ARM® 946E-S

Technical Reference Manual

ARM®

ARM DDI 0155A

ARM 946E-S
Technical Reference Manual

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Change
11th August 2000	A	First release

Proprietary Notice

Confidentiality Status

This document is Open Access. This means there is no restriction on the distribution of information.

Product Status

The information in this document is Final (information on a developed product).

Web Address

`http://www.arm.com`

Contents
ARM 946E-S Technical Reference Manual

List of Tables
ARM 946E-S Technical Reference Manual

List of Figures
ARM 946E-S Technical Reference Manual

Preface

This preface introduces the ARM946E-S and its reference documentation. It contains the following sections:

About this document

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

Intended audience

This document has been written for hardware and software engineers who want to design or develop products based upon the ARM946E-S processor. It assumes no prior knowledge of ARM products.

Using this manual

This document is organized into the following chapters:

Chapter 1 ***Introduction***

This chapter provides an introduction to the ARM946E-S.

Chapter 2 ***Programmer's Model***

This chapter describes the programmer's model of the ARM946E-S 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 protection unit.

Chapter 5 ***Tightly-coupled SRAM***

This chapter describes the requirements and operation of the tightly-coupled SRAM.

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 ***Debug Support***

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

Chapter 9 ***ETM Interface***

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

Chapter 10 ***Test Support***

This chapter describes the test methodology used for the ARM946E-S synthesized logic and tightly-coupled SRAM.

Appendix A ***AC Parameters***

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

Appendix B ***Signal Descriptions***

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

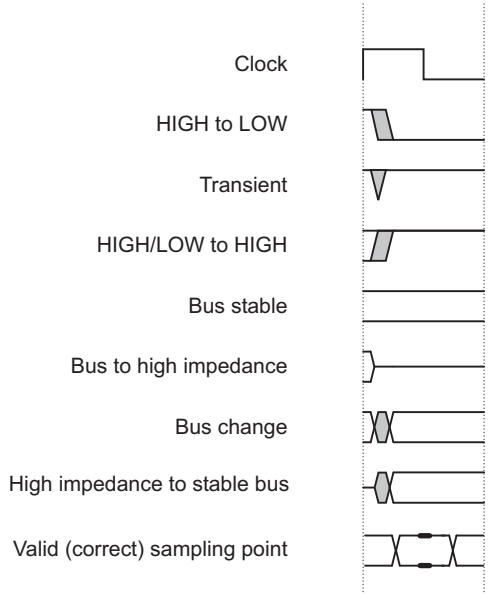
Typographical conventions

The following typographical conventions are used in this document:

bold	Highlights ARM processor signal names within text, and interface elements such as menu names. Can also be used for emphasis in descriptive lists where appropriate.
italic	Highlights special terminology, cross-references and citations.
`typewriter`	Denotes text that can be entered at the keyboard, such as commands, file names and program names, and source code.
`typewriter`	Denotes a permitted abbreviation for a command or option. The underlined text may be entered instead of the full command or option name.
`typewriter italic`	Denotes arguments to commands or functions where the argument is to be replaced by a specific value.
`typewriter bold`	Denotes language keywords when used outside example code.

Timing diagram conventions

This manual contains a number of timing diagrams. The following key explains the components used in these diagrams. Any variations are clearly labeled when they occur. Therefore, no additional meaning should be attached unless specifically stated.



Key to timing diagram conventions

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.

Further reading

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

If you would like further information on ARM products, or if you have questions not answered by this document, please contact `info@arm.com` or visit out web site at `http://www.arm.com`.

ARM publications

ARM Architecture Reference Manual (ARM DDI 0100).

ARM9E-S Technical Reference Manual (ARM DDI 0165).

AMBA Specification (Rev 2.0) (ARM IHI 0011).

Other publications

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

Feedback

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

Feedback on the ARM946E-S

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

- the product name
- a concise explanation of your comments

Feedback on the document

If you have any comments about this document, please 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.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

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

1.1 About the ARM946E-S

The ARM946E-S is a synthesizable macrocell combining an ARM processor. It is a member of the ARM9 Thumb family of high-performance, 32-bit system-on-chip processor solutions.

The ARM946E-S has tightly-coupled SRAM memory, and instruction and data caches and is targeted at a wide range of embedded applications where high-performance, low system cost, small die size and low power are all important.

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

- An ARM9E-S RISC integer CPU core featuring:
  - ARMv5TExP 32-bit instruction set with improved ARM/Thumb code interworking and enhanced multiplier designed for improved DSP performance.
  - ARM debug architecture with additional support for real-time debug. This allows critical exception handlers to execute while debugging the system.
- Tightly-coupled SRAM for each of the instruction and data CPU interfaces. The size of both the instruction and data SRAM are implementor-configurable.
- Instruction and data caches. The design can be easily modified to allow any combination of caches from 4 Kbytes to 1 Mbyte.
- A protection unit that allows the memory to be segmented and protected in a simple manner, ideal for embedded control applications.
- An AMBA AHB bus interface. ARM946E-S 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 allowing 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 tightly-coupled SRAM 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-a-chip designer to concentrate on design issues unique to their system. The synthesizable nature of the device eases integration into ASIC technologies.

1.2 Microprocessor block diagram

The ARM946E-S block diagram is shown in Figure 1-1 on page 1-4.

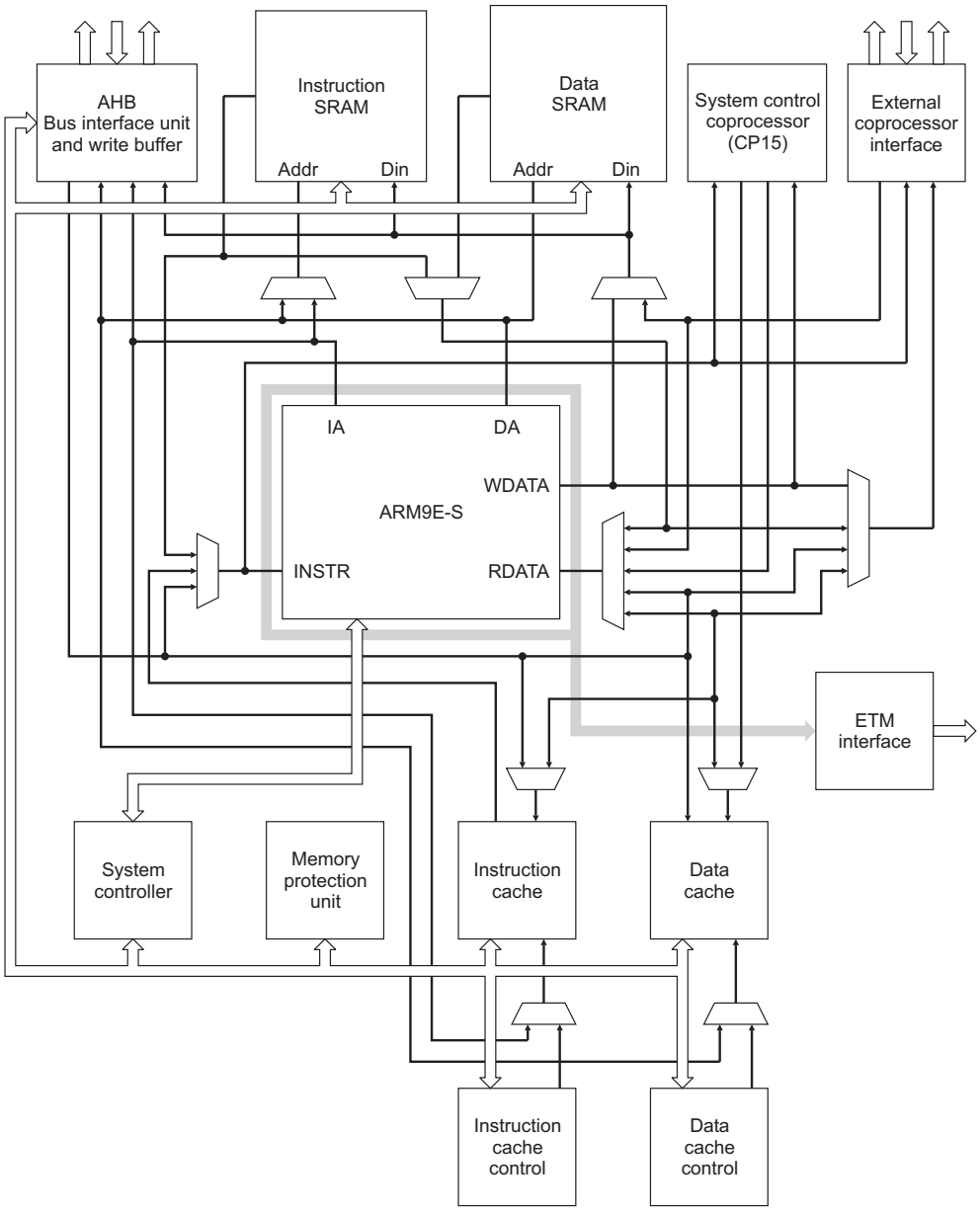


Figure 1-1 ARM946E-S block diagram

The blocks shown in Figure 1-1 on page 1-4are described in the locations listed in Table 1-1.

Table 1-1 Location of block descriptions

Block	Location of description
ARM9E-S	ARM9E-S Technical Reference Manual
AHB bus interface unit and write buffer	Chapter 6 *Bus Interface Unit and Write Buffer*
Instruction SRAM	Chapter 5 *Tightly-coupled SRAM*
Data SRAM	Chapter 5 *Tightly-coupled SRAM*
System control coprocessor (CP15)	Chapter 2 *Programmer's Model*
External coprocessor interface	Chapter 7 *Coprocessor Interface*
ETM interface	Chapter 9 *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*

Chapter 2
Programmer's Model

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

2.1 About the ARM946E-S programmer's model

The programmer's model for the ARM946E-S macrocell 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:

- CP14 within the ARM9E-S core allows software access to the debug communications channel
- CP15 allows configuration of the caches, tightly-coupled SRAM, 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 communications channel* on page 8-30.

The registers defined in CP15 are accessible with `MCR` and `MRC` instructions, and are described in *CP15 register map summary* on page 2-4.

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 processor core implements the ARMv5TExP 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*. Contact ARM for complete descriptions of both instruction sets.

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 unwind any base register update that might have been specified by the aborted instruction.

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

2.3 CP15 register map summary

The ARM946E-S incorporates CP15 for system control. CP15 allows configuration of the caches, tightly-coupled SRAM, and protection unit. It also allows configuration of ARM946E-S system options including big or little-endian operation. The register map for CP15 is shown in Table 2-1.

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
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
8	Unpredictable	Unpredictable
9	Cache lockdown b	Cache lockdown b
9	Tightly-coupled memory region b	Tightly-coupled memory region b
10	Unpredictable	Unpredictable
11	Unpredictable	Unpredictable
12	Unpredictable	Unpredictable
13	Process ID	Process ID
14	Unpredictable	Unpredictable
15	RAM and TAG BIST test a	RAM and TAG BIST test a
15	Test state a	Test state a
15	Cache debug index a	Cache debug index 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 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 zero by **HRESETn** except V-Bit in register 1, that takes the value of macrocell input **VINITHI when HRESETn is asserted**.

I-SRAM and D-SRAM sizes in register 9 reflect the physical I-SRAM and D-SRAM sizes.

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 on page 2-6.

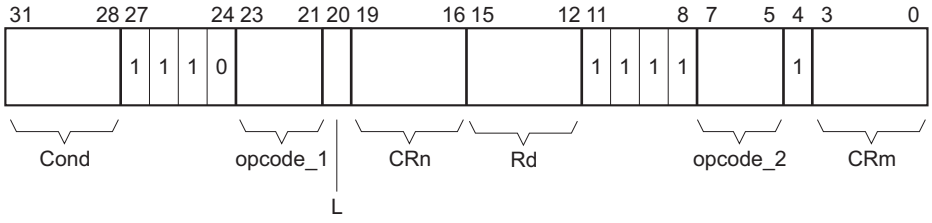


Figure 2-1 CP15 MRC and MCR bit pattern

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
```

The contents of the ID code are shown in Table 2-3.

Table 2-3 Register 0, ID code

Register bits	Function	Value
31:24	Implementor	`0x41`
23:20	Reserved (variant)	`0x00`

Table 2-3 Register 0, ID code (continued)

Register bits	Function	Value
19:16	Architecture version ARM5TExP	0x04
15:4	Part number	0x946
3:0	Version (implementation-specific)	Revision

2.3.3 Register 0, Cache type register

This is a read-only register that contains information about the size and architecture of the instruction cache (ICache) and data cache (DCache), allowing operating systems to establish how to perform operations such as cache cleaning and lockdown. Future ARM cached processors will contain this register, allowing 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:

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

The format of the register is shown in Table 2-4.

Table 2-4 Cache type register format

Register bits	Function	Value
31:29	Reserved	000
28:25	Cache type	0111
24	Harvard/Unified	1 (defines Harvard cache)
23:22	Reserved	00
21:18	DCache size	Implementation-specific
17:15	DCache associativity	Implementation-specific
14	DCache base size	Implementation-specific
13:12	DCache words per line	10 (defines 8 words per line)
11:10	Reserved	00
9:6	ICache size	Implementation-specific

Table 2-4 Cache type register format (continued)

Register bits	Function	Value
5:3	ICache associativity	Implementation-specific
2	ICache base size	Implementation-specific
1:0	ICache words per line	10 (defines 8 words per line)

Bits [28:25] indicate which major cache class the implementation falls into. `0x7` 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
000	Direct mapped
010	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 `010` 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 `000` to indicate 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.

Bit 14 gives the data cache base size.

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.

2.3.4 Register 0, Tightly-coupled memory size register

This is a read-only register that returns the size of the tightly-coupled instruction and data RAMs included within the ARM946E-S.

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 tightly-coupled memories. The format of the register is shown in Table 2-7.

Table 2-7 Tightly-coupled memory size register

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

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

Table 2-8 Memory size field

Bits [21:8] and bits [9:6]	Tightly-coupled RAM size
b0000	0KB
b0011	4KB
b0100	8KB
b0101	16KB
b0110	32KB
b0111	64KB
b1000	128KB

Table 2-8 Memory size field (continued)

Bits [21:8] and bits [9:6]	Tightly-coupled RAM size
b1001	256KB
b1010	512KB
b1011	1MB

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

2.3.5 Register 1, Control register

This register contains the control bits of the ARM946E-S. 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 lists the functions controlled by register 1.

Table 2-9 Register 1, control register

Register bit	Function
31:20	Reserved (SBZ)
19	Instruction RAM load mode
18	Instruction RAM enable
17	Data RAM load mode
16	Data RAM enable
15	Configure disable loading **TBIT**
14	Round-robin replacement
13	Alternate vector select
12	ICache enable
11:8	Reserved (SBZ)

Table 2-9 Register 1, control register (continued)

Register bit	Function
7	Big-endian
6:3	Reserved (SBO)
2	DCache enable
1	Reserved (SBZ)
0	Protection unit enable

The bits in the control register have the following function.

Bit 19, Instruction RAM load mode

This bit controls the operation of the instruction RAM load mode.

You can use the instruction RAM load mode for initializing the instruction RAM. The instruction RAM load mode allows you to load data into ARM registers from either data cache or main memory, and then write to the same address but within the tightly-coupled instruction RAM. This allows you to copy boot code from memory located at address `0x0` into the instruction RAM which, when enabled, also exists at address `0x0`. The operation of the load mode is described in *I-SRAM load mode* on page 5-3.

At reset this bit is cleared.

Bit 18, Instruction RAM enable

This bit controls operation of the tightly-coupled instruction RAM. When the instruction RAM is enabled, all instruction and data accesses to the instruction RAM address range access the instruction RAM.

At reset this bit is cleared.

Bit 17, Data RAM load mode

This bit controls the operation of the data RAM load mode.

You can use the data RAM load mode for initializing the data RAM. The data RAM load mode allows you to load data into ARM registers from either data cache or main memory, and then write to the same address but within the tightly-coupled data RAM. The operation of the load mode is described in *I-SRAM load mode* on page 5-3.

At reset this bit is cleared.

Bit 16, Data RAM enable

This bit controls operation of the tightly-coupled data RAM. When the data RAM is enabled, it takes precedence over the data cache and AHB for data accesses.

At reset this bit is cleared.

Bit 15, Configure disable loading TBIT

This bit controls the behavior of load PC instructions. When LOW the ARMv5TExP-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 HIGH, this ARMv5TExP behavior is disabled.

At reset this bit is cleared.

Bit 14, Round-robin replacement

This bit controls the cache replacement algorithm.

When HIGH, round-robin replacement is used. When LOW, 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 LOW, the base address for the exception vectors is `0x00000000`. When HIGH, the base address is `0xFFFF0000`.

Note

This bit is initialized either HIGH or LOW during system reset, depending on the value of the input pin, **VINITHI**. This allows 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, ICache enable

Controls the behavior of the ICache.

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

At reset this bit is cleared.

Bit 7, Endian

Selects the endian configuration of the ARM946E-S. When this bit is HIGH, big-endian configuration is selected. When LOW, little-endian configuration is selected.

At reset this bit is cleared.

Bit 2, DCache enable

This bit controls the behavior of the DCache.

To use the data cache, both the protection unit enable bit (bit 0) and the DCache enable bit must be HIGH. 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 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 I and D caches. If the `opcode_2` field = 0, then the data cache bits are programmed. If the `opcode_2` field = 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 given in Table 2-10.

Table 2-10 Programming instruction/data cachable bits

Register bit	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
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
```

The format for the bufferable bits in the data areas is given in Table 2-11.

Table 2-11 Programming data bufferable bits

Register bit	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

Table 2-11 Programming data bufferable bits (continued)

Register bit	Function
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 given in Table 2-12.

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

Register bit	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

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

Register bit	Function
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. The encoding is shown in Table 2-13.

