ARM926EJ-S

(r0p4/r0p5)

Technical Reference Manual

ARM

ARM DDI0198D

ARM926EJ-S
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
26 September 2001	A	First release
29 January 2002	B	Second release
5 December 2003	C	Third release. Includes r0p5 changes. Defects corrected.
26 January 2004	D	Fourth release. Includes r0p4. Technically identical to previous release.

Proprietary Notice

Confidentiality Status

This document is Open Access. This document has no restriction on distribution.

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM926EJ-S Technical Reference Manual

List of Tables
ARM926EJ-S Technical Reference Manual

List of Figures
ARM926EJ-S Technical Reference Manual

Preface

This preface introduces the *ARM926EJ-S Revision r0p4/r0p5 Technical Reference Manual* (TRM). It contains the following sections:

About this manual

This is the Technical Reference Manual for the ARM926EJ-S processor.

Product revision status

The r*n*p*n* identifier indicates the revision status of the product described in this manual, where:

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

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

Intended audience

This document has been written for experienced hardware and software engineers who have previous experience of ARM products, and who wish to use an ARM926EJ-S processor in their system design.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an overview of the ARM926EJ-S processor.

Chapter 2 *Programmer's Model*

Read this chapter for details of the programmer's model and ARM926EJ-S registers.

Chapter 3 *Memory Management Unit*

Read this chapter for details of the *Memory Management Unit* (MMU) and address translation process and how to use the CP15 register to enable and disable the MMU.

Chapter 4 *Caches and Write Buffer*

Read this chapter for a description of the instruction cache, the data cache, the write buffer, and the physical address tag RAM.

Chapter 5 *Tightly-Coupled Memory Interface*

Read this chapter for a description of the *Tightly-Coupled Memory* (TCM) interface and how to use the CP15 region register to enable and disable the caches. It includes examples on how various RAM types can be connected.

Chapter 6 *Bus Interface Unit*

Read this chapter for a description of the *Bus Interface Unit* (BIU) interface to AMBA.

Chapter 7 *Noncachable Instruction Fetches*

Read this chapter for a description of how speculative noncachable instruction fetches are used in the ARM926EJ-S processor to improve performance.

Chapter 8 *Coprocessor Interface*

Read this chapter for a description of the coprocessor interface. The chapter includes timing diagrams for coprocessor operations.

Chapter 9 *Instruction Memory Barrier*

Read this chapter for the *Instruction Memory Barrier* (IMB) description and how IMB operations are used to ensure consistency between data and instruction streams processed by the ARM926EJ-S processor.

Chapter 10 *Embedded Trace Macrocell Support*

Read this chapter to understand how *Embedded Trace Macrocell* (ETM) is supported in the ARM926EJ-S processor.

Chapter 11 *Debug Support*

Read this chapter for a description of the debug interface and EmbeddedICE-RT.

Chapter 12 *Power Management*

Read this chapter for a description of the power management facilities provided by the ARM926EJ-S processor.

Appendix A *Signal Descriptions*

This appendix lists the ARM926EJ-S processor signals in functional groups.

Appendix B *CP15 Test and Debug Registers*

Read this appendix for detailed information on the registers used for test and debug.

Conventions

This section describes the conventions that this manual uses:

- *Typographical*
- *Timing diagrams*
- *Signal naming* on page xix
- *Numbering* on page xx.

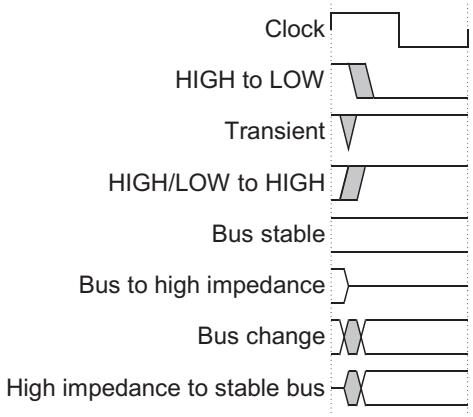
Typographical

This manual uses the following typographical conventions:

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

Timing diagrams

This manual contains one or more timing diagrams. The figure named *Key to timing diagram conventions* on page xix on page xix explains the components used in these diagrams. When variations occur they have clear labels. You must not assume any timing information that is not explicit in the diagrams.



