DTCMEn	Output	Data TCM enable.
DTCMWEn[3:0]	Output	Data TCM write enables. There is one write enable for each byte.
PhyDTCMSize[3:0]	Input	Encoded size of the Data TCM. The encoding for these signals is given in Table 2-8 on page 2-11.
ITCMAdrs[17:0]	Output	Instruction TCM address. This is a word address.
ITCMWData[31:0]	Output	Write data to the Instruction TCM.
ITCMRData[31:0]	Input	Read data from the Instruction TCM.
ITCMEn	Output	Instruction TCM enable.
ITCMWEn[3:0]	Output	Instruction TCM write enables. There is one write enable for each byte.
PhyITCMSize[3:0]	Input	Encoded size of the Instruction TCM. The encoding for these signals is given in Table 2-8 on page 2-11.

B.4 AHB signals

Table B-3 shows the ARM946E-S AHB signals.

Table B-3 AHB signals

Name	Direction	Description
HADDR[31:0] Address bus	Output	The 32-bit AHB system address bus.
HBURST[2:0] Burst type	Output	Indicates if the transfer forms part of a burst. The ARM946E-S processor supports: • SINGLE transfer cycle (b000) • incremental burst cycles: — INCR(b001) — INCR4(b011) — INCR8(b101).
HBUSREQ Bus request	Output	Indicates that the ARM946E-S processor requires the bus.
HGRANT Bus grant	Input	Indicates that the ARM946E-S processor is currently the highest priority master. Ownership of the address/control signals changes at the end of a transfer when **HREADY** is HIGH, so the ARM946E-S processor gets access to the bus when both **HREADY** and **HGRANT** are HIGH.
HLOCK Request locked transfers	Output	When HIGH, indicates that the ARM946E-S processor requires locked access to the bus and no other master must be granted until this signal has gone LOW. Asserted by the ARM946E-S processor when executing SWP instructions to AHB address space.
HPROT[3:0] Protection control	Output	Indicates that the ARM946E-S processor transfer is an: • opcode fetch (b---0) • data access (b---1). Indicates if the transfer is: • User mode access (b--0-) • Supervisor mode access (b--1-). Indicates that an access is: • nonbufferable (b-0--) • bufferable (b-1--). Indicates that an access is: • noncachable (b0---) • cachable (b1---)

Table B-3 AHB signals (continued)

Name	Direction	Description
HRDATA[31:0] Read data bus	Input	The 32-bit read data bus transfers data from a selected bus slave to the ARM946E-S processor during read operations.
HREADY Transfer done	Input	When HIGH indicates that a transfer has finished on the bus. This signal can be driven LOW by the selected bus slave to extend a transfer.
HRESETn Not reset	Input	This is the active LOW reset signal for initializing the ARM946E-S processor system state. This signal can be asserted asynchronously but must be deasserted synchronously.
HRESP[1:0] Transfer response	Input	The transfer response from the selected slave provides additional information on the status of the transfer. The response can be: • OKAY (b00) • ERROR (b01) • RETRY (b10) • SPLIT (b11).
HSIZE[2:0] Transfer size	Output	Indicates the size of an ARM946E-S processor transfer. This can be: • Byte (b000) • Halfword (b001) • Word (b010). Bit 2 is tied LOW.
HTRANS[1:0] Transfer type	Output	Indicates the type of ARM946E-S processor transfer. This can be: • IDLE (b00) • BUSY (b01) • NONSEQ (b10) • SEQ (b11).
HWDATA[31:0] Write data bus	Output	The 32-bit write data bus transfers data from the ARM946E-S processor to a selected bus slave during write operations.
HWRITE Transfer direction	Output	When HIGH indicates a write transfer. When LOW indicates a read transfer.

B.5 Coprocessor interface signals

Table B-4 shows the ARM946E-S coprocessor interface signals.