Table 2-13 Access permission encoding (extended)

I/DApn[3:0]	Access permission	
	Privileged	User
0000	No access	No access
0001	Read/write access	No access
0010	Read/write access	Read-only
0011	Read/write access	Read/write access
0100	UNP	UNP
0101	Read-only	No access
0110	Read-only	Read-only
0111	UNP	UNP
1xxx	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
```

The data format for these registers is shown in Table 2-14.

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

Register bit	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
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. The encoding is shown in Table 2-15.

Table 2-15 Access permission encoding (standard)

I/DApn[1:0]	Access permission	
	Privileged	User
00	No access	No access
01	Read/write access	No access
10	Read/write access	Read-only
11	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, 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 *Access permission encoding (extended)* on page 2-17), and then reprogrammed using the standard access permissions (see Table 2-15 on page 2-18), the access permissions applied are as if Apn[3:2] are programmed to 00 in Table 2-13 on page 2-17.

2.3.9 Register 6, Protection region/base size registers

These registers define the protection region base address/size registers. 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 size registers are listed in Table 2-16.

Table 2-16 Accessing protection region/base size registers

ARM instruction	Protection region/ base 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 size register has the format shown in Table 2-17.

Table 2-17 Protection region/base size register format

Register bit	Function
31:12	Region base
5:1	Area size
0	1 = Region enable 0 = Region disable Reset to 0.

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

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

Table 2-18 Area size encoding

Bit encoding	Area size
00000 to 01010	Reserved (UNP)
01011	4KB
01100	8KB
01101	16KB
01110	32KB
01111	64KB
10000	128KB
10001	256KB
10010	512KB
10011	1MB
10100	2MB
10101	4MB
10110	8MB

Table 2-18 Area size encoding (continued)

Bit encoding	Area size
10111	16MB
11000	32MB
11001	64MB
11010	128MB
11011	256MB
11100	512MB
11101	1GB
11110	2GB
11111	4GB

Example base setting

An 8KB size region aligned to an 8KB boundary at `0x0000 2000` (covering the address range `0x0000 2000` to `0x0000 3FFF`) is programmed as `0x0000 2019`.

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 allows the protection region registers to be read.

Writes to the protection region/base size registers with opcode_2 set to 1 are unpredictable.

2.3.10 Register 7, Cache operations register

A write to this register can be used to perform the following operations:

- flush ICache and DCache
- prefetch an ICache line
- wait for interrupt
- drain the write buffer
- clean and flush the DCache.

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

Table 2-19 Cache operations

ARM instruction	Function	Data
`MCR p15, 0, rd, c7, c5, 0`	Flush ICache	SBZ[a]
`MCR p15, 0, rd, c7, c5, 1`	Flush ICache single entry	Address
`MCR p15, 0, rd, c7, c13, 1`	Prefetch ICache line	Address
`MCR p15, 0, rd, c7, c6, 0`	Flush DCache	SBZa
`MCR p15, 0, rd, c7, c6, 1`	Flush DCache single entry	Address
`MCR p15, 0, rd, c7, c10, 1`	Clean DCache entry	Address
`MCR p15, 0, rd, c7, c14, 1`	Clean and flush DCache entry	Address
`MCR p15, 0, rd, c7, c10, 2`	Clean DCache entry	Index/segment
`MCR p15, 0, rd, c7, c14, 2`	Clean and flush DCache entry	Index/segment

a. The value transferred in Rd should be zero.

The data format for index/segment operations is shown in Figure 2-2.

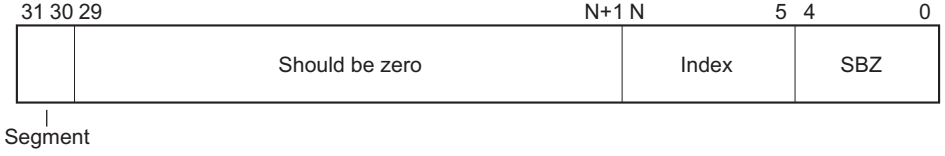


Figure 2-2 Index and segment format

The size of the index 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.

Table 2-20 Index fields for supported cache sizes

Cache size	Index
4KB	**Addr[9:5]**
8KB	**Addr[10:5]**
16KB	**Addr[11:5]**

Table 2-20 Index fields for supported cache sizes (continued)

32KB	**Addr[12:5]**
64KB	**Addr[13:5]**
128KB	**Addr[14:5]**
256KB	**Addr[15:5]**
512KB	**Addr[16:5]**
1MB	**Addr[17:5]**

For the ICache prefetch operation, the data format is shown in Figure 2-3.

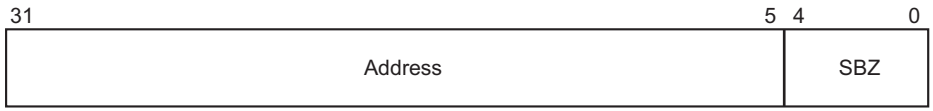


Figure 2-3 ICache address format

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 cp15, 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 allows the ARM946E-S to enter a low-power standby mode. When you invoke the operation, the **CLKEN** signal to the processor core is negated and the cache and tightly-coupled memories 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, ARM946E-S 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 core is *woken up* regardless of whether the interrupts are enabled or disabled (that is, independent of the I and F bits in the processor CPSR). The debug related *waking* only occurs if **DBGEN** is HIGH, that is, only when debug is enabled.

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

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 allow 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
```


The format of the register, rd, transferred during this operation is shown in Table 2-21.

Table 2-21 Lockdown register format

Register bit	Function
31	Load bit
30:2	UNP/SBZ
1:0	Cache segment

Lockdown is described in *Cache lockdown* on page 3-13.

2.3.12 Register 9, Tightly-coupled memory region registers

These registers allow you to modify the visible size of the tightly-coupled memories.

You can either increase or decrease the size of the tightly-coupled memories from the physical sizes described in register 0 (see *Register 0, Tightly-coupled memory size register* on page 2-9). Increasing the visible size of the tightly-coupled memories above the physical size allows aliasing within the tightly-coupled memory space. This feature is useful for debugging multitasking systems.

There is a memory region register for each of the tightly-coupled memories:

```
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
```

Each tightly-coupled memory region register has the format shown in Table 2-22.

Table 2-22 Protection region/base size register format

Register bit	Function
31:12	Region base
5:1	Area size Minimum size = 4KB Maximum size = 4GB (See Table 2-20 on page 2-22).
0	SBZ

For a given number of aliases for the physical memory size, the following function can be used:

Area size = Physical size + N

where 2^N is the required number of aliases.

The encodings for the supported tightly-coupled memory area sizes are shown in Table 2-23.

Table 2-23 Tightly-coupled memory area size encoding

Bit encoding	Tightly-coupled memory 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

Table 2-23 Tightly-coupled memory area size encoding (continued)

Bit encoding	Tightly-coupled memory area size
b10101	1GB
b10110	2GB
b10111	4GB

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

The instruction tightly-coupled memory base address is fixed at `0x00000`. For the instruction tightly-coupled memory, the region base returns the value `0x00000` when read.

When writing to the instruction tightly-coupled memory, 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 tightly-coupled memory region registers are cleared to `0x00000`.

At reset, the area size for the instruction and data tightly-coupled memory region registers takes the value defined in the tightly-coupled memory size register (see *Register 0, Tightly-coupled memory size register* on page 2-9).

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

Note

If the data tightly-coupled memory is located at the same address as the instruction tightly-coupled memory, then the instruction memory takes precedence for data accesses.

If the data tightly-coupled memory is located at the same address as the instruction tightly-coupled memory, and the instruction RAM is in load mode, data accesses read from the data RAM and write to the instruction RAM.

2.3.13 Register 13, Trace process identifier register

This register allows 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.

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

```
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.

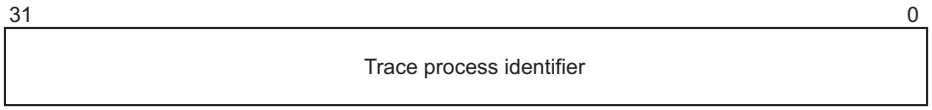


Figure 2-4 Process ID format

2.3.14 Register 15, RAM and TAG BIST test registers

Register 15 gives you access to the test features included within the ARM946E-S.

The register map for CP15 register 15 BIST-related instructions is shown in Table 2-24.

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`
RAM 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 lists 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 RAM BIST address register	`MRC p15, 1, rd, c15, c0, 2`	`MCR p15, 1, rd, c15, c0, 2`
Instruction RAM BIST general register	`MRC p15, 1, rd, c15, c0, 3`	`MCR p15, 1, rd, c15, c0, 3`
Data RAM BIST address register	`MRC p15, 1, rd, c15, c0, 6`	`MCR p15, 1, rd, c15, c0, 6`
Data RAM BIST general register	`MRC p15, 1, rd, c15, c0, 7`	`MCR p15, 1, rd, c15, c0, 7`
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

ARM recommends that you do not write application code that relies on the presence of the BIST address and general registers. ARM does not guarantee to support these registers in future versions of the ARM946E-S.

2.3.15 Register 15, Test state register

Register 15 gives you access to the test features included within the ARM946E-S. The register is accessed by:

```
MCR {cond} p15, 0, rd, c15, c0, 0; write test state register
MRC {cond} p15, 0, rd, c15, c0, 0; read test state register
```

The bit assignments of the test state access register are shown in Table 2-26.

Table 2-26 Test state register bit assignments

Bit	Function
31:13	Unpredictable
12	Disable DCache streaming
11	Disable ICache streaming
10	Disable DCache linefill
9	Disable ICache linefill
8:0	Reserved

Reading the test state register returns bits [12:0] in the least significant bits. The 19 most significant bits are 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 STRs, if they hit the cache, must update the memory and the cache, and that for LDRs or instruction prefetches that miss, a linefill is not performed. When set, bits [10:9] 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.

When set, bits [12:11] 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

Register 15 gives you access to the test features included within the ARM946E-S.

Additional instructions and operations are required to support debug operations within the cache. Instructions for the additional operations are listed in Table 2-27.

Table 2-27 Additional operations

Function	Data	Instruction
Write CP15 cache debug index register	Index/ segment	`MCR p15, 3, rd, c15, c0, 0`
Read CP15 cache debug index register	Index/ 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`
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 (CP15 r15), 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.

The format of the index/segment data is shown in Figure 2-5.

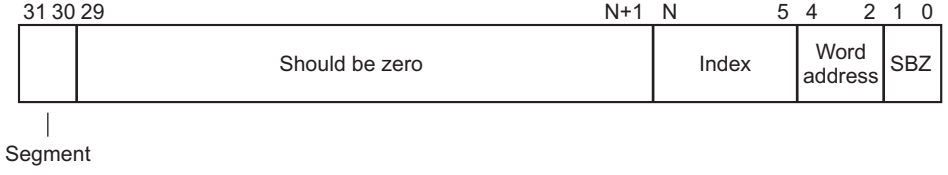


Figure 2-5 Index/segment format

The size of the index varies depending on the implemented cache size. Table 2-20 on page 2-22 shows how the index address field size changes for the cache sizes supported by the ARM946E-S.

Note

For TAG operations, the word address field in the cache debug register is ignored.

The data format for the TAG read/write operations is shown in Figure 2-6.

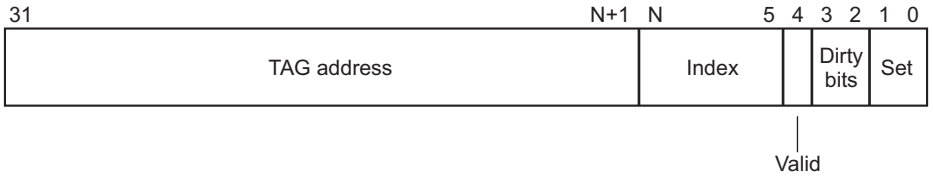


Figure 2-6 Data format TAG read/write operations

The size of the index and address TAGs vary depending on the implemented cache size. Table 2-28 shows how the index and TAG address field sizes change for the cache sizes supported by the ARM946E-S.

Table 2-28 Index fields for supported cache sizes

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

Chapter 3
Caches

To reduce the effective memory access time, the ARM946E-S uses a cache controller, an *Instruction Cache* (ICache), and a *Data Cache* (DCache). This chapter describes the features and behavior of each of these blocks. It contains the following sections:

3.1 Cache architecture

The ARM946E-S incorporates ICache and DCache. 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 ICache and DCache sizes independently.

The ICache and DCache are formed from synchronous SRAM, and have similar architectures. An example 8K cache is shown in Figure 3-1 on page 3-3.

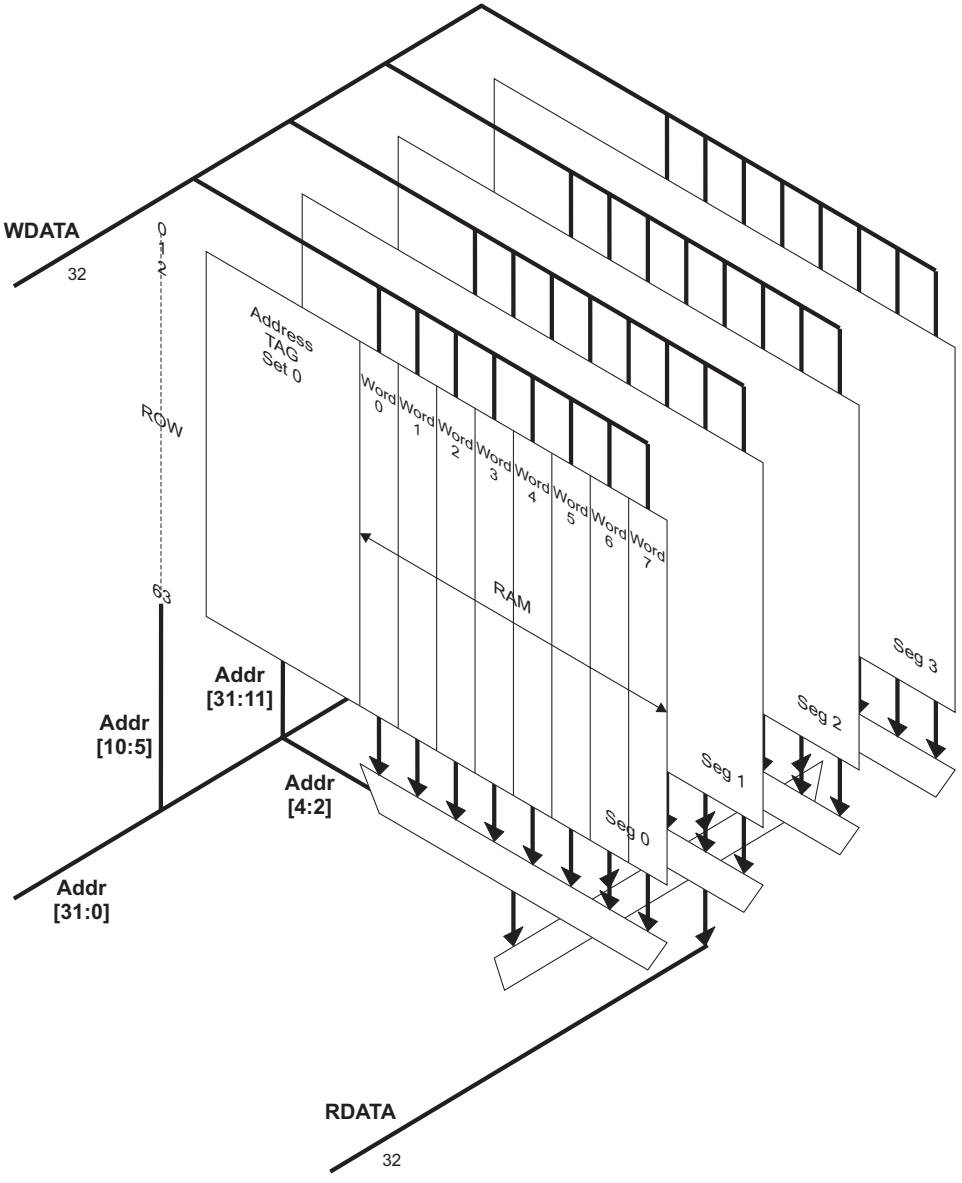


Figure 3-1 Example 8K cache

The ICache and DCache 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 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, unless the access is a buffered write when 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.

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
- address TAG.

The size of the index and address TAGs vary depending on the implemented cache size. Table 3-1 shows how the index and TAG sizes change for the cache sizes supported by the ARM946E-S.

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 4Kbyte cache.

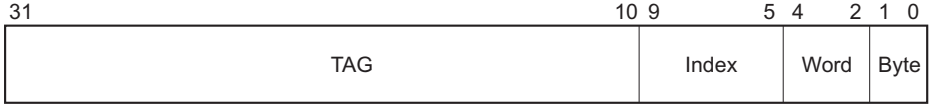


Figure 3-2 Access address for a 4KB cache

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 DCache and are used to indicate that a cache line contains data that differs from data stored at the address in external memory.

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

3.2 ICache

The ARM946E-S has a four-way set-associative ICache. You can choose the size of the ICache from any of the supported cache sizes. The ICache 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 ICache

You can enable the ICache 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 ICache is enabled, a cachable read-miss places lines in the ICache.

You can enable the ICache 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 ICache.

3.2.2 ICache operation

When enabled, the ICache 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 ICache 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 ICache by clearing bit 12 of the CP15 control register. This prevents all ICache look-ups and line fills, and forces all instruction fetches to be performed as single external accesses.

3.2.3 ICache validity

The ARM946E-S does not support external memory snooping. Therefore if you write self-modifying code, the instructions in the ICache 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 ICache.

You can flush the entire ICache by software in one operation, or you can flush individual cache lines by writing to the CP15 cache operations register (register 7). The ICache is automatically flushed during reset. The ICache never has to be cleaned because its only source of data is from external memory. (The ARM9E-S processor only performs reads from the ICache, except during debug operations.)

Flushing the entire cache

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

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

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, r0, c7, c5, 1; Flush single cache line
```

3.3 DCache

The ARM946E-S has a four-way set-associative DCache. You can choose the size of the DCache from any of the supported cache sizes. The DCache 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 DCache supports both *write back* (WB) and *write through* (WT) modes. For data stores that hit in the DCache, in WB 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 WT mode, a store that hits in the DCache causes the cache line to be updated but not masked as dirty, as the data store is also written to the write buffer to keep the external memory consistent. In both WB and WT 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 DCache, 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 DCache. For this reason the protection unit must be enabled when the DCache is enabled.

3.3.1 Enabling and disabling the DCache

You can enable the DCache 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.

You can enable the DCache 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 disable the DCache by clearing bit2 of the CP15 control register.

The DCache is automatically disabled and flushed on reset.

When the DCache is disabled, cache searches are prevented. This marks all data accesses as noncachable, forcing the ARM946E-S 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 DCache 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 DCache.

The Bd and Cd bits affect writes that both hit and miss in the DCache. If the Bd and Cd bits are both 1, the area of memory is marked as write back, and stores that hit in the DCache 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 DCache update both the cache and external memory.

3.3.3 DCache operation

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

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 a randomly chosen segment of the DCache.
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), allowing 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 DCache validity

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

The ARM946E-S 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 DCache clean and flush

The DCache has flexible cleaning and flushing utilities that allow the following operations:

- You can invalidate the whole DCache (*flush DCache*) in one operation without writing back dirty data.
- You can invalidate individual lines without writing back any dirty data (*flush DCache 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 DCache single entry*). You can clean cache lines using either their index within the DCache, or their address within memory.
- You can clean and flush individual lines in one operation (*clean and flush DCache entry*). You can clean and flush individual lines using either their index within the DCache, or their address within memory.

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

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

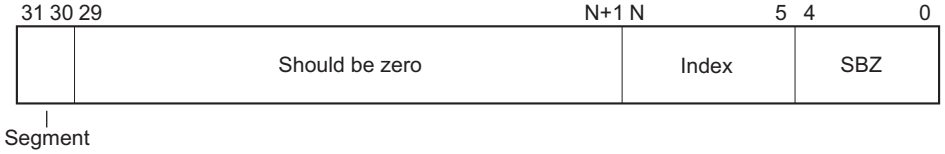


Figure 3-3 Register 7, Rd format

The value of N is dependent 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 equation in Figure 3-4:

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

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

Figure 3-4 Equation for calculating N

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 4Kbyte DCache).

```
    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
```


3.4 Cache lockdown

To provide predictable code behavior in embedded systems, a mechanism is provided for locking code into the ICache and DCache respectively. 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 in order to reduce external bus traffic.

You can lock down a region of the ICache or DCache 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 DCache using CP15 register 9. ICache lockdown uses both CP15 registers 7 and 9.

As described in *Cache architecture* on page 3-2, the ARM946E-S ICache and DCache each comprise four segments. You can perform lockdown with a granularity of one segment. The smallest space that you can lock down is one segment (one quarter of cache size). 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 ICache and DCache 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.

DCache lockdown

For the DCache, the procedure is as follows:

1. Write to CP15 register 9, setting DL=1 (DL is bit 31, the load bit) and Dindex=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=0 and Dindex=1.

If there is more data to lockdown, at the final step, the DL bit must be left HIGH and the process repeated. The DL bit must only be set LOW 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.

ICache lockdown

For the ICache, the procedure is as follows:

1. Write to CP15 register 9, setting IL=1 (the load bit) and Iindex=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 (ICache 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 ICache.
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=0 and Iindex=1.

If there are more instructions to lockdown, at the final step, the IL bit must be left HIGH and the process repeated. The IL bit must only be set LOW 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 DCache and ICache is that an MCR instruction must be used to force the linefill in the ICache, instead of an LDR. The rest of the sequence is the same as for DCache lockdown.