Key to timing diagram conventions

Signal naming

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means HIGH for active-HIGH signals and LOW for active-LOW signals:

Prefix H Denotes *Advanced High-performance Bus* (AHB) signals.

Prefix n Denotes active-LOW signals except in the case of AHB or *Advanced Peripheral Bus* APB reset signals. These are named **HRESETn** and **PRESETn** respectively.

Prefix DH Denotes data side AHB signals.

Prefix IH Denotes instruction side AHB signals.

Prefix DR Denotes data side TCM interface signals.

Prefix IR Denotes instruction side TCM interface signals.

Prefix ETM Denotes ETM interface signals.

Prefix DBG Denotes debug/JTAG signals.

Prefix CP Denotes coprocessor interface signals.

Numbering

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

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

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

Further reading

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

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

ARM publications

This manual contains information that is specific to the ARM926EJ-S processor. Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM AMBA Specification (Rev 2.0)* (ARM IHI 0001)
- *ARM926EJ-S Implementation Guide* (ARM DII 0015)
- *ARM926EJ-S Test Chip Implementation Guide* (ARM DXI 0131)
- *ARM9EJ-S Technical Reference Manual* (ARM DDI 0222)
- *Multi-layer AHB Overview* (ARM DVI 0045)
- *ETM9 Technical Reference Manual* (ARM DDI 0157).

Feedback

ARM Limited welcomes feedback on the ARM926EJ-S processor and its documentation.

Feedback on the product

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

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

Feedback on this manual

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

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

ARM Limited also welcomes general suggestions for additions and improvements.

Chapter 1
Introduction

This chapter introduces the ARM926EJ-S processor and its features. It contains the following section:

- *About the ARM926EJ-S processor* on page 1-2.

1.1 About the ARM926EJ-S processor

The ARM926EJ-S processor is a member of the ARM9 family of general-purpose microprocessors. The ARM926EJ-S processor is targeted at multi-tasking applications where full memory management, high performance, low die size, and low power are all important.

The ARM926EJ-S processor supports the 32-bit ARM and 16-bit Thumb instruction sets, enabling the user to trade off between high performance and high code density. The ARM926EJ-S processor includes features for efficient execution of Java byte codes, providing Java performance similar to JIT, but without the associated code overhead.

The ARM926EJ-S processor supports the ARM debug architecture and includes logic to assist in both hardware and software debug. The ARM926EJ-S processor has a Harvard cached architecture and provides a complete high-performance processor subsystem, including:

- an ARM9EJ-S integer core
- a *Memory Management Unit* (MMU)
- separate instruction and data AMBA AHB bus interfaces
- separate instruction and data TCM interfaces.

The ARM926EJ-S processor provides support for external coprocessors enabling floating-point or other application-specific hardware acceleration to be added. The ARM926EJ-S processor implements ARM architecture version 5TEJ.

The ARM926EJ-S processor is a synthesizable macrocell. This means that you can optimize the macrocell for a particular target library, and that you can configure the memory system to suit your target application. You can individually configure the cache sizes to be any power of two between 4KB and 128KB.

The tightly-coupled instruction and data memories are instantiated externally to the ARM926EJ-S macrocell, providing you with the flexibility of optimizing the memory subsystem for performance, power, and particular RAM type. The TCM interfaces enable nonzero wait state memory to be attached, as well as providing a mechanism for supporting DMA.

Figure 1-1 on page 1-3 shows the main blocks in the ARM926EJ-S processor.