Table B-4 Coprocessor interface signals

Name	Direction	Description
CPCLKEN Coprocessor clock enable	Output	Synchronous enable for coprocessor pipeline follower. When HIGH on the rising edge of **CLK** the pipeline follower logic can advance.
CPINSTR[31:0] Coprocessor instruction data	Output	The 32-bit coprocessor instruction bus used to transfer instructions to the coprocessor pipeline follower.
CPDOUT[31:0] Coprocessor read data	Output	The 32-bit coprocessor read data bus for transferring data to the coprocessor.
CPDIN[31:0] Coprocessor write data	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, that 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 causes a Data Abort to occur.
CHSDE[1:0] Coprocessor handshake decode	Input	The handshake signals from the Decode stage of the coprocessor pipeline follower. Indicates: • ABSENT (b10) • WAIT (b00) • GO (b01) • LAST (b11).
CHSEX[1:0] Coprocessor handshake execute	Input	The handshake signals from the Execute stage of the coprocessor pipeline follower. Indicates: • ABSENT (b10) • WAIT (b00) • GO (b01) • LAST (b11).

Table B-4 Coprocessor interface signals (continued)

Name	Direction	Description
CPTBIT Coprocessor instruction Thumb bit	Output	When HIGH indicates that the ARM946E-S processor is in Thumb state. When LOW indicates that the ARM946E-S processor is in ARM state. Sampled by the coprocessor pipeline follower.
nCPMREQ Not coprocessor instruction request	Output	When LOW on the rising edge of **CLK** and **CPCLKEN** is HIGH, the instruction on **CPINSTR** must enter the coprocessor pipeline.
nCPTRANS Not coprocessor memory translate	Output	When LOW indicates that the ARM946E-S processor is in User mode. When HIGH indicates that the ARM946E-S processor is in Privileged mode. Sampled by the coprocessor pipeline follower.

B.6 Debug signals

Table B-5 shows the ARM946E-S debug signals.

Table B-5 Debug signals

Name	Direction	Description
COMMRX Communication channel receive	Output	When HIGH denotes that the communication channel receive buffer contains valid data waiting to be read.
COMMTX Communication channel transmit	Output	When HIGH, denotes that the communication channel transmit buffer is empty waiting for data to be written.
DBGACK Debug acknowledge	Output	When HIGH indicates that the processor is in debug state.
DBGDEWPT Data watchpoint	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 ARM946E-S processor to enter debug state.
DBGEN Debug enable	Input	This signal enables the debug features of the ARM9E-S core. If you intend to use the ARM9E-S debug features, tie this signal HIGH. Drive this signal LOW only when debugging is not required.
DBGEXT[1:0] EmbeddedICE-RT external input	Input	Input to the EmbeddedICE-RT logic enables breakpoints/watchpoints to be dependent on external conditions.
DBGIEBKPT Instruction breakpoint	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 ARM946E-S processor to enter debug state if that instruction reaches the Execute stage of the processor pipeline.
DBGINSTREXEC Instruction executed	Output	Indicates that the instruction in the Execute stage of the processors pipeline has been executed.
DBGRNG[1:0] EmbeddedICE-RT Rangeout	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.
DBGRQI Internal debug request	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 External debug request	Input	An external debugger can force the processor into debug state by asserting this signal.

B.7 JTAG signals

Table B-6 shows the ARM946E-S JTAG signals.

Table B-6 JTAG signals

Name	Direction	Description
DBGIR[3:0] TAP Controller Instruction Register	Output	These four bits reflect the current instruction loaded into the TAP Controller Instruction Register. These bits change when the TAP controller is in the UPDATE-IR state.
DBGnTRST Not test reset	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 Not **DBGTDO** enable	Output	When LOW, the serial data is being driven out of the **DBGTDO** output. Normally used as an output enable for a **DBGTDO** signal 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 External scan chain serial input data	Output	Contains the serial data to be applied to an external scan chain.
DBGSDOUT External scan chain serial data output	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] TAP controller state machine	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.
TAPID[31:0] Boundary scan ID code	Input	Specifies the ID code value shifted out on **DBGTDO** when the IDCODE instruction is entered into the TAP controller.

B.8 Miscellaneous signals

Table B-7 shows the miscellaneous signals on the ARM946E-S processor.