The MCR to perform the ICache fetch is a CP15 register 7 operation:

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

Example ICache lockdown subroutine

A subroutine that you can use to lock down code in the ICache is:

```
; Subroutine lock_i_cache
; r1 contains the start address
; r2 contains the end address
; Assumes that r2 - r1 fits within one cache set
; The subroutine performs a lockdown of instructions in the
; instruction cache
; It first reads the current lock_down index and then locks
; down the number of sets required
; Note - This subroutine must be located in a noncachable
;        region of memory
;      - Interrupts must be disabled
;      - Subroutine must be called using the BL instruction
;      - r1-r3 can be corrupted in line with ARM/Thumb
;        Procedure Call Standards (ATPCS)
;      - Returns final ICache lockdown index in r0 if successful
;      - Returns 0xFFFFFFFF in r0 if an error occurred

lock_I_cache
    BIC r1, r1, #0x7f           ;Align address to cache line
    MRC p15, 0, r3, c9, c0, 1   ;Get current ICache index
    AND r3, r3, #0x3            ;Mask unwanted bits
    CMP r3, #0x3                ;Check for available set
    BEQ error                   ;If no sets available,
                                ;generate an error
    ORR     r3, r3, #0x8000000  ;Set the lockdown bit
    MCR     p15, 0, r3, c9, c0, 1;Write lockdown register

lock_loop
    MCR p15, 0, r1, c7, c13, 1  ;Force an instruction fetch
                                ;from address r1
    ADD r1, r1, #0x20           ;Increment address by a
                                ;cache line length
    CMP r2, r1                  ;Reached our end address yet?
    BLT lock_loop               ;If not, repeat loop
    ADD r3, r3, #0x1            ;Increment ICache index
    BIC r0, r3, #0x8000000      ;Clear lockdown bit and
                                ;Write index into r0
```

```
    MCR p15, 0, r3, c9, c0, 1   ;Write lockdown register
    MOV pc, lr                  ;Return from subroutine

error
    MVN r0, #0                  ;Move 0xFFFFFFFF into r0
    MOV pc, lr                  ;Return from subroutine
```


Chapter 4
Protection Unit

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

- *About the protection unit* on page 4-2
- *Memory regions* on page 4-3
- *Enabling the protection unit* on page 4-2.

4.1 About the protection unit

The protection unit allows 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.

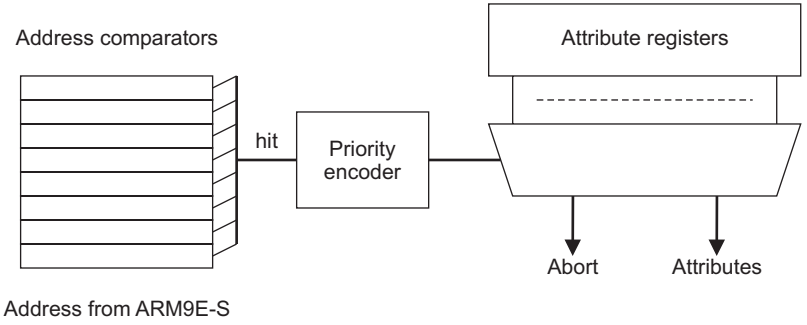


Figure 4-1 ARM946E-S protection unit

The protection unit is programmed using CP15 registers 1, 2, 3, 5, and 6 (see *CP15 register map summary* on page 2-4).

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 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 allows both data and instruction accesses.

The base address and size properties are programmed using CP15 register 6. The format for this is shown in Table 4-1.

Table 4-1 Protection register format

Register bit	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. The encoding is shown in Table 4-2.

Table 4-2 Region size encoding

Bit encoding	Area size
00000 to 01010	Reserved
01011	4KB
01100	8KB
01101	16KB
01110	32KB
01111	64KB
10000	128KB
10001	256KB
10010	512KB
10011	1MB
10100	2MB
10101	4MB
10110	8MB
10111	16MB
11000	32MB
11001	64MB
11010	128MB
11011	256MB
11100	512MB
11101	1GB
11110	2GB
11111	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 6-12.

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 Dap[3:0] = 0010. (Privileged mode full access, User mode read only.)

Region 1 Is programmed to be 16KB in size, starting from address `0x0000` with Dap[3:0] = 0001. (Privileged mode access only.)

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

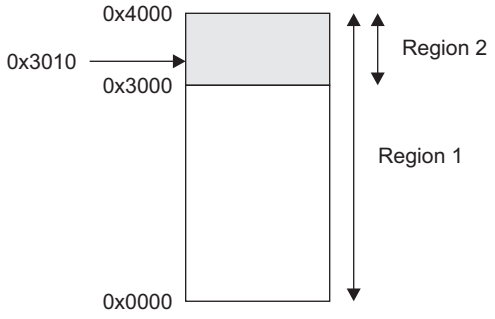


Figure 4-2 Overlapping memory regions

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 SRAM

This chapter describes the tightly-coupled SRAM in the ARM946E-S. 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 SRAM requirements

The ARM946E-S tightly-coupled SRAM is built from blocks of ASIC library compiled SRAM. The instruction SRAM (I-SRAM) and data SRAM (D-SRAM) can each be of any size supported by the protection unit, from 0 bytes to 1MB, although to ease implementation the size must be an integer power of two. The (I-SRAM) and (D-SRAM) can have different sizes.

ARM946E-S supports synchronous SRAM for the tightly-coupled RAM. The memory cells must be capable of returning data to the ARM9E-S core in a single cycle. This requirement applies to both the I-SRAM and D-SRAM.

To allow the I-SRAM 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 I-SRAM. This means that the ARM946E-S must multiplex the instruction and data addresses before entering the I-SRAM. It also means that the instruction data is routed to both the instruction and data interfaces of the core. See Figure 1-1 on page 1-4 for details of this data and address multiplexing.

Figure 5-1 shows a typical read cycle (I-SRAM shown).

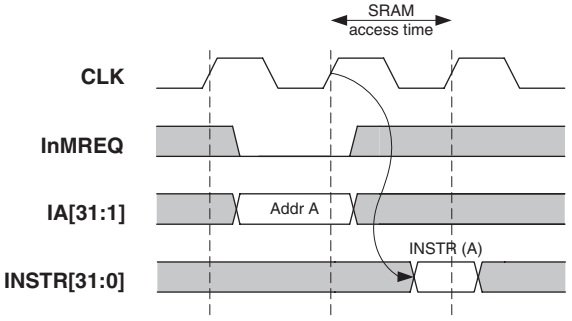


Figure 5-1 SRAM read cycle

The I-SRAM is located at address `0x00000000` in the memory map. This simplifies the implementation of the design by removing the need for complex address comparators on both the instruction and data interfaces of the ARM9E-S core to generate the chip select logic for the SRAM. Fixing the SRAM location at `0x0` allows an address decode to control the chip selects to give greater power efficiency.

5.2 Using CP15 control register

When out of reset, the behavior of the tightly-coupled SRAM is controlled by the state of CP15 control register.

5.2.1 Enabling the I-SRAM

You can enable the I-SRAM 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 I-SRAM, all future ARM9E-S instruction fetches and data accesses to the I-SRAM address space cause the I-SRAM to be accessed.

Enabling the I-SRAM greatly increases the performance of the ARM946E-S 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.

Note

You must take care to ensure that the I-SRAM 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 I-SRAM, the behavior is unpredictable.

5.2.2 Disabling the I-SRAM

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

Note

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

5.2.3 I-SRAM load mode

You must initialize the I-SRAM with the required code image before execution from the I-SRAM.

You can initialize the I-SRAM by writing to the memory from the AM9E-S core data interface.

The I-SRAM load mode allows this to be done in an efficient manner. Using the load mode allows you to copy from an address in the data cache or external memory into the same address within the I-SRAM.

The I-SRAM load mode bit of CP15 Register 1 inhibits reads from the I-SRAM, forcing reads from addresses that are within the I-SRAM address range to access either main memory, the data cache. Writes to addresses that are within the I-SRAM range are not affected by the *Instruction Load Mode* bit.

The procedure for initializing the I-SRAM using the load mode is as follows:

1. Enable the I-SRAM and instruction load mode
2. Load ARM registers from main memory, data cache or data RAM
3. Store ARM registers into I-SRAM
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 RAM and Load Mode
CopyLoop
    LDMIA R0, {R2 - R9}     ; Load 8 registers from main memory
    STMIA R0!, {R2 - R9}    ; Store 8 regs into instruction SRAM
    CMP R1, R0              ; Check if limit reached
    BGT CopyLoop            ; Repeat if more to do
```

`SWP` and `SWPB` operations to the instruction tightly-coupled memory while it is in load mode have unpredictable results. The read accesses external memory or the data cache, and the write updates the instruction tightly-coupled memory.

`SWP` and `SWPB` operations must not be performed to addresses in the instruction tightly-coupled SRAM space while it is in load mode.

5.2.4 Enabling the D-SRAM

You can enable the D-SRAM 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 D-SRAM, see *Register 9, Tightly-coupled memory region registers* on page 2-25, all future read and write accesses to the D-SRAM address space cause the D-SRAM to be accessed.

5.2.5 Disabling the D-SRAM

You can disable the D-SRAM by clearing bit 16 of the CP15 control register. When you have disabled the D-SRAM, see *Register 9, Tightly-coupled memory region registers* on page 2-25, all future reads and writes to the D-SRAM address space access the AHB. Read and write accesses to I-SRAM address space either use the I-SRAM or access the AHB depending on whether I-SRAM is enabled or not.

5.2.6 D-SRAM load mode

You must initialize the D-SRAM with the required data image before use.

You can initialize the D-SRAM by writing to the memory from the AM9E-S core data interface.

The D-SRAM load mode allows this to be done in an efficient manner. Using the load mode allows you to copy from an address in the data cache or external memory into the same address within the D-SRAM.

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

The procedure for initializing the D-SRAM using the load mode is as follows:

1. Enable the D-SRAM and data 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 load/store steps until the data image has been copied.

A suggested assembler code sequence for this procedure is:

```
    LDR R0, #ImageStart         ; Initialise 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 RAM and Load Mode
CopyLoop
    LDMIA R0, {R2 - R9}     ; Load 8 registers from main memory
    STMIA R0!, {R2 - R9}    ; Store 8 regs into instruction SRAM
    CMP R1, R0              ; Check if limit reached
    BGT CopyLoop            ; Repeat if more to do
```

SWP and SWPB operationsto the data tightly-coupled memory while it is in load mode have unpredictable results. The read accesses external memory or the data cache, and the write updates the data tightly-coupled memory.

SWP and SWPB operations must not be performed to addresses in the instruction tightly-coupled SRAM space while it is in load mode.

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 supports an *Advanced Microprocessor Bus Architecture* (AMBA) *Advanced High-performance Bus* (AHB) interface. The AHB is a new generation of AMBA interface that addresses the requirements of 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 SRAM and the external AHB memory. The AHB memory must be accessed for cache linefills and for initializing the tightly coupled memories, and to access code and data that are not within the cachable or tightly-coupled memory 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 allow 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 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.

6.2.1 About the AHB

The AHB architecture is based on separate cycles for address and data (rather than separate clock phases, as in ASB). 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 ARM946E-S transfer descriptions

The ARM946E-S only generates three of the possible transfer types defined in the *AMBA Specification*. These are:

IDLE **HTRANS[1:0]** = 00

NONSEQ **HTRANS[1:0]** = 10

SEQ **HTRANS[1:0]** = 11

The BUSY encoding (**HTRANS[1:0]** = 01) is not used by the ARM946E-S.

6.2.3 Burst sizes

The ARM946E-S supports the burst types listed in Table 6-1.

Table 6-1 Supported burst types

Burst type	HBURST encoding	Use
SINGLE	000	Single writes (STR/STRH/STRB) Uncached single reads Uncached instruction fetches
INCR	001	Store multiple (STM) Uncached burst reads (LDM)
INCR4	011	Dirty half-cache line write back
INCR8	101	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 is optimized to run with both the ICache and DCache enabled. If a memory request (either instruction or data) to a cachable area misses in the cache the ARM946E-S performs a linefetch.

A linefetch transfer is shown in Figure 6-1.

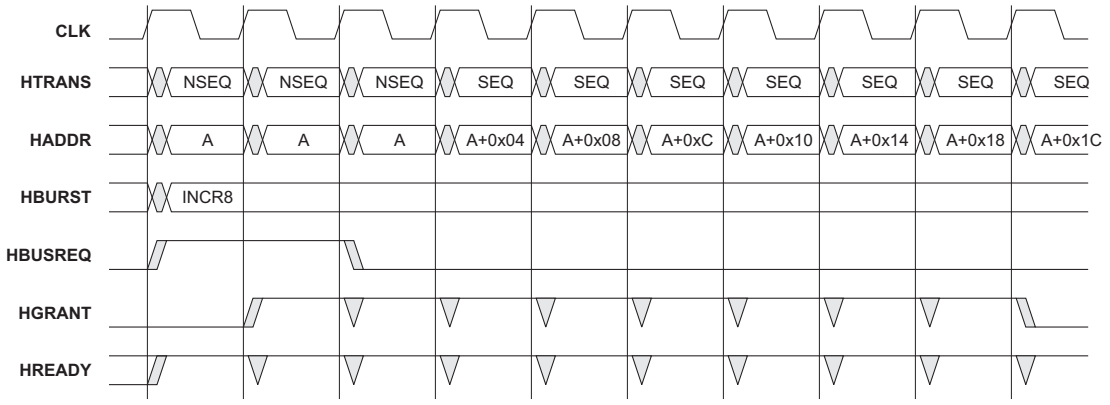


Figure 6-1 Linefetch transfer

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 asserts the bus request **HBUSREQ** until the arbiter grants the AHB bus (**HGRANT** asserted). The bus request is then negated. This allows optimum system performance as the arbiter can accurately predict the end of the defined length burst.

6.2.5 Back to back linefetches

The ARM946E-S supports streaming of data and instructions (core execution is advanced during the linefetch). To allow for cache look-ups when crossing a cache line boundary the ARM946E-S must insert IDLE cycles onto the AHB bus. The effect of this is shown in Figure 6-2. It is assumed in Figure 6-2 that **HGRANT** is asserted throughout, and that the **HCLK** frequency is the same as **CLK**.

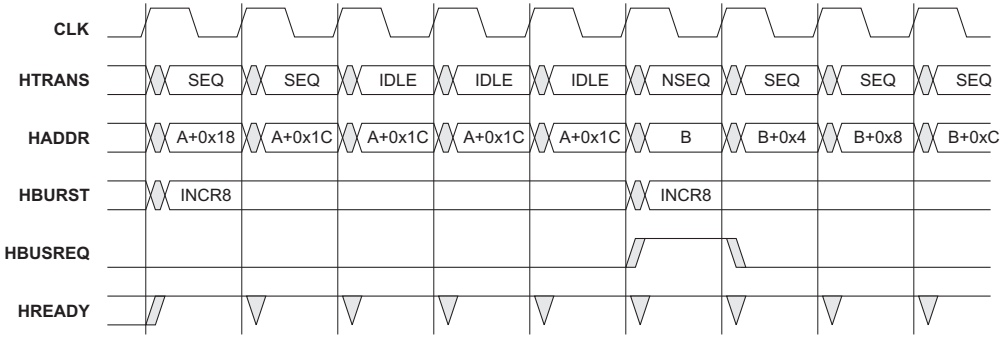


Figure 6-2 Back to back linefetches

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 6-6 shows uncached instruction fetches. Nonsequential uncached data operations exhibit similar bus timings.

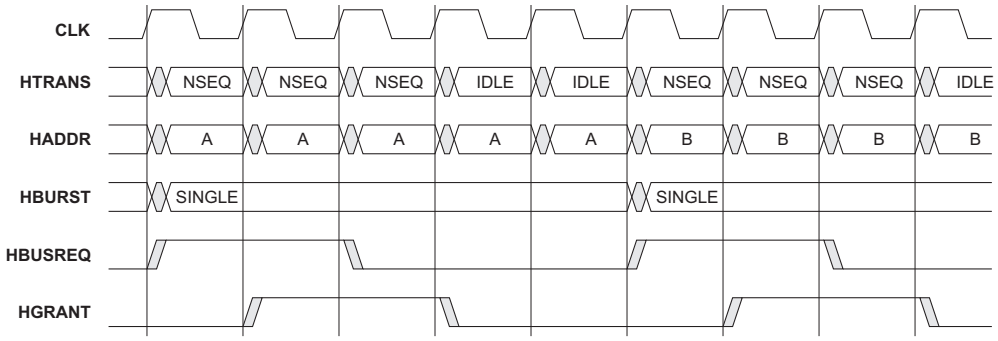


Figure 6-3 Nonsequential uncached accesses

6.2.7 Burst accesses

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

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

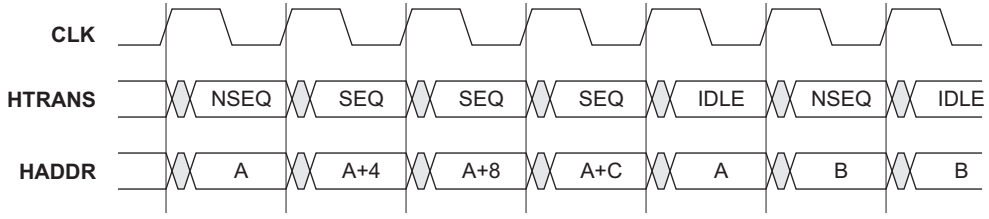


Figure 6-4 Data burst followed by instruction fetch

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 1 KB boundary. An example of this is shown in Figure 6-5 on page 6-7. The burst is restarted by inserting a nonsequential transfer as the boundary is crossed.

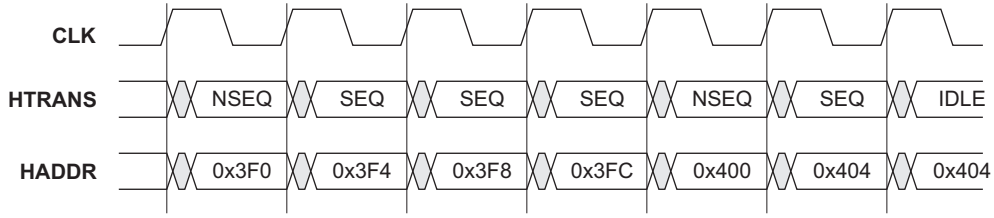


Figure 6-5 Crossing a 1KB boundary

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

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

When the ARM9E-S is running from tightly-coupled SRAM or performing writes using the write buffer, the ARM946E-S **HCLKEN** and **HREADY** inputs are not used to generate the **SYSCLKEN** core stall signal. The core is only stalled by SRAM 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. As 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 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 drives out the AHB address on the rising edge of **CLK** when the **HCLKEN** input is TRUE. 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 are timed

relative to **HCLK** but are sampled within the ARM946E-S with **CLK**. This leads to hold time issues, from **CLK** to **HCLK** on outputs, and from **HCLK** to **CLK** on inputs. In order to minimize this effect you must minimize the skew between **HCLK** and **CLK**.

Figure 6-6 shows the AHB clock relationships.

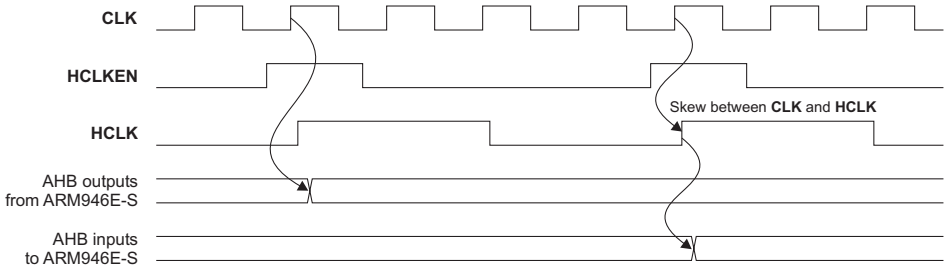


Figure 6-6 AHB clock relationships

Clock tree insertion at top level

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

Figure 6-7 on page 6-11 shows an example of an AHB slave connected to the ARM946E-S.

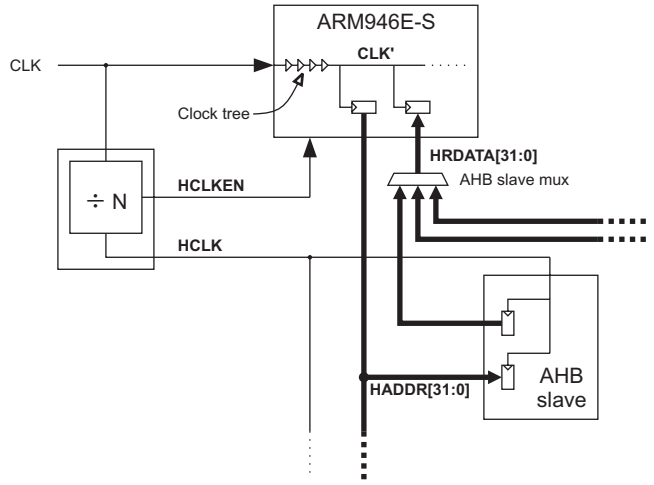


Figure 6-7 ARM946E-S CLK to AHB HCLK sampling

In Figure 6-7, the slave peripheral has an input setup and hold, and an output hold and valid time relative to **HCLK**. The ARM946E-S 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 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 input setup and hold requirements. As the ARM946E-S 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 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 DCache 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 DCache 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 DCache hits and/or misses, depending on the mode of operation. Table 6-2 shows how the Cd and Bd bits control the behavior of the write buffer.

Table 6-2 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 DCache hits for this type of access, there has been a programming error. DCache hits are ignored and the DCache line is not updated for a read.

Swap instructions operation on data in an NCB region are made to perform NCNB type accesses and are *not* buffered.

WT Searches the DCache for reads and writes. Reads that miss in the DCache cause a line fill. Reads that hit in the DCache do not perform an external access. All writes are buffered, regardless of whether they hit or miss in the DCache. Writes that hit in the DCache 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. DCache linefills cause the write buffer to drain before the linefill starts.

WB Searches the DCache for reads and writes. Reads that miss in the DCache cause a line fill. Reads that hit in the DCache do not perform an external access. Writes that miss in the DCache are buffered. Writes that hit in the DCache update the cache line, mark it as dirty, and do not send the data to the write buffer. DCache write-backs are buffered. Writes (write-miss and write-back) cannot be externally aborted. DCache 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

ARM946E-S fully supports the connection of on-chip coprocessors through an external coprocessor interface. All types of coprocessor instructions are supported. For a description of all the interface signals referred to in this chapter, see the *ARM9E-S Technical Reference Manual*.

Coprocessors determine the instructions they must execute using a *pipeline follower* in the coprocessor. As each instruction arrives from memory it enters both the ARM pipeline and the coprocessor pipeline. To avoid a critical path for the instruction being registered by the coprocessor, the coprocessor pipeline operates one clock cycle behind the ARM9E-S core pipeline. However, there is a mechanism inside ARM946E-S that stalls the ARM9E-S pipeline so the external coprocessor pipeline can catch up with the processor pipeline. So, practically, consider that the two pipelines are synchronized. The ARM9E-S core informs the coprocessor when instructions move from Decode into Execute, and whether the instruction has to be executed.

To enable coprocessors to continue executing coprocessor data operations while the ARM9E-S core pipeline is stalled (for example, when waiting for a cache linefill to occur), the coprocessor receives the clock **CLK**, and a clock enable signal **CPCLKEN**.

If **CPCLKEN** is LOW on the rising edge of **CPCLK** then the ARM9E-S core pipeline is stalled and the coprocessor pipeline must not advance. Figure 7-1 indicates the timing for these signals and when the coprocessor pipeline must advance its state.

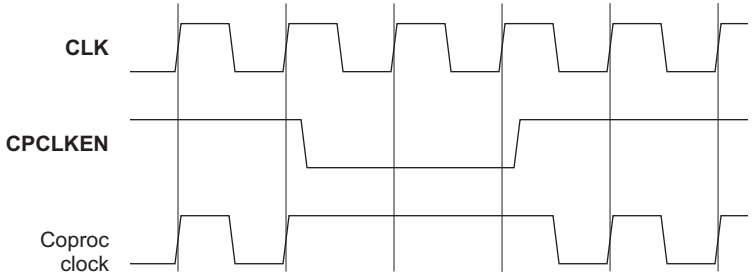


Figure 7-1 Coprocessor clocking

Coproc clock shows the result of ORing **CLK** with the inverse of **CPCLKEN. T**his is one technique for generating a clock that reflects the ARM9E-S core pipeline advancing.

7.1.1 Coprocessor instructions

There are three classes of coprocessor instructions:

`LDC/STC`	Load from memory to coprocessor, or store from coprocessor to memory.
`MCR/MRC`	Register transfer between coprocessor and ARM processor core.
`CDP`	Coprocessor data operation.

The following sections give examples of how a coprocessor must execute these instruction classes:

- *LDC/STC* on page 7-4
- *MCR/MRC* on page 7-8
- *Interlocked MCR* on page 7-10
- *CDP* on page 7-11
- *Privileged instructions* on page 7-12
- *Busy-waiting and interrupts* on page 7-13.

7.2 LDC/STC

The LDC and STC instructions are used respectively to transfer data to and from external coprocessor registers and memory. For the ARM946E-S, the memory can be either internal memory (cache or tightly-coupled memory) or AHB depending on the address range of the access and the protection unit settings.

The cycle timing for these operations is shown in Figure 7-2.

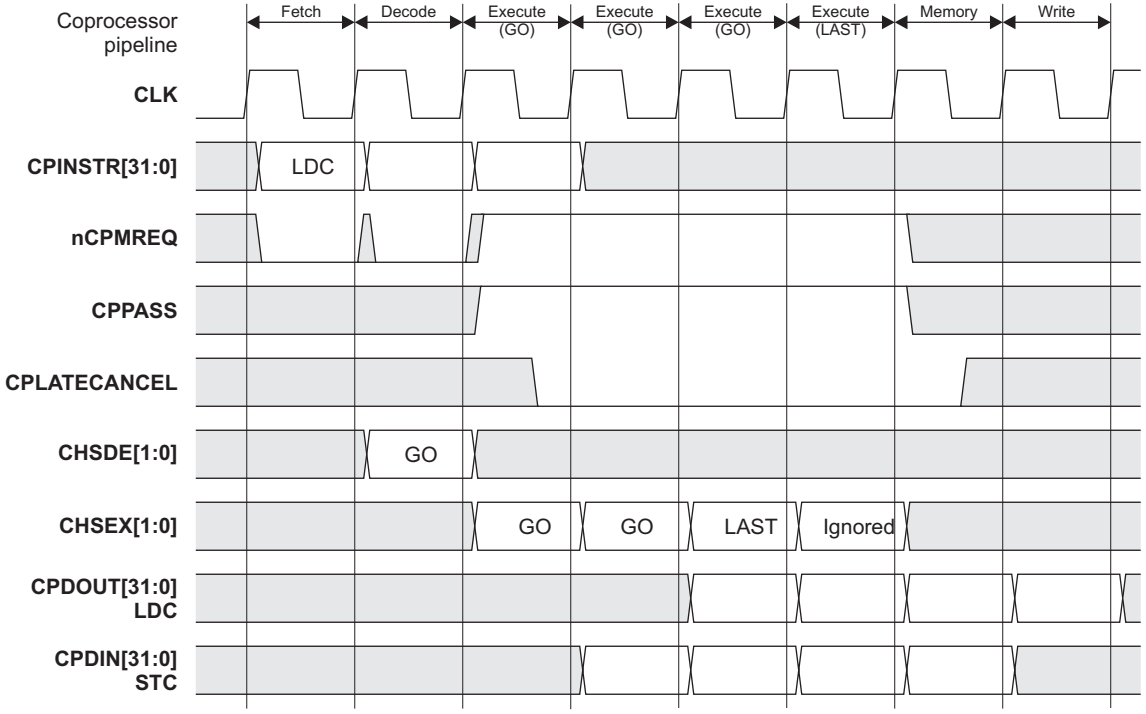


Figure 7-2 LDC/STC cycle timing

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 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 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:

1. The last instruction fetched enters the Decode stage of the coprocessor pipeline.
2. The instruction in the Decode stage of the coprocessor pipeline enters its Execute stage.
3. The fetched instruction is sampled.

In all other cases, the ARM9E-S pipeline is stalled, and the coprocessor pipeline does 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 causes a Data Abort. This is valid on the rising edge of **CLK** on the cycle that follows the first Execute cycle of the coprocessor instruction. This is the only cycle that **CPLATECANCEL** can be asserted in.

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]**.

7.2.1 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 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 below. 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 it has seen **CPPASS** HIGH, at the same time as 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 another cycle of execution. Both the ARM9E-S processor core and the coprocessor must also consider the state of the **CPPASS** signal before actually committing to the instruction. For an `LDC` or `STC` instruction, the coprocessor instruction drives the handshake signals with GO when two or more words still have to be transferred. When only one more word remains 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`. 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 7-2 on page 7-4.

LAST You can use an `LDC` or `STC` 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.

7.2.2 Coprocessor handshake encoding

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

Table 7-1 Handshake encoding

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

7.2.3 Multiple external coprocessors