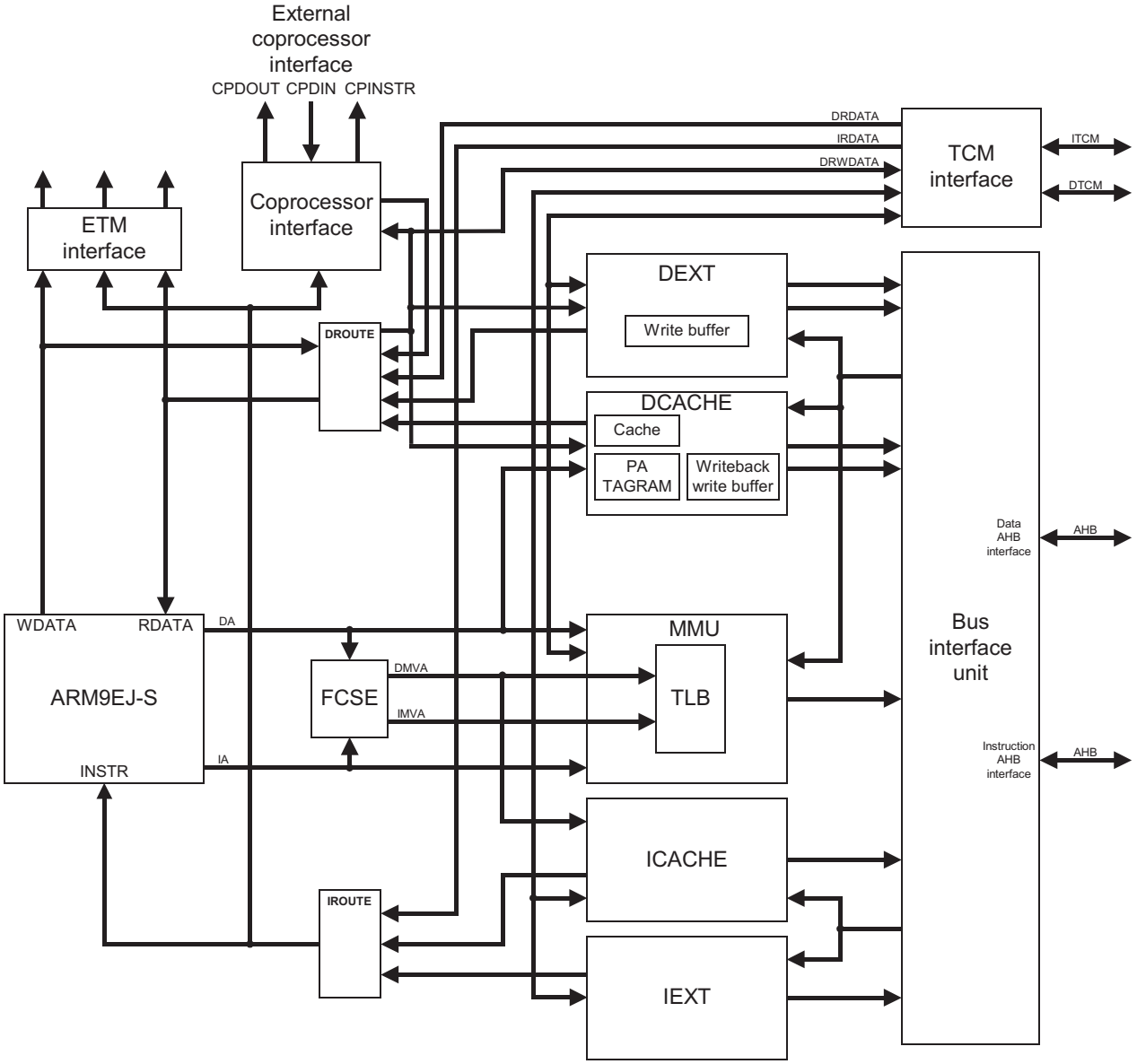


Figure 1-1 ARM926EJ-S block diagram

Figure 1-2 on page 1-4 and Figure 1-3 on page 1-5 show the ARM926EJ-S interfaces.