Table B-7 Miscellaneous signals

Name	Direction	Description
BIGENDOUT	Output	When HIGH, the ARM946E-S processor treats bytes in memory as being in big-endian format. When LOW, memory is treated as little-endian.
nFIQ Not fast interrupt request	Input	This is the Fast Interrupt Request signal. This signal must be synchronous to **CLK**.
nIRQ Not interrupt request	Input	This is the Interrupt Request signal. This signal must be synchronous to **CLK**.
VINITHI Exception vector location at reset	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`.
TESTMODE	Input	Prevents the cache from being inadvertently flushed when scan patterns are shifted through the scan chains. Must only be asserted during scan test of the ARM946E-S processor.
INITRAM	Input	Determines if the Instruction TCM is enabled at reset. If HIGH, it is enabled, if LOW, it is disabled.
DCacheSize[3:0]	Input	Encoded size of the data cache. The encoding for these signals is given in Table 2-5 on page 2-9.
ICacheSize[3:0]	Input	Encoded size of the instruction cache. The encoding for these signals is given in Table 2-5 on page 2-9.

B.9 ETM interface signals

Table B-8 shows the ARM946E-S ETM interface signals.

Table B-8 ETM interface signals

Name	Direction	Description
ETMEN	Input	Synchronous ETM interface enable. This signal must be tied LOW if an ETM is not used.
ETMFIFOFULL	Input	Indication that the ETM FIFO is FULL, and that trace data might be lost. The ARM946E-S stalls on the next instruction boundary. This signal must be tied low if an ETM is not used.
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.
ETMIABORT	Output	Instruction Abort for the ETM.
ETMDA[31:0]	Output	Data address for the ETM.
ETMDMAS[1:0]	Output	Data size indication for the ETM.
ETMDMORE	Output	More sequential data indication for the ETM.
ETMDnMREQ	Output	Data memory request for the ETM.
ETMDnRW	Output	Data not read/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.

Table B-8 ETM interface signals (continued)

Name	Direction	Description
ETMRNGOUT[1:0]	Output	Watchpoint register match indication for the ETM.
ETMID31To25[31:25]	Output	Instruction data field for the ETM.
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 identifier for the ETM.
ETMPROCIDWR	Output	ETMPROCID write strobe.
ETMINSTRVALID	Output	Instruction valid indication for the ETM.

B.10 INTEST wrapper signals

The INTEST wrapper is optionally added as part of the synthesis process. That is, it does not form part of the source RTL. The signals associated with this wrapper are therefore redundant but remain for backward compatibility with ARM946E-S Rev 0. The signals, **SI**, **SO**, **SCANEN**, **TESTEN**, **SERIALEN**, and **INnotEXTEST** are therefore unconnected and have no function.

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 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 High-performance Bus (AHB)

The AMBA Advanced High-performance Bus system connects embedded processors such as an ARM core to high-performance peripherals, DMA controllers, on-chip memory, and interfaces. It is a high-speed, high-bandwidth bus that supports multi-master bus management to maximize system performance.

See also Advanced Microcontroller Bus Architecture.

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.

See also Advanced High-performance Bus.

Aligned

Refers to data items stored so that their address is divisible by the highest power of two that divides their size. Aligned words and halfwords therefore have addresses that are divisible by four and two respectively. The terms word-aligned and halfword-aligned therefore refer to addresses that are divisible by four and two respectively. The terms byte-aligned and doubleword-aligned are defined similarly.

AMBA

See Advanced Microcontroller Bus Architecture.

Architecture

The organization of hardware and/or software that characterizes a processor and its attached components, and enables devices with similar characteristics to be grouped together when describing their behavior, for example, Harvard architecture, instruction set architecture, ARMv6 architecture.

ARM state

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

See also Thumb state.

Base register

A register specified by a load or store instruction that is used to hold the base value for the address calculation for the instruction. Depending on the instruction and its addressing mode, an offset can be added to or subtracted from the base register value to form the virtual address that is sent to memory.

Base register write-back

Updating the contents of the base register used in an instruction target address calculation so that the modified address is changed to the next higher or lower sequential address in memory. This means that it is not necessary to fetch the target address for successive instruction transfers and enables faster burst accesses to sequential memory.