If multiple external coprocessors are to be attached to the ARM946E-S interface, you can combine the handshaking signals by ANDing bit 1, and ORing bit 0. In the case of two coprocessors that have handshaking signals **CHSDE1**, **CHSEX1** and **CHSDE2**, **CHSEX2** respectively:

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

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

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

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

7.3 MCR/MRC

MCR/MRC cycles look very similar to STC/LDC. An example, with a busy-wait state, is shown in Figure 7-3.

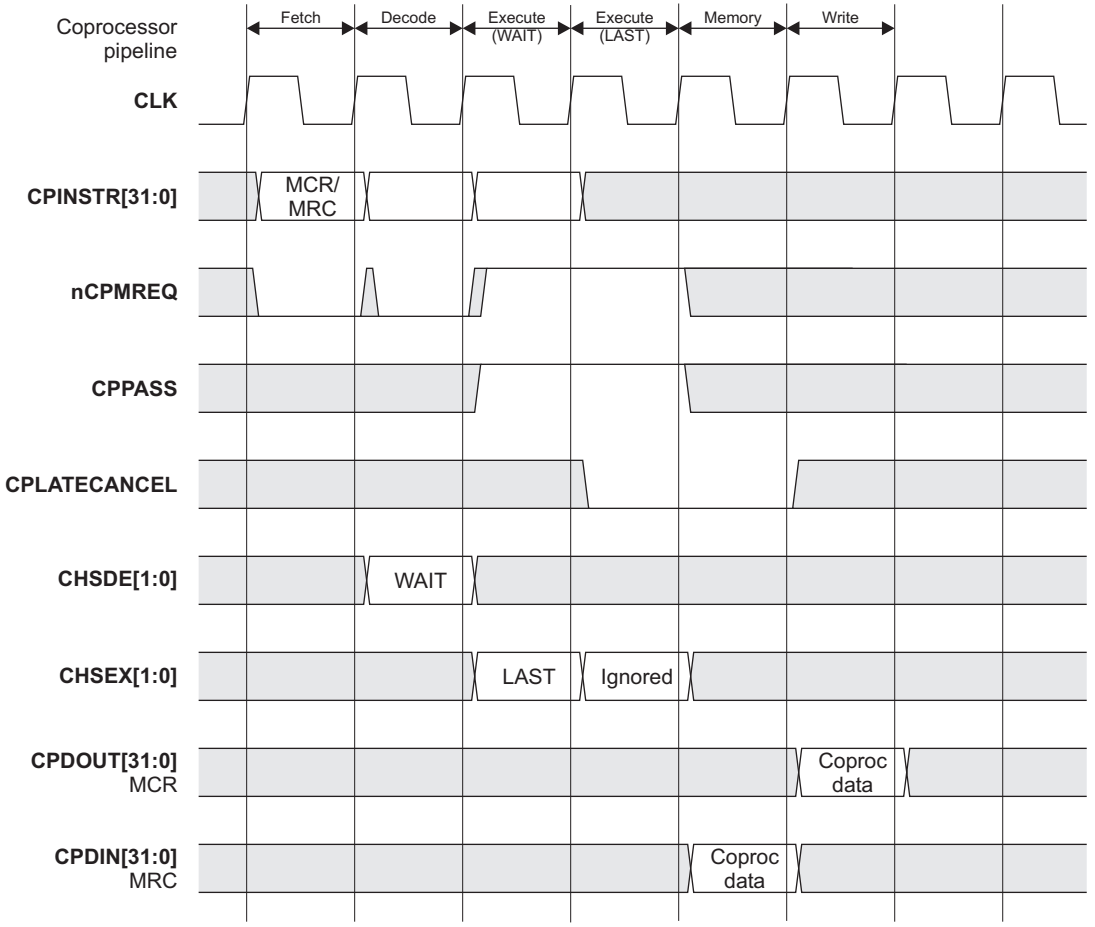


Figure 7-3 MCR/MRC transfer timing with busy-wait

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]** as required. 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 therefore, the instruction is to be executed, then the **CPPASS** signal is driven HIGH and the **CHSDE[1:0]** handshake bus is examined. It is ignored in all other cases.

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 an `MCR`, the **CPDOUT[31:0]** bus is driven with the registered data during the coprocessor Write stage. In the case of an `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.

7.4 Interlocked MCR

If the data for an MCR operation is not available inside the ARM9E-S core pipeline during its first Decode cycle, then the ARM9E-S core pipeline interlocks for one or more cycles until the data is available. An example of this is where the register being transferred is the destination 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 7-4 gives an example of an interlocked MCR that also has a busy-wait state.

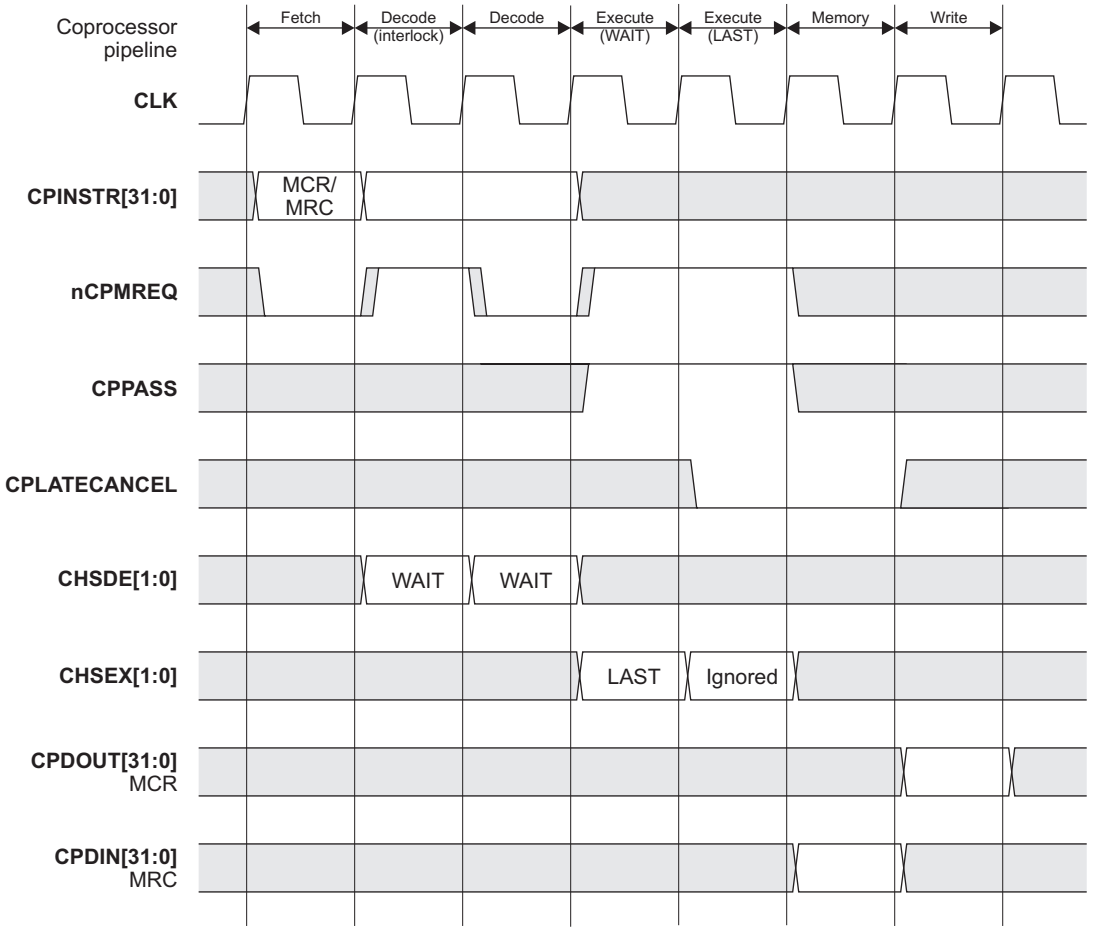


Figure 7-4 Interlocked MCR/MRC timing with busy-wait

7.5 CDP

CDP instructions normally execute in a single cycle. Like all the previous cycles, **nCPMREQ** is driven LOW to signal when an instruction is entering the Decode and then the Execute stage of the pipeline. If the instruction 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 7-5 shows a CDP cancelled because the previous instruction caused a Data Abort.

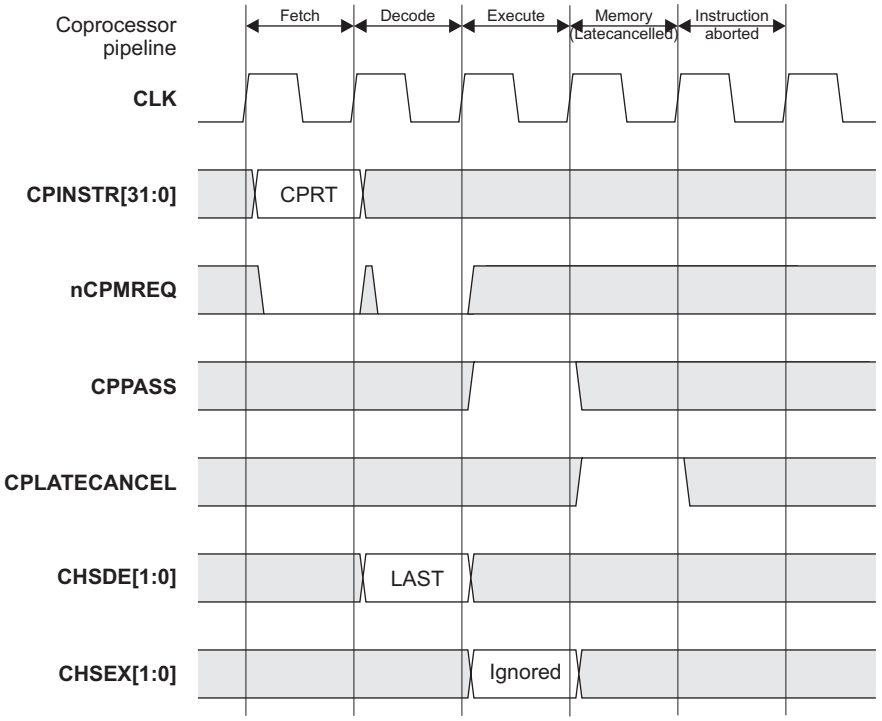


Figure 7-5 Late cancelled CDP

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 for it to cause no state changes to the coprocessor.

7.6 Privileged instructions

The coprocessor can restrict certain instructions for use in privileged modes only. To do this, the coprocessor tracks the **nCPTRANS** output. Figure 7-6 shows how **nCPTRANS** changes after a mode change.

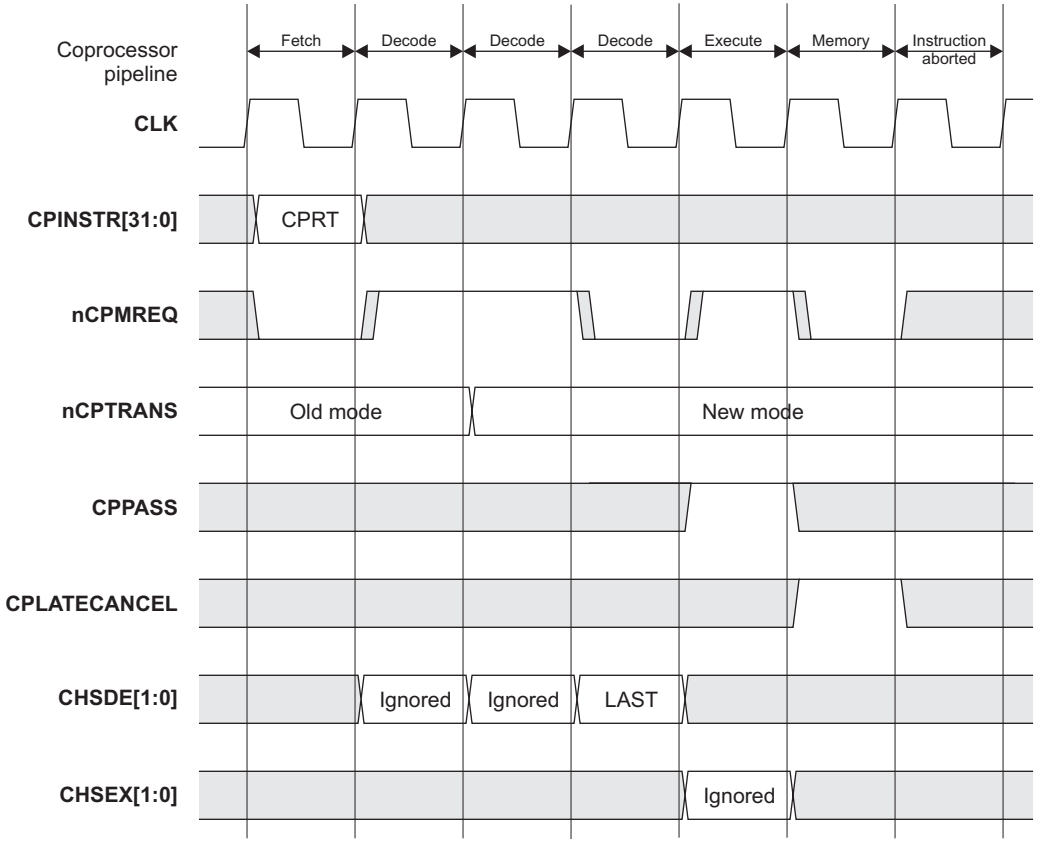


Figure 7-6 Privileged instructions

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

7.7 Busy-waiting and interrupts

The coprocessor is permitted to stall, or busy-wait, the processor during the execution of a coprocessor instruction if, for example, it is still busy with an earlier coprocessor instruction. To do so, the coprocessor associated with the Decode stage instruction drives WAIT onto **CHSDE[1:0]**. When the instruction concerned enters the Execute stage of the pipeline, the coprocessor can drive 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 can be interrupted while busy-waiting. This causes 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 7-7 shows a busy-waited coprocessor instruction abandoned due to an interrupt. **CPLATECANCEL** is also asserted as a result of the Execute interruption.

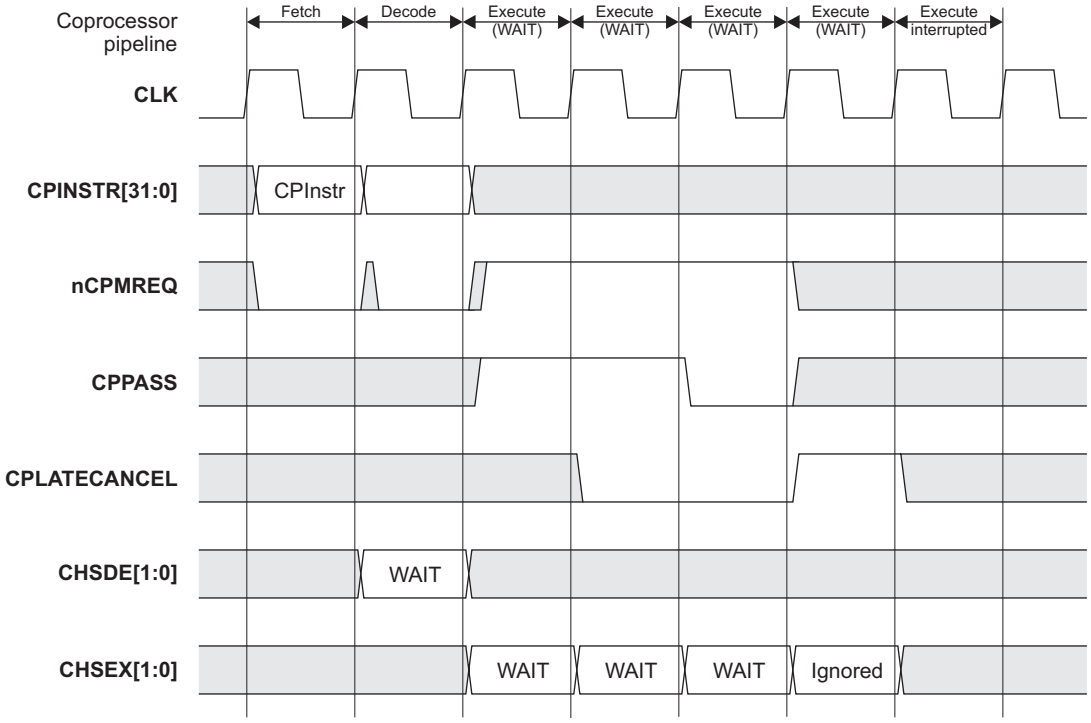


Figure 7-7 Busy-waiting and interrupts

Chapter 8
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:

8.1 About the debug interface

Debug support is implemented using the ARM9E-S core embedded within the ARM946E-S. The ARM946E-S debug interface is based on IEEE Std. 1149.1- 1990, *Standard Test Access Port and Boundary-Scan Architecture*. See this standard for an explanation of the terms used in this chapter and for a description of the TAP controller states.

The ARM9E-S processor core within the ARM946E-S contains hardware extensions for advanced debugging features. These make it easier to develop application software, operating systems, and the hardware itself.

The debug extensions allow you to force the core to be stopped by:

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

This is known as debug state. In debug state, the core and ARM946E-S memory system are effectively stopped, and isolated from the rest of the system. This is known as *halt mode* operation and allows you to examine the internal state of the ARM9E-S core, ARM946E-S system, 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 uses a JTAG-style interface, that allows 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.

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

When used in conjunction with a debug monitor program activated by the abort exception entry, you can debug the ARM946E-S while allowing the execution of critical interrupt service routines. The debug monitor program typically communicates with the debug host over the ARM946E-S debug communication channel. Real-time debug is described in *Real-time debug* on page 8-34.

8.1.1 Debug clocks

You must synchronize the system and test clocks externally to the ARM946E-S macrocell. 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 macrocell you must use a three-stage synchronizer. The off-chip device (for example,

Multi-ICE) issues a **TCK** signal, and waits for the **RTCK** (**R**eturned **TCK**) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next **TCK** until after **RTCK** is received.

Figure 8-1 shows this synchronization.

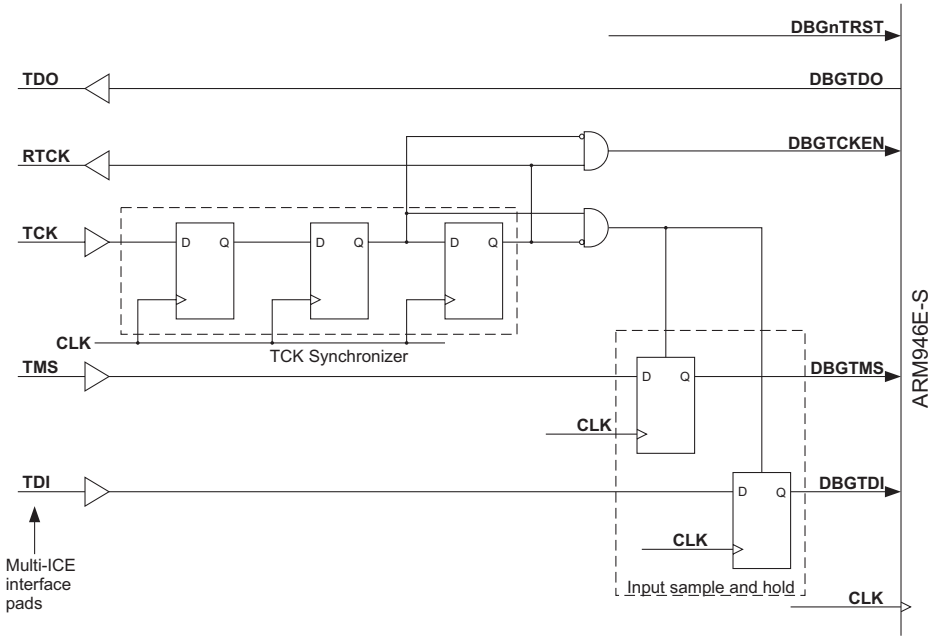


Figure 8-1 Clock synchronization

8.2 Debug systems

The ARM946E-S 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. Figure 8-2 shows a typical debug system.

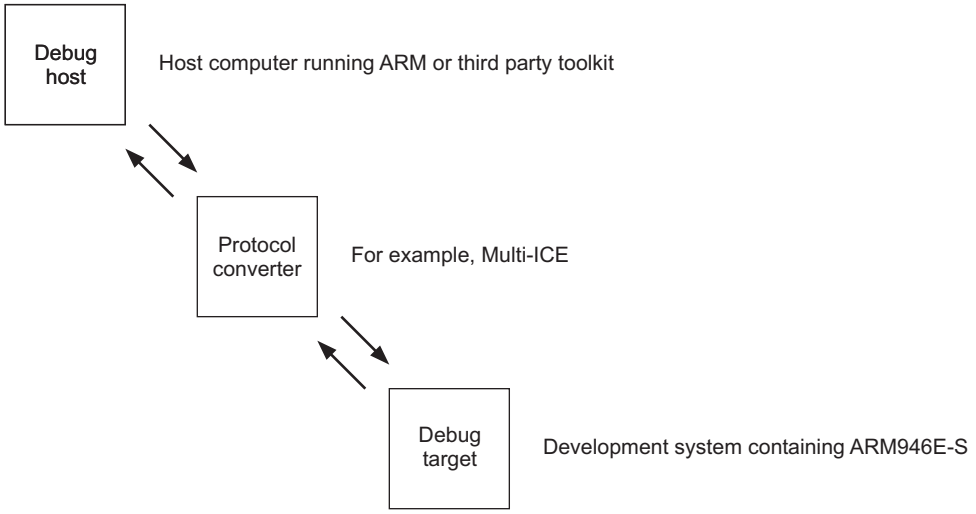


Figure 8-2 Typical debug system

A debug system typically has three parts:

- *The debug host*
- *The protocol converter*
- *ARM946E-S debug target* on page 8-5.

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

8.2.1 The debug host

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

8.2.2 The protocol converter

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

8.2.3 ARM946E-S debug target

The ARM9E-S core within the ARM946E-S has hardware extensions that ease debugging at the lowest level. The debug extensions:

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

The following major blocks of the ARM9E-S are shown in the *ARM9E-S block diagram*:

ARM9E-S CPU Core

With 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 8-27.

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

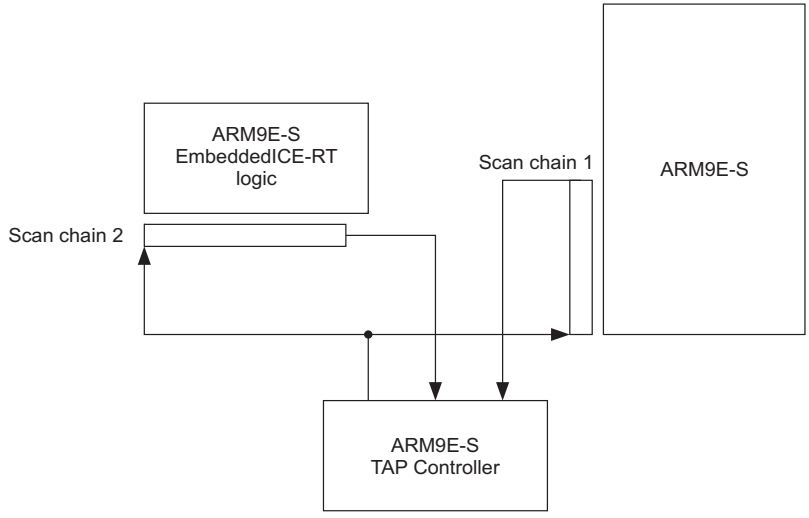


Figure 8-3 ARM9E-S block diagram

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

8.3 The JTAG state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure 8-4 on page 8-8 shows the state transitions that occur in the TAP controller.

The state numbers are also shown on the diagram. These are output from the ARM946E-S on the **TAPSM[3:0]** bits.

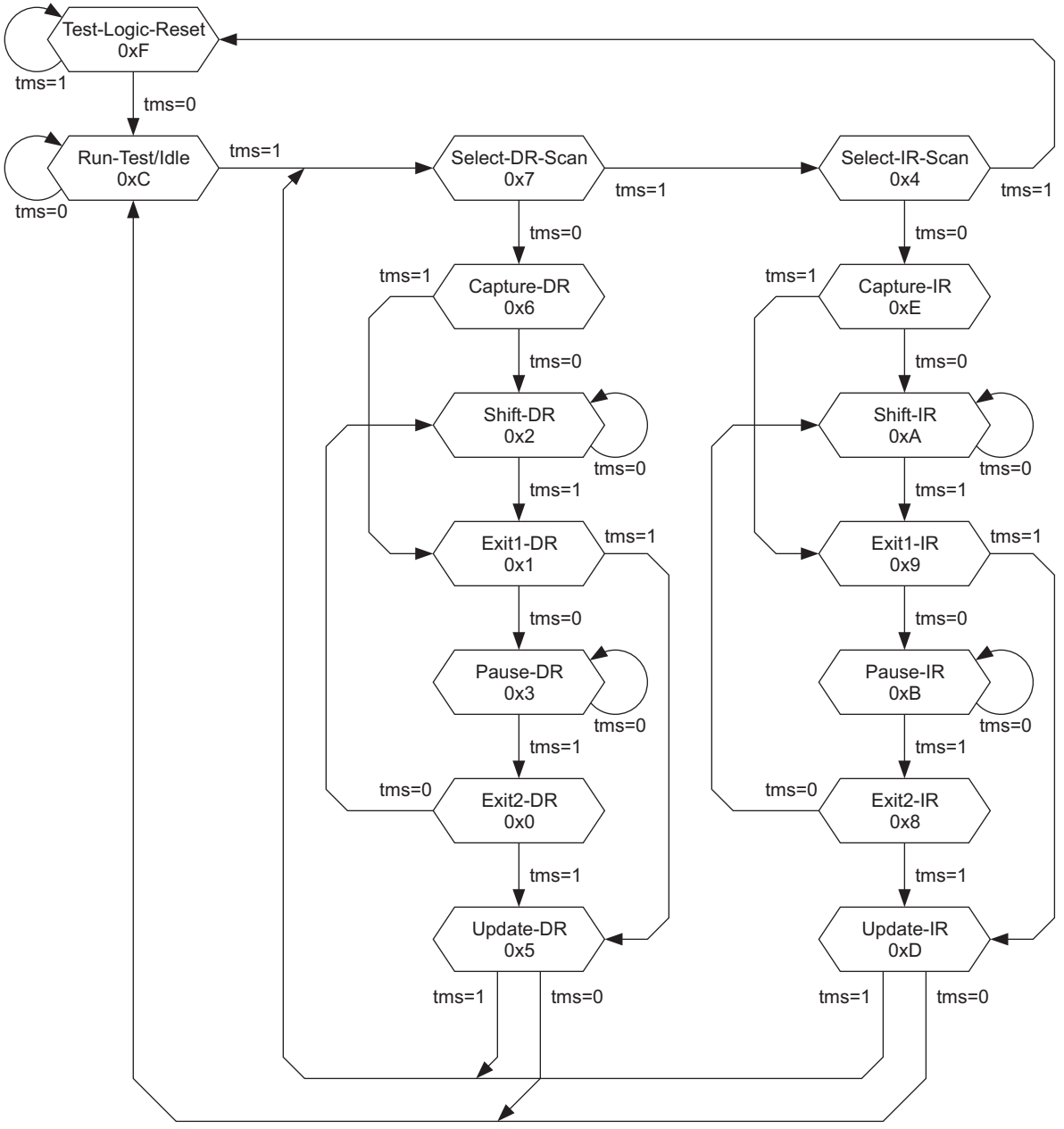


Figure 8-4 Test access port (TAP) controller state transitions[1]

1. From IEEE Std 1149.1-1990. Copyright 1999IEEE. All rights reserved.

8.3.1 Reset

The JTAG interface includes a state-machine controller (the TAP controller). To force the TAP controller into the correct state after power-up of 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. Before you can use the JTAG interface, you must drive **DBGnTRST** LOW, and then HIGH again. If you do not intend using the boundary scan 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**.

8.3.2 Pull-up resistors

The IEEE 1149.1 standard effectively requires **TDI** and **TMS** to have internal pull-up resistors. In order to minimize static current draw, these resistors are *not* fitted to the ARM9E-S core. Accordingly, the four inputs to the test interface (the **TDO**, **TDI**, and **TMS** signals plus **TCK**) must all be driven to valid logic levels to achieve normal circuit operation.

8.3.3 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 0001.

8.3.4 Public instructions

Table 8-1 lists the public instructions that are supported.

Table 8-1 Public instructions

Instruction	Binary code
EXTEST	0000
SCAN_N	0010
INTEST	1100
IDCODE	1110
BYPASS	1111
SAMPLE/PRELOAD	0011
RESTART	0100

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 (0000)

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 (0010)

This instruction connects the scan path select register between **TDI** and **TDO**.

During the CAPTURE-DR state, the fixed value 10000 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 (1100)

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** pin, while new test data is shifted in on the **TDI** pin.

IDCODE (1110)

The IDCODE instruction connects the device identification register (or ID register) between **TDI** and **TDO**. The ID register is a 32-bit register that allows 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** pin, while data is shifted in on the **TDI** pin into the ID register.

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

BYPASS (1111)

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 logic 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 (0011)

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** pin, while new data is shifted in on the **TDI** 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 (0100)

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.

8.4 Scan chains

ARM946E-S supports 32 scan chains. Three scan chains are used inside ARM946E-S. These allow testing, debugging, and programming of the EmbeddedICE macrocell watchpoint units.

The supported scan chains are listed in Table 8-2.

Table 8-2 ARM946E-S scan chain allocations

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

8.4.1 Scan chain 1

This scan chain is primarily used for debugging and provides access to the core instruction and data buses.

Scan chain 1 is 67 bits long and is made up of:

- 32 bits for data values
- 3 control bits
- 32 bits for instruction data.

These are arranged as shown in Table 8-3.

Table 8-3 Scan chain 1 bits

Bit	Function
67:35	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 has entered debug state, the first time SYSSPEED is captured and scanned out tells the debugger whether the core has entered debug state due to a breakpoint (SYSSPEED LOW) or a watchpoint (SYSSPEED HIGH). 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 HIGH, otherwise it is LOW.

8.4.2 Scan chain 2

Scan chain 2 allows access to the EmbeddedICE-RT logic registers.The order of the scan chain, from **DBGTDI** to **DBGTDO**, is:

- read/write
- register address bits 4:0
- data value bits 31:0.

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).

8.4.3 Scan chain 3

This scan chain allows ARM946E-S to control an optional external boundary scan chain. You can determine the length of scan chain 3.

8.4.4 Scan chain 15

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

The order of scan chain 15 from the **DBGTDI** input to the **DBGTDO** output is shown in Table 8-4.

Table 8-4 Scan chain 15 addressing mode bit order

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

The mapping of the CP15 register address field of scan chain 15 to CP15 registers is shown in Table 8-5.

Table 8-5 Mapping of scan chain 15 address field to CP15 registers

Address			Register		
[37]	[36:33]	[32]	Number	Name	Type
0	0000	0	C0.ID	ID register	Read
0	0000	1	C0.C	Cache type	Read
0	0001	0	C1	Control	Read/write
0	0010	0	C2.D	Data cachable bits	Read/write
0	0010	1	C2.I	Instruction cachable bits	Read/write
0	0011	0	C3	Write buffer control	Read/write
0	0100	0	C0.M	Tightly-coupled memory size	Read
0	0101	0	C5.D	Data space access permissions	Read/write

Table 8-5 Mapping of scan chain 15 address field to CP15 registers (continued)

Address			Register		
[37]	[36:33]	[32]	Number	Name	Type
0	0101	1	C5.I	Instruction address access permissions	Read/write
1	<Crm>[a]	0	C6.[7:0]	Memory region protection	Read/write
0	0111	0	C7.FD	Flush data cache	Write
0	0111	1	C7.FI	Flush instruction cache	Write
0	1110	0	C7.FD.s	Flush DCache single (uses C15.C.Ind)	Write
0	1110	1	C7.FI.s	Flush ICache single (uses C15.C.Ind)	Write