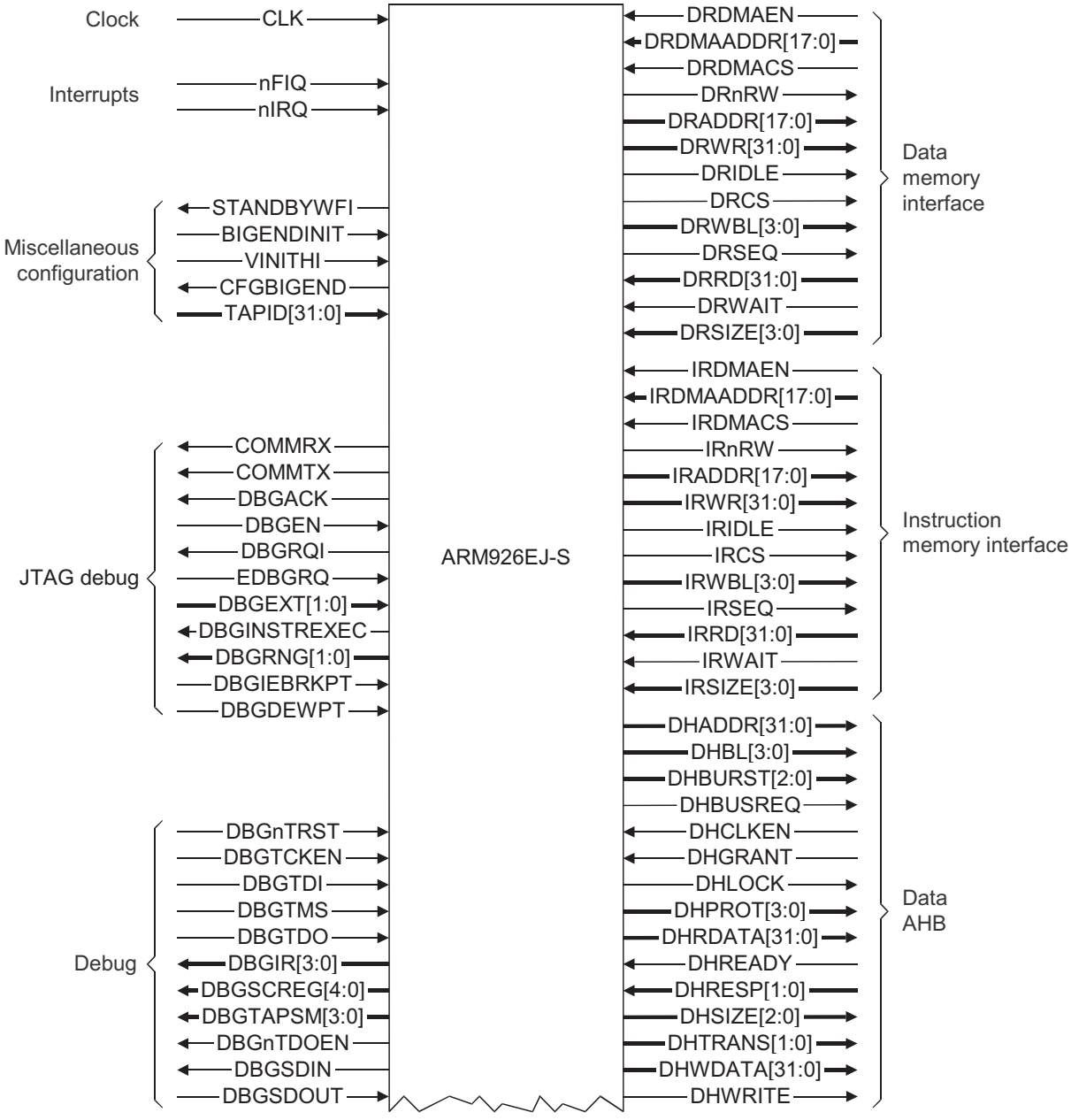


Figure 1-2 ARM926EJ-S interface diagram (part one)