Big-endian

Memory organization in which the least significant byte of a word is at a higher address than the most significant byte.

See also Little-endian and Endianness.

Block address	An address that comprises a tag, an index, and a word field. The tag bits identify the way that contains the matching cache entry for a cache hit. The index bits identify the set being addressed. The word field contains the word address that can be used to identify specific words, halfwords, or bytes within the cache entry.
Breakpoint	A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is halted unconditionally. Breakpoints are inserted by programmers to allow inspection of register contents, memory locations, and/or variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested. *See also* Watchpoint.
Burst	A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AHB buses are controlled using the **HBURST** signals to specify if transfers are single, four-beat, eight-beat, or 16-beat bursts, and to specify how the addresses are incremented.
Cache	A block of on-chip or off-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions and/or data. This is done to increase the average speed of memory accesses and therefore to increase processor performance.
Cache hit	A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.
Cache line	The basic unit of storage in a cache. It is always a power of two words in size (usually 4 or 8 words), and is required to be aligned to a suitable memory boundary. *See also* Cache terminology.
Cache lockdown	To fix a line in cache memory so that it cannot be overwritten. Cache lockdown enables critical instructions and/or data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. This ensures that all subsequent accesses to the instructions or data concerned are cache hits, and therefore complete as quickly as possible.
Cache miss	A memory access that cannot be processed at high speed because the instruction or data it addresses is not in the cache and a main memory access is required.
Cache set	A cache set is a group of cache lines (or blocks). A set contains all the ways that can be addressed with the same index. The number of cache sets is always a power of two.
Cast out	*See* Victim.
Central Processing Unit (CPU)	The part of a processor that contains the ALU, the registers, and the instruction decode logic and control circuitry. Also commonly known as the processor core.

Clean

A cache line that has not been modified while it is in the cache is said to be clean. To clean a cache is to write dirty cache entries into main memory. If a cache line is clean, it is not written on a cache miss because the next level of memory contains the same data as the cache.

See also Dirty.

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.

CPU

See Central Processing Unit.

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 attempting to access invalid data memory.

See also Abort, External Abort, and Prefetch Abort.

Data Cache (DCache)

A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used data. This is done to greatly increase the average speed of memory accesses and therefore to increase processor performance.

DCache

See Data Cache.

Debug 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 Communications Channel.

Debugger

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

An application that monitors and controls the operation of a second application. Usually used to find errors in the application program flow.

Dirty

A cache line in a Write-Back cache that has been modified while it is in the cache is said to be dirty. A cache line is marked as dirty by setting the dirty bit. If a cache line is dirty, it must be written to memory on a cache miss because the next level of memory contains data that has not been updated. The process of writing dirty data to main memory is called cache cleaning.

See also Clean.

EmbeddedICE-RT

The JTAG-based hardware provided by debuggable ARM processors to aid debugging in real-time.

Embedded Trace Macrocell (ETM)	A hardware macrocell that outputs instruction and data trace information on a trace port.
Endianness	Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory. *See also* Little-endian and Big-endian.
ETM	*See* Embedded Trace Macrocell.
Exception	An event that occurs during program operation that makes continued normal operation inadvisable or impossible, and so makes it necessary to change the flow of control in a program. Exceptions can be caused by error conditions in hardware or software. The processor can respond to exceptions by running appropriate exception handler code that attempts to remedy the error condition, and either restarts normal execution or ends the program in a controlled way.
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.
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.
Halt mode	One of two mutually exclusive debug modes. In halt mode all processor execution halts when a breakpoint or watchpoint is encountered. All processor state, coprocessor state, memory and input/output locations can be examined and altered by the JTAG interface. *See also* Monitor mode.
High vectors	Alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.
Host	A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.
ICache	*See* Instruction Cache.
Index register	A register specified in some load or store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address, which is sent to memory. Some addressing modes optionally enable the index register value to be shifted prior to the addition or subtraction.

Instruction Cache (ICache)

A block of on-chip fast access memory locations, situated between the processor and main memory, used for storing and retrieving copies of often used instructions. This is done to increase the average speed of memory accesses and therefore to increase processor performance.