1	1010	1	C7.CD.s	Clean DCache single (uses C15.C.Ind)	Write
0	1001	0	C9.D	Data cache lock-down	Read/write
0	1001	1	C9.I	Instruction cache lock-down	Read/write
1	1000	1	C9.Dram	Data SRAM size/location	Read/write
1	1001	1	C9.Iram	Instruction SRAM size/location	Read/write
0	1101	1	C13.TPID	Trace process identifier	Read/write
0	1111	0	C15.State	Test state	Read/write
0	1111	1	C15.TAG	TAG BIST control	Read/write
1	1111	1	C15.RAM	Cache RAM BIST control	Read/write
1	1101	0	C15.C.Ind	Cache index (address/segment)	Read/write
0	1010	0	C15.DC	Data cache read/write (uses C15.C.Ind)	Read/write
0	1010	1	C15.IC	Instruction cache read/write (uses C15.C.Ind)	Read/write

Table 8-5 Mapping of scan chain 15 address field to CP15 registers (continued)

Address			Register		
[37]	[36:33]	[32]	Number	Name	Type
0	1011	0	C15.DT	Data tag read/write (uses C15.C.Ind)	Read/write
0	1011	1	C15.IT	Instruction tag read/write (uses C15.C.Ind)	Read/write
1	1110	1	C15.Mem	Memory RAM BIST control	Read/write

a. For CP15 register 6, CRm corresponds to the region number (0 to 7).

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.

8.5 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.

8.5.1 Step 1

This is done by reading the ICache and DCache TAG arrays using scan chain 15. The debugger must do this for each entry set within the cache. The format of the data returned is shown in Figure 8-5.

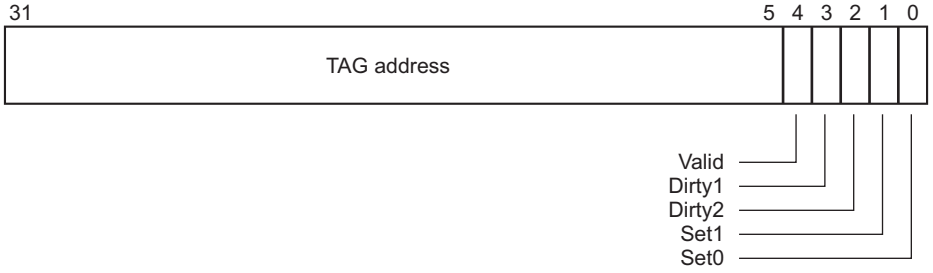


Figure 8-5 TAG address format

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.

8.5.2 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 DCache, the debugger can execute system speed accesses that hit in the cache and, therefore, return the cache contents. Writes to the DCache 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 DCache, 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. The format of the cache index register is shown in Figure 8-6.

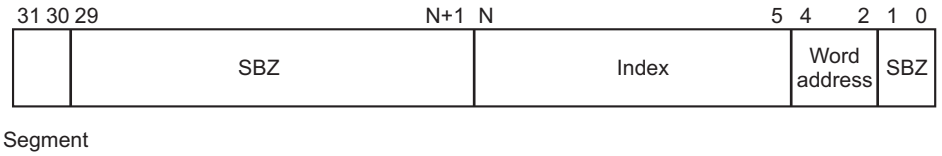


Figure 8-6 Cache index register format

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 ICache has been updated in this way. The debugger must restore the original cache contents after executing the breakpoint.

8.6 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 to enter debug state.
- **DBGACK** is used by the ARM946E-S to flag back to the system that it is in debug state.

8.6.1 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. This is shown in Figure 8-7.

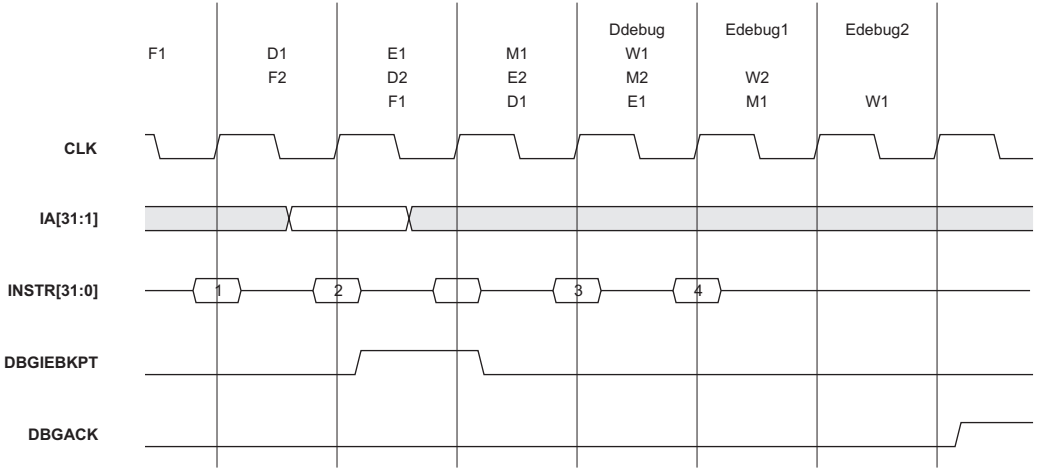


Figure 8-7 Breakpoint timing

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 ORed with the internally-generated **Breakpoint** signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external breakpoints are possible.

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

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

8.6.2 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. (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. This means that the previously breakpointed instruction is fetched again, and if the breakpoint is still set, the processor enters debug state when it 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 stop-mode debug state, interrupts are disabled, although the state of the I and F bits in the *Program Status Register (PSR)* are not affected.

8.6.3 Watchpoints

Entry into debug state following a watchpointed memory access is imprecise. This is necessary 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 ORed with the internally-generated **Watchpoint** signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external watchpoints are possible.

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

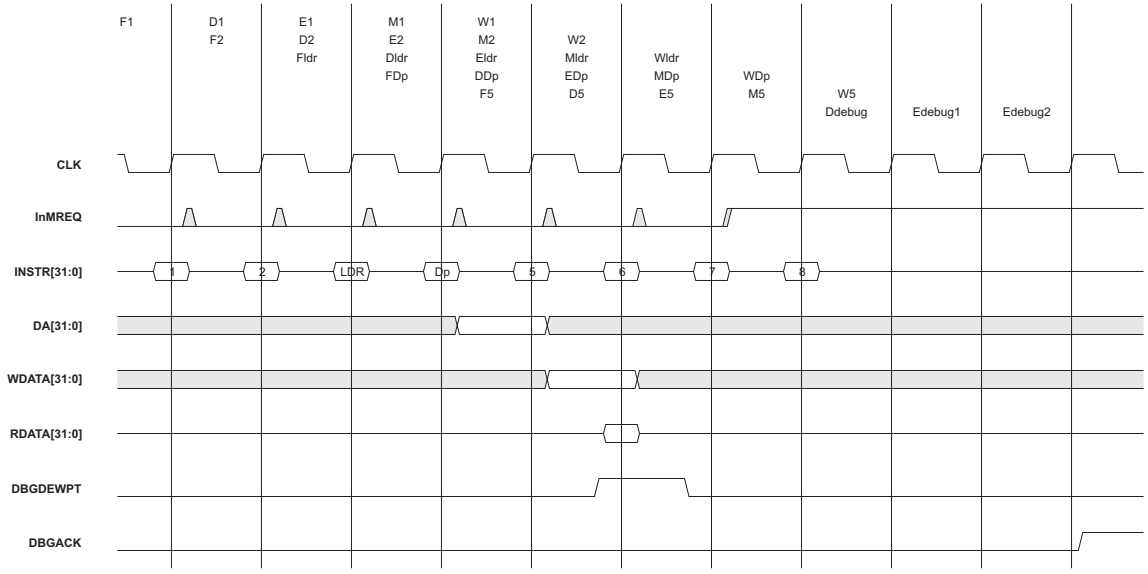


Figure 8-8 Watchpoint entry with data processing instruction

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. However, you can always restart the processor. A timing diagram showing debug entry after a watchpoint where the next instruction is a branch is shown in *Watchpoint entry with branch* on page 8-23.

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.

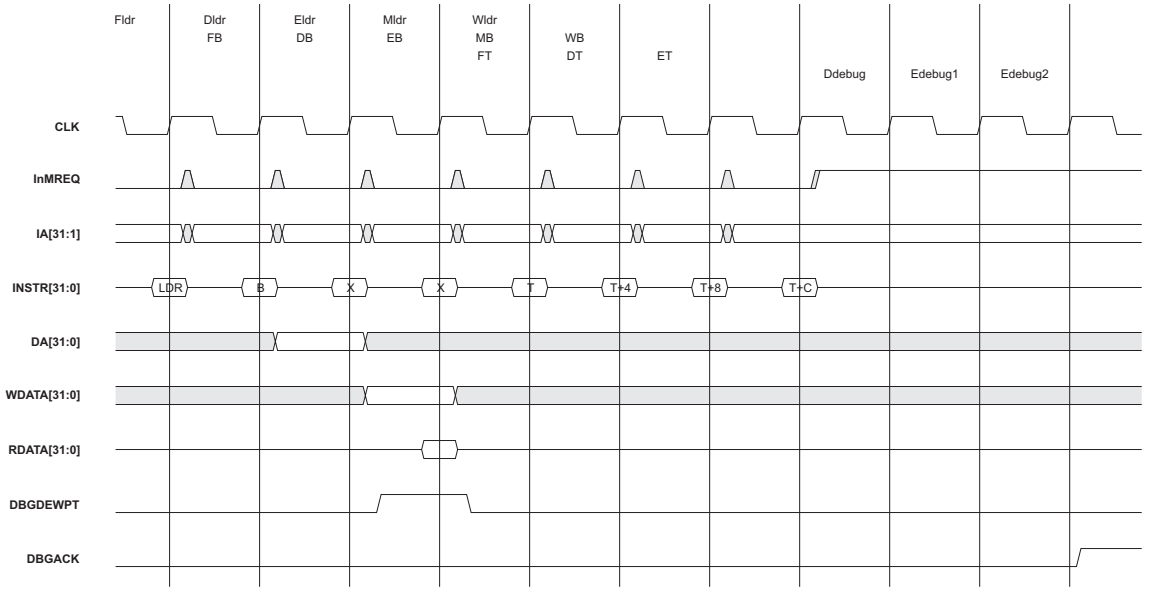


Figure 8-9 Watchpoint entry with branch

8.6.4 Watchpoints and exceptions

If a watchpointed data access is also abort, the watchpoint condition is registered and the exception entry sequence performed. Then the processor enters debug state. If there is an interrupt pending, the ARM9E-S allows the exception entry sequence to occur and then enters debug state.

8.6.5 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. Debugrequest 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.

8.6.6 Actions of the ARM9E-S in debug state

When the ARM9E-S is in debug state, both memory interfaces indicate internal cycles. This ensures that the tightly-coupled SRAM within the ARM946E-S, and the AHB interface, are both quiescent, allowing the rest of the AHB system to ignore the ARM9E-S and function as normal. Because the rest of the system continues operation, the ARM9E-S ignores aborts and interrupts.

The **nRESET** signal must be held stable during debug. If the system applies reset to the ARM946E-S (**nRESET** is driven LOW), the state of the ARM9E-S changes without the knowledge of the debugger.

8.7 ARM9E-S core clock domains

The ARM9E-S has a single clock, **CLK**, that is qualified by two clock enables:

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

During normal operation, **SYSCLKEN** conditions **CLK** to clock the core. When the ARM946E-S is in debug state, **DBGTCKEN** conditions **CLK** to clock the core.

8.8 Determining the core and system state

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

Before you can examine the core and system state, the debugger must determine 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 HIGH, the core has entered debug from Thumb state.

8.9 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.

EmbeddedICE-RT is programmed serially using the ARM9E-S TAP controller. Figure 8-10 illustrates the relationship between the core, EmbeddedICE-RT, and the TAP controller, showing only the signals that are pertinent to EmbeddedICE-RT.

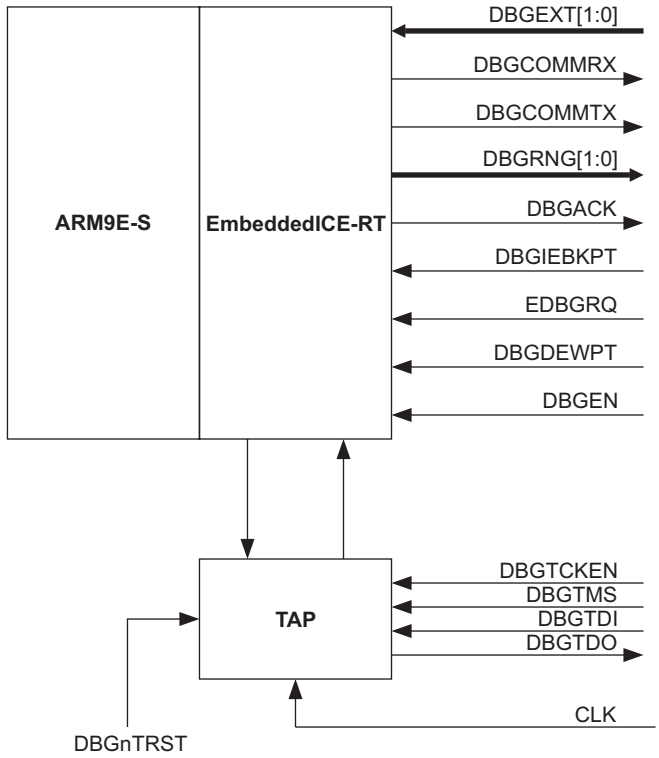


Figure 8-10 The ARM9E-S, TAP controller, and EmbeddedICE-RT

The EmbeddedICE-RT logic comprises:

- two real-time watchpoint units
- two independent registers:
  - — the debug control register
  - — the debug status register
- debug comms 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.

8.10 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 is always LOW.

8.11 The debug communications channel

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

The communications channel comprises:

- a 32-bit comms data read register
- a 32-bit wide comms data write register
- a 6-bit wide comms control register for synchronized handshaking between the processor and the asynchronous debugger.

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

In addition to the comms channel registers, the processor can access a 1-bit debug status register for use in the real-time debug configuration.

8.11.1 Debug comms channel registers

CP14 contains 4 registers. These have the register allocations listed in Table 8-6.

Table 8-6 Coprocessor 14 register map

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

8.11.2 Debug comms channel status register

The debug comms channel status register is read-only. It controls synchronized handshaking between the processor and the debugger. The debug comms channel status register is shown in Figure 8-11 on page 8-31.

Figure 8-11 Debug comms 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 0011).

Bits 27:2 Are reserved.

Bit 1 Denotes whether the comms data write register is available (from the point of view of the processor). If, from the point of view of the processor, the comms data write register is free (W=0), new data can be written. If the register is not free (W=1), the processor must poll until W=0. From the point of view of the debugger, when W=1, some new data has been written that can then be scanned out.

Bit 0 Denotes whether there is new data in the comms data read register. If, from the point of view of the processor, R=1, there is new data that can be read using an `MRC` instruction. From the point of view of the debugger, if R=0, the comms data read register is free, and new data can be placed there through the scan chain. If R=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.

You are recommended to use the following instructions:

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

This returns the debug comms control register into Rd.

`MCR p14, 0, Rn, cl, c0`

This writes the value in Rn to the comms data write register.

`MRC p14, 0, Rd, cl, 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.

8.11.3 Debug status register

A debug monitor can use the coprocessor 14 debug status register when the ARM9E-S is configured into real-time debug mode.

The coprocessor 14 debug status register is a 1-bit wide read/write register with the format shown in Figure 8-12.

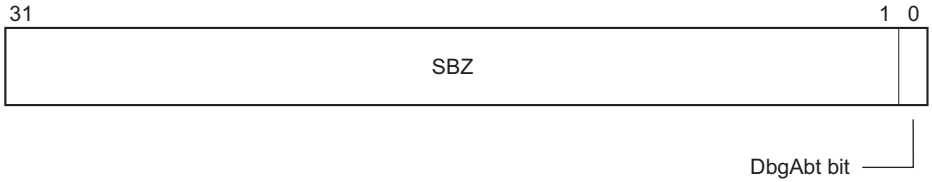


Figure 8-12 Coprocessor 14 debug status register format

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

A typical use of this bit is by a real-time debug aware abort handler. This examines the DbgAbt bit to determine whether the abort has been externally or internally generated. If the DbgAbt bit is set, the abort handler initiates communication with the debugger over the comms channel.

8.11.4 Communications using the comms channel

You can send and receive messages using the comms channel.

Sending a message to the debugger

When the processor has to send a message to the debugger, it must check the comms data write register is free for use by finding out whether the W bit of the debug comms control register is clear.

The processor reads the debug comms control register to check the status of the W bit:

- If the W bit is clear, the comms 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 is written by a register transfer to coprocessor 14. As the data transfer occurs from the processor to the comms data write register, the W bit is set in the debug comms control register.

The debugger sees both the R and W bits when it polls the debug comms control register through the JTAG interface. When the debugger sees that the W bit is set, it can read the comms data write register, and scan the data out. The action of reading this data register clears the debug comms control register W bit. At this point, the communications process can begin again.

Receiving a message from the debugger

Transferring a message from the debugger to the processor is similar to sending a message to the debugger. In this case, the debugger polls the R bit of the debug comms control register.

- if the R bit is LOW, the comms 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 comms data read register is free, data is written there using the JTAG interface. The action of this write sets the R bit in the debug comms control register.

The processor polls the debug comms control register. If the R bit is set, there is data that can be read using an `MRC` instruction to coprocessor 14. The action of this load clears the R bit in the debug comms control 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.

8.12 Real-time debug

The ARM9E-S within ARM946E-S contains logic that allows you to debug a system without stopping the core entirely. This allows 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/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 real-time 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 (**DBGEXTERN**)
  - User/Privileged mode access (**DnTRANS/InTRANS**)
  - read/write access (watchpoints)
  - access size (breakpoints: **ITBIT**, watchpoints: **DMAS[1:0]**).
- 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.
- No support is provided to mix halt mode/monitor mode debug functionality. When the core is configured into the monitor mode, asserting the external **EDBGRQ** signal results in unpredictable behavior. Setting the internal **EDBGRQ** bit results in unpredictable behavior.