Invalidate

To mark a cache line as being not valid by clearing the valid bit. This must be done whenever the line does not contain a valid cache entry. For example, after a cache flush all lines are invalid.

Little-endian

Memory organization where the least significant byte of a word is at a lower address than the most significant byte.

See also Big-endian and Endianness.

Load/store architecture

A processor architecture where data-processing operations only operate on register contents, not directly on memory contents.

Macrocell

A complex logic block with a defined interface and behavior. A typical VLSI system comprises several macrocells (such as an ARM processor, an Embedded Trace Macrocell, and a memory block) plus application-specific logic.

Monitor mode

One of two mutually exclusive debug modes. In monitor mode the ARM1136JF-S processor enables a software abort handler provided by the debug monitor or operating system debug task. When a breakpoint or watchpoint is encountered, this enables vital system interrupts to continue to be serviced while normal program execution is suspended.

See also Halt mode.

Prefetching

In pipelined processors, the process of fetching instructions from memory to fill up the pipeline before the preceding instructions have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

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.

Read

Reads are defined as memory operations that have the semantics of a load. That is, the ARM instructions LDM, LDRD, LDC, LDR, LDRT, LDRSH, LDRH, LDRSB, LDRB, LDRBT, LDREX, RFE, STREX, SWP, and SWPB, and the Thumb instructions LDM, LDR, LDRSH, LDRH, LDRSB, LDRB, and POP. Java instructions that are accelerated by hardware can cause a number of reads to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

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.

Reserved A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as zero and are read as zero.

SBO *See* Should Be One.

SBZ *See* Should Be Zero.

SBZP *See* Should Be Zero or Preserved.

Scan chain A scan chain is made up of serially-connected devices that implement boundary scan technology using a standard JTAG TAP interface. Each device contains at least one TAP controller containing shift registers that form the chain connected between **TDI** and **TDO**, through which test data is shifted. Processors can contain several shift registers to enable you to access selected parts of the device.

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.

Should Be Zero or Preserved (SBZP)

Should be written as 0 (or all 0s for bit fields) by software, or preserved by writing the same value back that has been previously read from the same field on the same processor.

Tag The upper portion of a block address used to identify a cache line within a cache. The block address from the CPU is compared with each tag in a set in parallel to determine if the corresponding line is in the cache. If it is, it is said to be a cache hit and the line can be fetched from cache. If the block address does not correspond to any of the tags it is said to be a cache miss and the line must be fetched from the next level of memory.

TAP *See* Test Access Port.

Test Access Port (TAP)	The collection of four mandatory terminals and one optional terminal 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.
Unaligned	Memory accesses that are not appropriately word-aligned or halfword-aligned. *See also* Aligned.
Undefined	Indicates an instruction that generates an Undefined instruction trap. See the *ARM Architecture Reference Manual* for more information on ARM exceptions.
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.
Vector operation	An operation involving more than one destination register, perhaps involving different source registers in the generation of the result for each destination.
Victim	A cache line, selected to be discarded to make room for a replacement cache line that is required as a result of a cache miss. The way in which the victim is selected for eviction is processor-specific. A victim is also known as a cast out.
Watchpoint	A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to enable inspection of register contents, memory locations, and variable values when memory is written to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested. *See also* Breakpoint.
WB	*See* Write-back.
Write	Writes are defined as operations that have the semantics of a store. That is, the ARM instructions SRS, STM, STRD, STC, STRT, STRH, STRB, STRBT, STREX, SWP, and SWPB, and the Thumb instructions STM, STR, STRH, STRB, and PUSH. Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.
Write-back(WB)	In a write-back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. (Also known as copyback).

Write buffer	A block of high-speed memory, arranged as a FIFO buffer, between the Data Cache and main memory, whose purpose is to optimize stores to main memory. Each entry in the write buffer can contain the address of a data item to be stored to main memory, the data for that item, and a sequential bit that indicates if the next store is sequential or not.
Write-through (WT)	In a write-through cache, data is written to main memory at the same time as the cache is updated.
WT	*See* Write-through.