When an abort is generated by the monitor mode, it is recorded in the debug status register in coprocessor 14 (see *Debug status register* on page 8-32).

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

If false matches can occur during changes to the watchpoint registers, caused by old data in some registers and new data in others, then you must:

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

8.12.1 Further reading - debug in depth

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

Chapter 9
ETM Interface

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

9.1 About the ETM interface

The ARM946E-S 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 consists of two parts:

A trace port A trace protocol has been developed to provide a real-time trace capability for processor cores that are deeply embedded in much larger ASIC designs. As the ASIC typically includes significant amounts of on-chip memory, you cannot determine how the processor core is operating simply by observing the pins of the ASIC. A trace port is required to confirm the performance of the processor in operational use.

Triggering facilities

An extensible specification exists, allowing you to specify the exact set of trigger resources required for a particular application. Resources include address and data comparators, counters, and sequencers.

The ETM compresses the trace information and exports it through the trace port. An external *Trace Port Analyzer* (TPA) is used to capture the trace information.

The ETM interface is primarily *one way*. To provide code tracing, the ETM block must be able to monitor various ARM9E-S inputs and outputs. The required ARM9E-S inputs and outputs are collected and driven out from the ARM946E-S as the ETM interface.

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 signals. The latency of the pipelined outputs does not affect ETM trace behavior, because all outputs are delayed by the same amount.

Figure 9-1 on page 9-3 shows the ARM946E-S ETM interface.

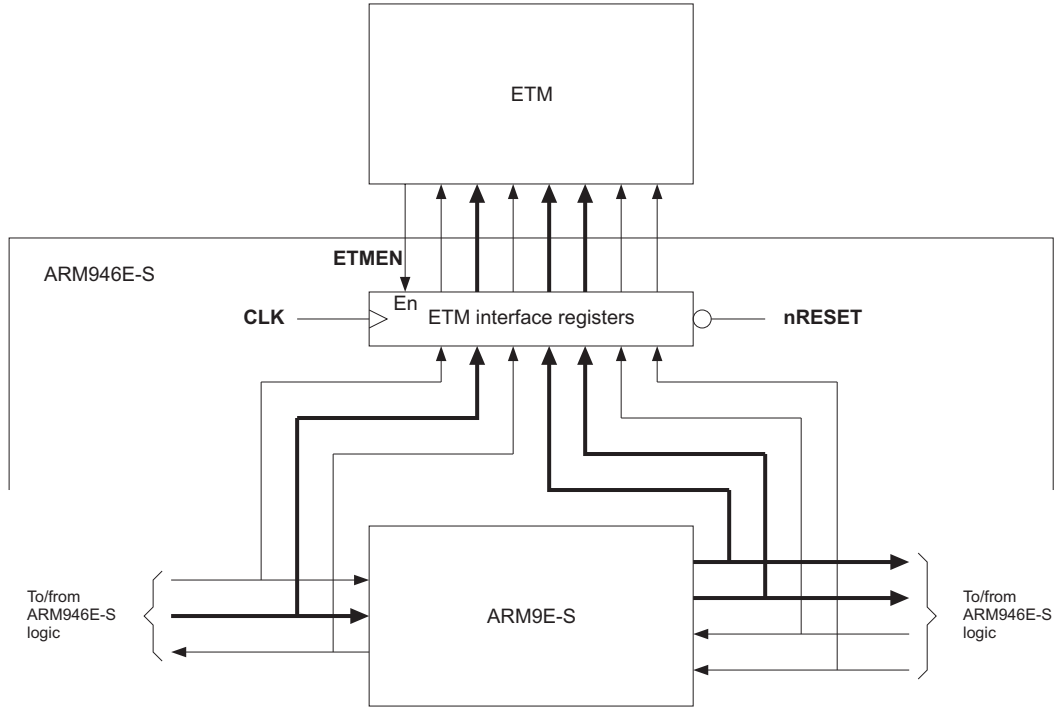


Figure 9-1 ARM946E-S ETM interface

9.2 Enabling the ETM interface

The only input to the ETM interface of the ARM946E-S is an enable signal that determines whether the required ARM9E-S inputs and outputs are driven out from the ARM946E-S.

The ETM enable is controlled 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 is disabled. At reset, all ETM interface outputs are reset LOW.

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

Note

If you do not use an ETM in an embedded ARM946E-S design, you must tie the **ETMEN** input LOW to save power.

Chapter 10
Test Support

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

10.1 About the ARM946E-S 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 control logic as part of the synthesis flow. You can use BIST to provide high fault coverage of the compiled SRAM.

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, when 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, due to 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, ARM946E-S includes a dual-purpose INTEST scan chain wrapper. This facilitates ATPG and provides an additional method for activating BIST of the SRAM.

ATPG

You can use the INTEST scan chain to enable an ATPG tool to access the ARM946E-S top-level inputs and outputs in an embedded design. This wrapper adds a scan source for each ARM946E-S 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 ARM946E-S **SERIALEN** input is true, this scan chain scans in serialized MCR instructions to initiate BIST test using 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 allows the full range of BIST tests 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.

10.3 BIST of memory arrays

Adding a simple memory test controller allows you to perform an exhaustive test of the memory arrays. You can activate the BIST test using an MCR to the CP15 BIST control register.

When you perform a BIST test on an SRAM, the functional enable for that SRAM is automatically disabled, forcing all memory accesses to that SRAM address space to go to the AHB. This enables you to run BIST tests in the background, for instance the instruction SRAM can be BIST tested, 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 testing in debug state by using scan chain 15 to access the CP15 BIST control register. This is not necessarily recommended as the BIST test corrupts the contents of the SRAM 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, RAM and TAG BIST test registers* on page 2-28.

10.3.1 BIST control register

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

You can access the current status of a BIST test and 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.

In addition to returning the state for the size of the test memory array, having completed a BIST test, if you wish to use the memory array for 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 tightly coupled SRAM following a BIST test, without having first flushed the cache memory and reprogrammed the SRAM. This is necessary as the BIST algorithm corrupts all tested memory locations.

10.3.2 BIST address and general registers

The BIST control register enables you to perform standard BIST operations on each SRAM 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 a BIST test
- providing a nonzero starting address for a BIST test
- peek and poke of the SRAM
- returning an address location for a failed BIST test.

This additional functionality is most useful for debugging faulty silicon during production test. The exception to this is the start address for a BIST test. It is possible that BIST of the SRAM is performed periodically during program execution, the memory being tested in smaller pieces 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

ARM recommends that you do not write application code that relies on the presence of the BIST address and general registers. ARM does not guarantee to support these registers in future versions of the ARM946E-S.

Table 10-1 and Table 10-2 on page 10-7 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

Table 10-1 Instruction BIST address and general registers (continued)

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

10.3.3 Pause modes

ARM recommends that you use the following production test sequence for the SRAM:

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

To allow testing of the BIST hardware, a pause mechanism enables you to halt the BIST test. This allows you to corrupt data within the SRAM. The sequence for this is:

1. Write the address for the location to be corrupted with an `MCR` to the relevant BIST address register
2. Write the corrupted data using a `MCR` to the BIST general register.

You can then restart the test using an `MCR` to the BIST control register and check to see that the corrupted data causes the test to fail. You can read the fail address and data from the BIST address and general registers.

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

- *Auto pause* on page 10-8
- *User pause* on page 10-8.

Note

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

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. The BIST test pauses at predetermined points of the BIST algorithm, for instance when the algorithm has reached the top or the bottom of the memory array being tested.

You can poll the BIST control register to detect when a test has paused (the running flag is LOW). You can then corrupt the data, as described in *Pause modes* on page 10-7, before you restart the BIST test.

User pause

If the pause bit is clear when the test is activated, the test is run in user pause mode. The BIST algorithm is only paused by an MCR to the BIST control register setting the pause bit for the SRAM being tested. The SRAM contents are then corrupted as previously. This stops the BIST algorithm at a potentially unknown point, resulting in the possibility that the corrupted data is overwritten by the BIST algorithm and therefore not cause a test to fail.

Note

The user pause mode is provided for production test debugging where you might wish to shorten a test by pausing the algorithm very early. The auto-pause mechanism is recommended to provide deterministic BIST hardware testing for all other occasions.

Appendix A
AC Parameters

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

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-3
- *Coprocessor interface timing* on page A-4
- *Debug interface timing* on page A-5
- *JTAG interface timing* on page A-6
- *DBGSDOUT to DBGTDO timing* on page A-6
- *Exception and configuration timing* on page A-7
- *INTEST wrapper timing* on page A-7
- *ETM interface timing* on page A-8.

Clock, reset and AHB enable timing parameters are shown in Figure A-1.

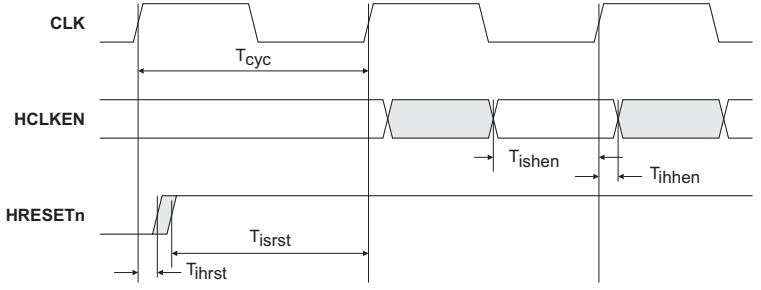


Figure A-1 Clock, reset, and AHB enable timing

AHB bus request and grant related timing parameters are shown in Figure A-2 on page A-3.

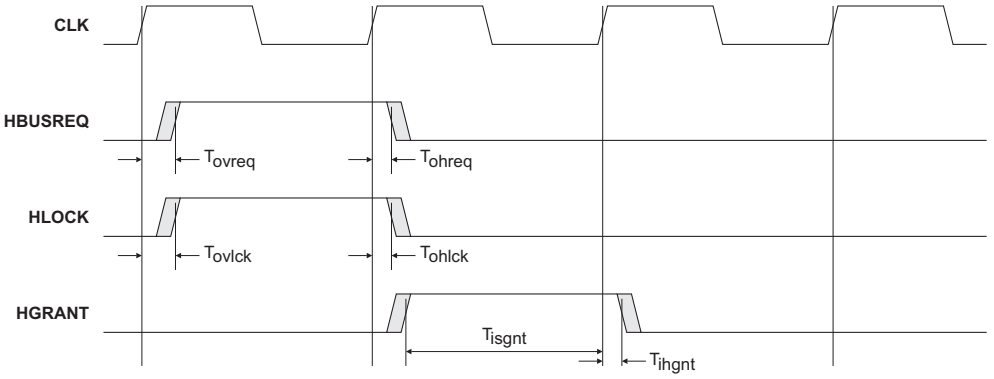


Figure A-2 AHB bus request and grant related timing

AHB bus master timing parameters are shown in Figure A-3.

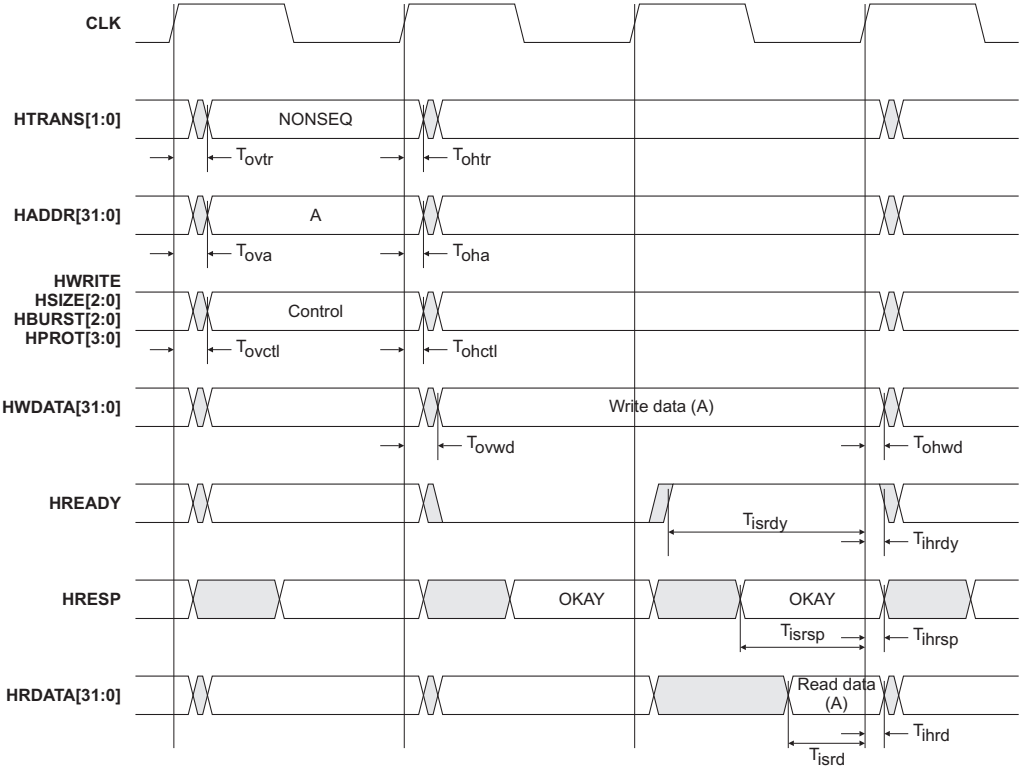


Figure A-3 AHB bus master timing

Coprocessor interface timing parameters are shown in Figure A-4.

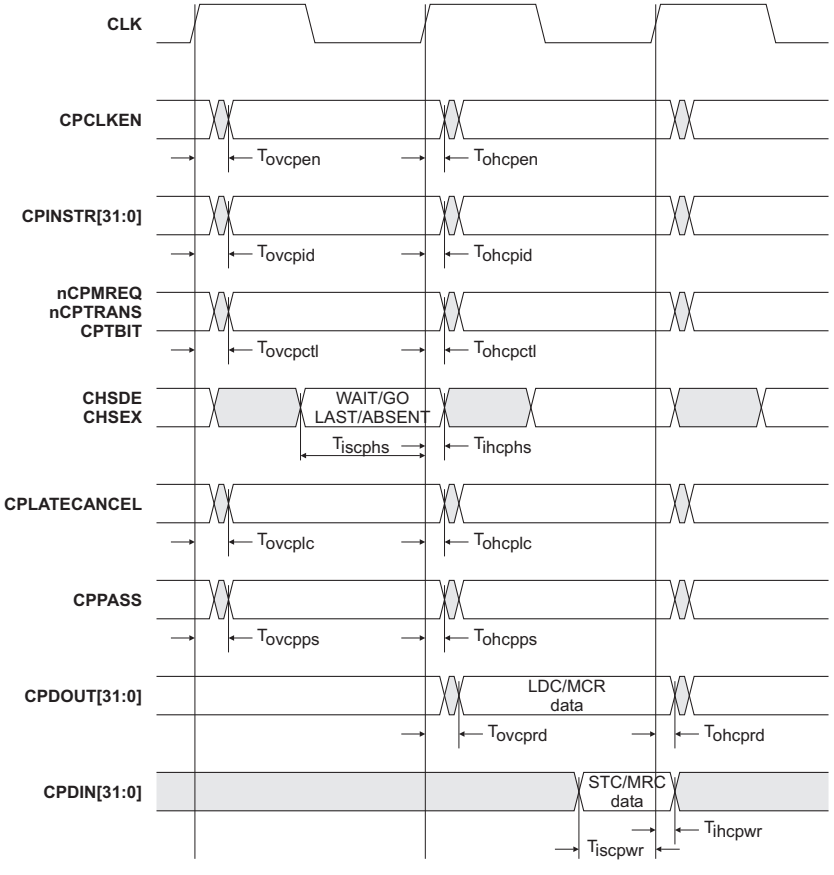


Figure A-4 Coprocessor interface timing

Debug interface timing parameters are shown in Figure A-5 on page A-5.

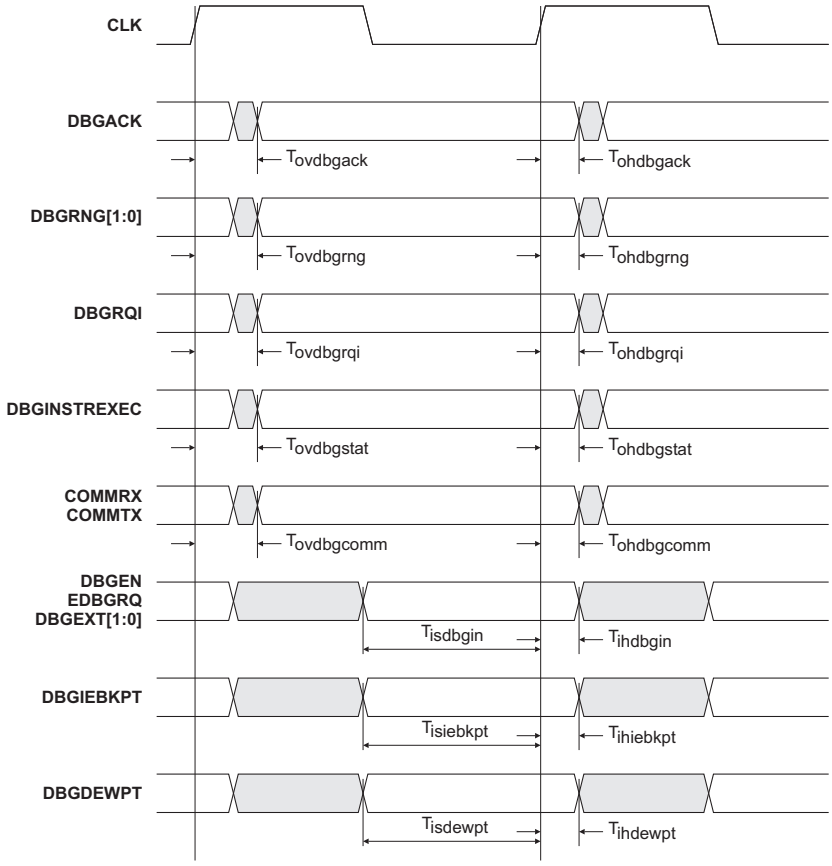


Figure A-5 Debug interface timing

JTAG interface timing parameters are shown inFigure A-6 on page A-6.

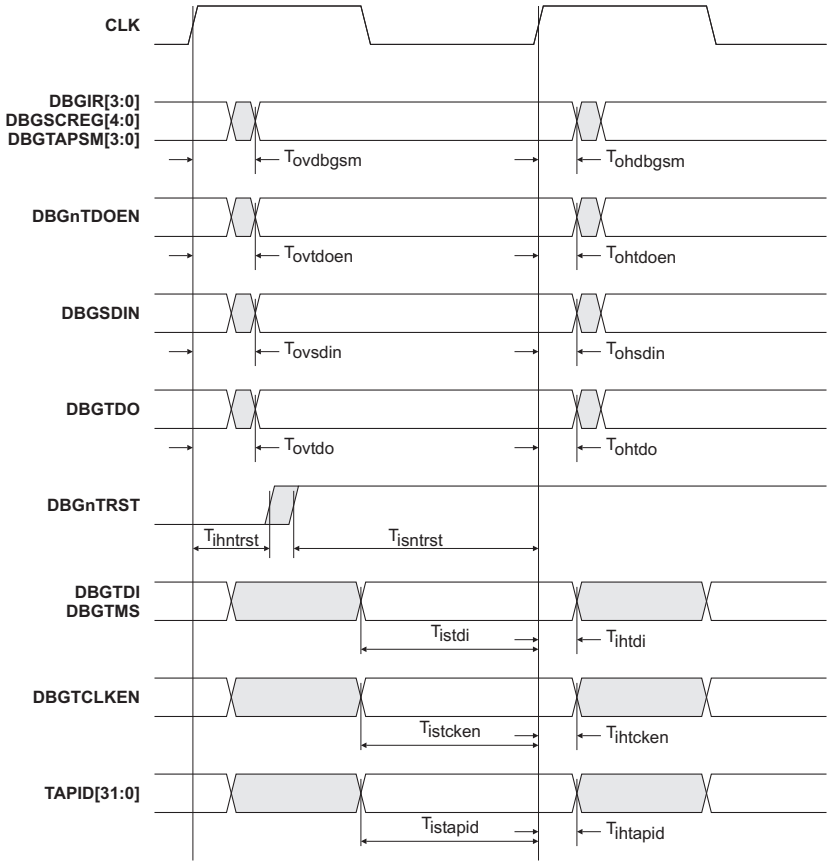


Figure A-6 JTAG interface timing

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

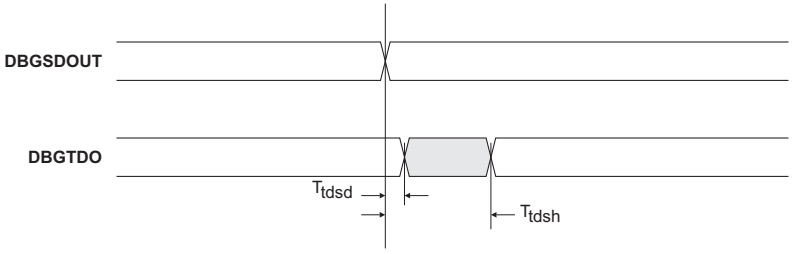


Figure A-7 DBGSDOUT to DBGTDO timing

Exception and configuration timing parameters are shown in Figure A-8.

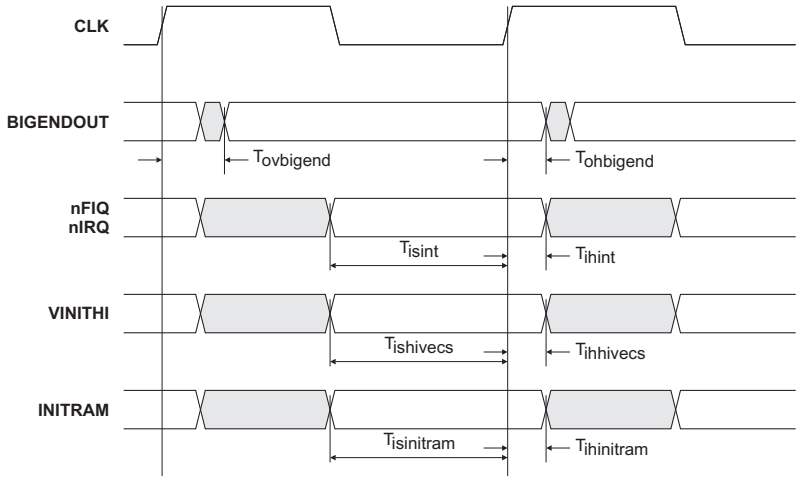


Figure A-8 Exception and configuration timing

The INTEST wrapper timing parameters are shown in Figure A-9.

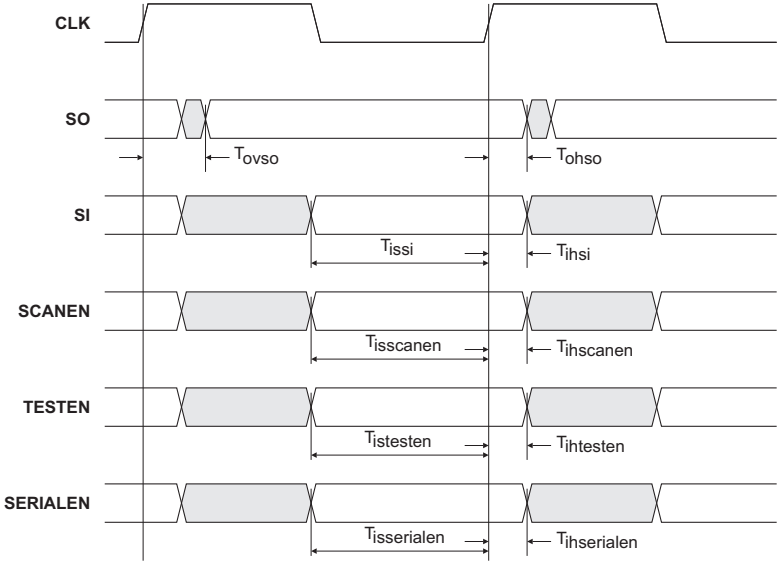


Figure A-9 INTEST wrapper timing

The ETM interface timing parameters are shown in Figure A-10 on page A-8.

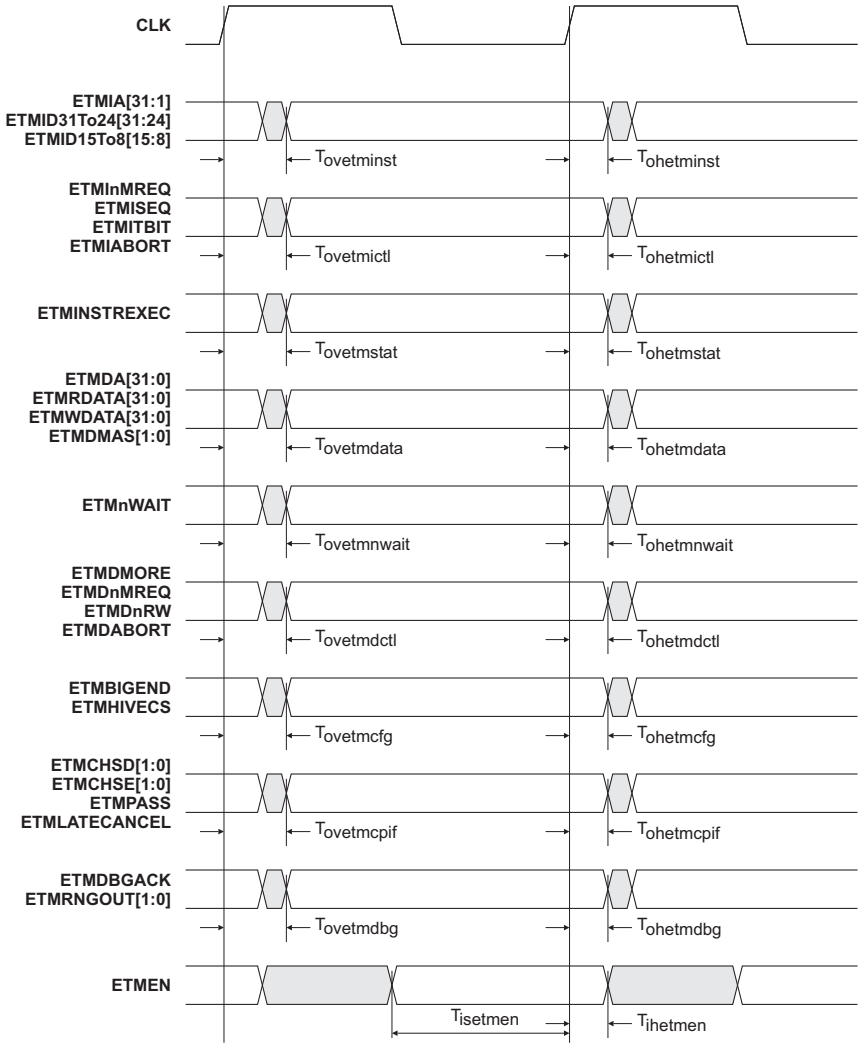


Figure A-10 ETM interface timing

A.2 AC timing parameter definitions

Table A-1 shows target AC 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

Table A-1 Timing parameter definitions

Symbol	Parameter	Min	Max
Tcyc	**CLK** cycle time	100%	-
Tishen	**HCLKEN** input setup to rising **CLK**	85%	-
Tihhen	**HCLKEN** input hold from rising **CLK**	-	0%
Tisrst	**HRESETn** *de-assertion* input setup to rising **CLK**	90%	-
Tihrst	**HRESETn** *de-assertion* input hold from rising **CLK**	-	0%
Tovreq	Rising **CLK** to **HBUSREQ** valid	-	30%
Tohreq	**HBUSREQ** hold time from rising **CLK**	>0%	-
Tovlck	Rising **CLK** to **HLOCK** valid	-	30%
Tohlck	**HLOCK** hold time from rising **CLK**	>0%	-
Tisgnt	**HGRANT** input setup to rising **CLK**	40%	-
Tihgnt	**HGRANT** input hold from rising **CLK**	-	0%
Tovtr	Rising **CLK** to **HTRANS[1:0]** valid	-	30%
Tohtr	**HTRANS[1:0]** hold time from rising **CLK**	>0%	-
Tova	Rising **CLK** to **HADDR[31:0]** valid	-	30%
Toha	**HADDR[31:0]** hold time from rising **CLK**	>0%	-
Tovctl	Rising **CLK** to AHB control signals valid	-	30%
Tohctl	AHB control signals hold time from rising **CLK**	>0%	-

Table A-1 Timing parameter definitions (continued)

Symbol	Parameter	Min	Max
Tovwd	Rising **CLK** to **HWDATA[31:0]** valid	-	30%
Tohwd	**HWDATA[31:0]** hold time from rising **CLK**	>0%	-
Tisrdy	**HREADY** input setup to rising **CLK**	75%	-
Tihrdy	**HREADY** input hold from rising **CLK**	-	0%
Tisrsp	**HRESP[1:0]** input setup to rising **CLK**	50%	-
Tihrsp	**HRESP[1:0]** input hold from rising **CLK**	-	0%
Tisrd	**HRDATA[31:0]** input setup to rising **CLK**	40%	-
Tihrd	**HRDATA[31:0]** input hold from rising **CLK**	-	0%
Tovcpen	Rising **CLK** to **CPCLKEN** valid	-	30%
Tohcpen	**CPCLKEN** hold time from rising **CLK**	>0%	-
Tovcpid	Rising **CLK** to **CPINSTR[31:0]** valid	-	30%
Tohcpid	**CPINSTR[31:0]** hold time from rising **CLK**	>0%	-
Tovcpctl	Rising **CLK** to transaction control valid	-	30%
Tohcpctl	Transaction control hold time from rising **CLK**	>0%	-
Tiscphs	Coprocessor handshake input setup to rising **CLK**	50%	-
Tihcphs	Coprocessor handshake input hold from rising **CLK**	-	0%
Tovcplc	Rising **CLK** to **CPLATECANCEL** valid	-	30%
Tohcplc	**CPLATECANCEL** hold time from rising **CLK**	>0%	-
Tovcpps	Rising **CLK** to **CPPASS** valid	-	30%
Tohcpps	**CPPASS** hold time from rising **CLK**	>0%	-
Tovcprd	Rising **CLK** to **CPDOUT[31:0]** valid	-	30%
Tohcprd	**CPDOUT[31:0]** hold time from rising **CLK**	>0%	-
Tiscpwr	**CPDIN[31:0]** input setup to rising **CLK**	40%	-
Tihcpwr	**CPDIN[31:0]** input hold from rising **CLK**	-	0%
Tovdbgack	Rising **CLK** to **DBGACK** valid	-	60%

Table A-1 Timing parameter definitions (continued)

Symbol	Parameter	Min	Max
Tohdbgack	**DBGACK** hold time from rising **CLK**	>0%	-
Tovdbgrng	Rising **CLK** to **DBGRNG[1:0]** valid	-	60%
Tohdbgrng	**DBGRNG[1:0]** hold time from rising **CLK**	>0%	-
Tovdbgrqi	Rising **CLK** to **DBGRQI** valid	-	45%
Tohdbgrqi	**DBGRQI** hold time from rising **CLK**	>0%	-
Tovdbgstat	Rising **CLK** to **DBGINSTREXEC** valid	-	30%
Tohdbgstat	**DBGINSTREXEC** hold time from rising **CLK**	>0%	-
Tovdbgcomm	Rising **CLK** to comms channel outputs valid	-	30%
Tohdbgcomm	Comms channel outputs hold time from rising **CLK**	>0%	-
Tisdbgin	Debug inputs input setup to rising **CLK**	30%	-
Tihdbgin	Debug inputs input hold from rising **CLK**	-	0%
Tisiebkpt	**DBGIEBKPT** input setup to rising **CLK**	20%	-
Tihiebkpt	**DBGIEBKPT** input hold from rising **CLK**	-	0%
Tisdewpt	**DBGDEWPT** input setup to rising **CLK**	20%	-
Tihdewpt	**DBGDEWPT** input hold from rising **CLK**	-	0%
Tovdbgsm	Rising **CLK** to debug state valid	-	30%
Tohdbgsm	Debug state hold time from rising **CLK**	>0%	-
Tovtdoen	Rising **CLK** to **DBGnTDOEN** valid	-	40%
Tohtdoen	**DBGnTDOEN** hold time from rising **CLK**	>0%	-
Tovsdin	Rising **CLK** to **DBGSDIN** valid	-	20%
Tohsdin	**DBGSDIN** hold time from rising **CLK**	>0%	-
Tovtdo	Rising **CLK** to **DBGTDO** valid	-	65%
Tohtdo	**DBGTDO** hold time from rising **CLK**	>0%	-
Tisntrst	**DBGnTRST** de-asserted input setup to rising **CLK**	35%	-

Table A-1 Timing parameter definitions (continued)

Symbol	Parameter	Min	Max
Tihntrst	**DBGnTRST** input hold from rising **CLK**	-	0%
Tistdi	Tap state control input setup to rising **CLK**	25%	-
Tihtdi	Tap state control input hold from rising **CLK**	-	0%
Tistcken	**DBGTCKEN** input setup to rising **CLK**	50%	-
Tihtcken	**DBGTCKEN** input hold from rising **CLK**	-	0%
Tistapid	**TAPID[31:0]** input setup to rising **CLK**	20%	-
Tihtapid	**TAPID[31:0]** input hold from rising **CLK**	-	0%
Tdsd	**DBGTDO** delay from **DBGSDOUTBS** changing	-	30%
Tdsh	**DBGTDO** hold time from **DBGSDOUTBS** changing	>0%	-
Tovbigend	Rising **CLK** to **BIGENDOUT** valid	-	30%
Tohbigend	**BIGENDOUT** hold time from rising **CLK**	>0%	-
Tisint	Interrupt input setup to rising **CLK**	15%	-
Tihint	Interrupt input hold from rising **CLK**	-	0%
Tishivecs	**VINITHI** input setup to rising **CLK**	95%	-
Tihhivecs	**VINITHI** input hold from rising **CLK**	-	0%
Tisinitram	**INITRAM** input setup to rising **CLK**	95%	-
Tihinitram	**INITRAM** input hold from rising **CLK**	-	0%
Tovso	Rising **CLK** to **SO** valid	-	30%
Tohso	**SO** hold time from rising **CLK**	>0%	-
Tissi	**SI** input setup to rising **CLK**	95%	-
Tihsi	**SI** input hold from rising **CLK**	-	0%
Tisscanen	**SCANEN** input setup to rising **CLK**	95%	-
Tihscanen	**SCANEN** input hold from rising **CLK**	-	0%
Tistesten	**TESTEN** input setup to rising **CLK**	95%	-
Tihtesten	**TESTEN**input hold from rising **CLK**	-	0%

Table A-1 Timing parameter definitions (continued)

Symbol	Parameter	Min	Max
Tisserialen	**SERIALEN** input setup to rising **CLK**	95%	-
Tihserialen	**SERIALEN** input hold from rising **CLK**	-	0%
Tovetminst	Rising **CLK** to ETM instruction interface valid	-	30%
Tohetminst	ETM instruction interface hold time from rising **CLK**	>0%	-
Tovetmictl	Rising **CLK** to ETM instruction control valid	-	30%
Tohetmictl	ETM instruction control hold time from rising **CLK**	>0%	-
Tovetmstat	Rising **CLK** to **ETMINSTREXEC** valid	-	30%
Tohetmstat	**ETMINSTREXEC** hold time from rising **CLK**	>0%	-
Tovetmdata	Rising **CLK** to ETM data interface valid	-	30%
Tohetmdata	ETM data interface hold time from rising CLK	>0%	-
Tovetmnwait	Rising **CLK** to **ETMnWAIT** valid	-	30%
Tohetmnwait	**ETMnWAIT** hold time from rising **CLK**	>0%	-
Tovetmdctl	Rising **CLK** to ETM data control valid	-	30%
Tohetmdctl	ETM data control hold time from rising **CLK**	>0%	-
Tovetmcfg	Rising **CLK** to ETM configuration valid	-	30%
Tohetmcfg	ETM configuration hold time from rising **CLK**	>0%	-
Tovetmcpif	Rising **CLK** to ETM coprocessor signals valid	-	30%
Tohetmcpif	ETM coprocessor signals hold time from rising **CLK**	>0%	-
Tovetmdbg	Rising **CLK** to ETM debug signals valid	-	30%
Tohetmdbg	ETM debug signals hold time from rising **CLK**	>0%	-
Tisetmen	**ETMEN** input setup to rising **CLK**	50%	-
Tihetmen	**ETMEN** input hold from rising **CLK**	-	0%

Note

The **VINITHI** pin is specified as 95% of the cycle because it is for input configuration during reset and can be considered static.

The INTEST wrapper inputs/outputs are specified as 95% of the cycle as they are production test related and expected to operate at typically 50% of the functional clock rate.

Appendix B
Signal Descriptions

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

B.1 Signal properties and requirements

In order to ensure ease of integration of the ARM946E-S 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 signals as all outputs change from the rising edge and all inputs are sampled with the rising edge of the clock. In addition, all signals are either input or output only, as bidirectional signals are not used.

Note

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

B.2 Clock interface signals

Table B-1 describes 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 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 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.

B.3 AHB signals

Table B-2 describes the ARM946E-S AHB signals.

Table B-2 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 supports SINGLE transfer (000) and INCRemental burst of unspecified length (001).
HBUSREQ Bus request	Output	Indicates that the ARM946E-S requires the bus.
HGRANT Bus grant	Input	Indicates that the ARM946E-S 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 gets access to the bus when both **HREADY** and **HGRANT** are HIGH.
HLOCK Request locked transfers	Output	When HIGH, indicates that the ARM946E-S requires locked access to the bus and no other master must be granted until this signal has gone LOW. Asserted by the ARM946E-S when executing SWP instructions to AHB address space.
HPROT[3:0] Protection control	Output	Indicates that the ARM946E-S transfer is an opcode fetch (0--0) or data access (0--1). Indicates if the transfer is User mode access (0-0-) or a Supervisor mode access (0-1-). Indicates that an access is nonbufferable (00--) or bufferable (01--). Bit [3] is tied LOW indicating noncachable.
HRDATA[31:0] Read data bus	Input	The 32-bit read data bus transfers data from a selected bus slave to the ARM946E-S 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	Asynchronously asserted LOW input used to initialize the ARM946E-S system state. Synchronously de-asserted.

Table B-2 AHB signals (continued)

Name	Direction	Description
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 (00), ERROR (01), RETRY (10), or SPLIT (11).
HSIZE[2:0] Transfer size	Output	Indicates the size of an ARM946E-S transfer. This can be Byte (000), Halfword (001), or Word (010). Bit [2] is tied LOW.
HTRANS[1:0] Transfer type	Output	Indicates the type of ARM946E-S transfer. This can be IDLE (00), NONSEQ (10), or SEQ (11).
HWDATA[31:0] Write data bus	Output	The 32-bit write data bus transfers data from the ARM946E-S 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.4 Coprocessor interface signals

Table B-3 describes the ARM946E-S coprocessor interface signals.

Table B-3 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's pipeline follower. Indicates ABSENT (10), WAIT (00), GO (01), or LAST (11).
CHSEX[1:0] Coprocessor handshake execute	Input	The handshake signals from the Execute stage of the coprocessor's pipeline follower. Indicates ABSENT (10), WAIT (00), GO (01), or LAST (11).

Table B-3 Coprocessor interface signals (continued)

Name	Direction	Description
CPTBIT Coprocessor instruction Thumb bit	Output	When HIGH indicates that the ARM946E-S is in Thumb state. When LOW indicates that the ARM946E-S 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 is in User mode. When HIGH indicates that the ARM946E-S is in Privileged mode. Sampled by the coprocessor pipeline follower.

B.5 Debug signals

Table B-4 describes the ARM946E-S debug signals.

Table B-4 Debug signals

Name	Direction	Description
COMMRX Communications channel receive	Output	When HIGH denotes that the comms channel receive buffer contains valid data waiting to be read.
COMMTX Communications channel transmit	Output	When HIGH, denotes that the comms channel transmit buffer is empty.
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 to enter debug state.
DBGEN Debug enable	Input	Enables the debug features of the processor. This signal must be tied LOW if debug is not required.
DBGEXT[1:0] EmbeddedICE-RT external input	Input	Input to the EmbeddedICE-RT logic allows 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 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 processor's pipeline has been executed.

Table B-4 Debug signals (continued)

Name	Direction	Description
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.6 JTAG signals

Table B-5 describes the ARM946E-S JTAG signals.

Table B-5 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	Internally synchronized active LOW reset signal for the EmbeddedICE-RT internal state.
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** 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.
DBGTCKEN	Input	Synchronous enable test clock.
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.7 Miscellaneous signals

Table B-6 describes the ARM946E-S miscellaneous signals.

Table B-6 Miscellaneous signals

Name	Direction	Description
BIGENDOUT	Output	When HIGH, the ARM946E-S 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`.

B.8 ETM interface signals

Table B-7 describes the ARM946E-S ETM interface signals.

Table B-7 ETM interface signals

Name	Direction	Description
ETMEN	Input	Synchronous ETM interface enable. 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.
ETMRNGOUT[1:0]	Output	Watchpoint register match indication for the ETM.
ETMID31TO25[31:25]	Output	Instruction data field for the ETM.

Table B-7 ETM interface signals (continued)

Name	Direction	Description
ETMID15TO11[15:11]	Output	Instruction data field for the ETM.
ETMCHSD[1:0]	Output	Coprocessor handshake decode signals for the ETM.
ETMCHSE[1:0]	Output	Coprocessor handshake execute signals for the ETM.
ETMPASS	Output	Coprocessor instruction execute indication for the ETM.
ETMLATECANCEL	Output	Coprocessor late cancel indication for the ETM.
ETMPROCID[31:0]	Output	Process identifier for the ETM.
ETMPROCIDWR	Output	ETMPROCID write strobe.
ETMINSTRVALID	Output	Instruction valid indication for the ETM.

B.9 INTEST wrapper signals

Table B-8 describes the ARM946E-S INTEST wrapper signals.

Table B-8 INTEST wrapper signals

Name	Direction	Description
INnotEXTEST	Input	Selects between INTEST and EXTEST mode of the INTEST wrapper scan chain.
SI	Input	Serial input data for the INTEST wrapper scan chain.
SO	Output	Serial output data from the INTEST wrapper scan chain.
SCANEN	Input	Enables scanning of data through the INTEST wrapper scan chain.
TESTEN	Input	Selects the INTEST wrapper scan chain as the source for ARM946E-S inputs.
SERIALEN	Input	Enables the INTEST wrapper BIST activation mode where the scan chain applies serialized ARM instructions to the ARM946E-S to activate BIST test of the tightly-coupled SRAM.

Index

D

E

F

I

J

L

M

N

O

P

R

S

T

U

W