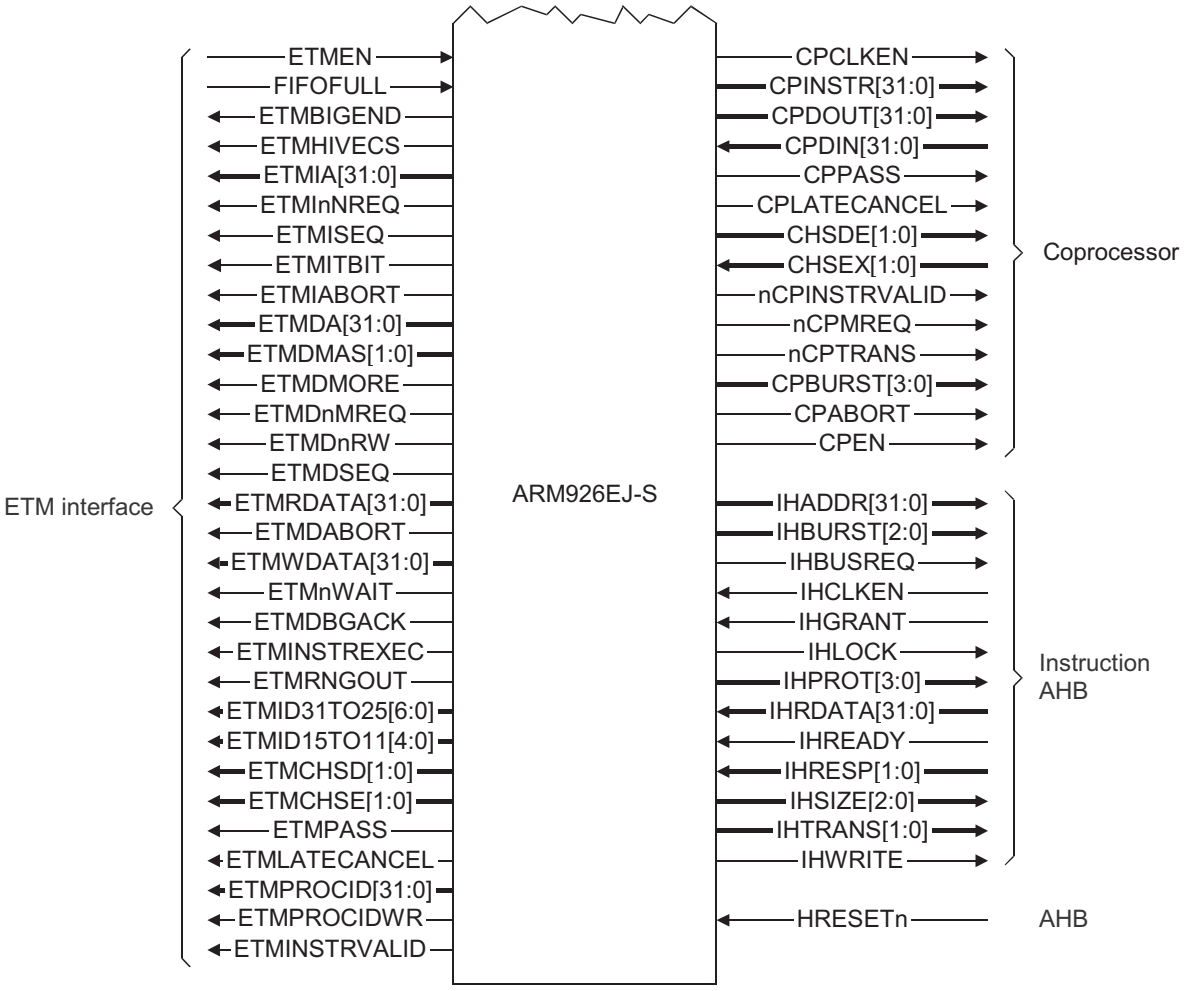


Figure 1-3 ARM926EJ-S interface diagram (part two)

Chapter 2
Programmer's Model

This chapter describes the ARM926EJ-S registers in CP15, the system control coprocessor, and provides information for programming the microprocessor. It contains the following sections:

2.1 About the programmer's model

The system control coprocessor (CP15) is used to configure and control the ARM926EJ-S processor. The caches, *Tightly-Coupled Memories* (TCMs), *Memory Management Unit* (MMU), and most other system options are controlled using CP15 registers. You can only access CP15 registers with MRC and MCR instructions in a privileged mode. CDP, LDC, STC, MCRR, and MRRC instructions, and unprivileged MRC or MCR instructions to CP15 cause the Undefined instruction exception to be taken.

2.2 Summary of ARM926EJ-S system control coprocessor (CP15) registers

CP15 defines 16 registers. Table 2-1 shows the read and write functions of the registers.

Table 2-1 CP15 register summary

Register	Reads	Writes
0	ID code[a]	Unpredictable
0	Cache type[a]	Unpredictable
0	TCM status[a]	Unpredictable
1	Control	Control
2	Translation table base	Translation table base
3	Domain access control	Domain access control
4	Reserved	Reserved
5	Data fault status[a]	Data fault status[a]
5	Instruction fault status[a]	Instruction fault status[a]
6	Fault address	Fault address
7	Cache operations	Cache operations
8	Unpredictable	TLB operations
9	Cache lockdown[b]	Cache lockdown
9	TCM region	TCM region
10	TLB lockdown	TLB lockdown
11 and 12	Reserved	Reserved
13	FCSE PID[a]	FCSE PID[a]
13	Context ID[a]	Context ID[a]
14	Reserved	Reserved
15	Test configuration	Test configuration

a. Register locations 0, 5, and 13 each provide access to more than one register. The register accessed depends on the value of the `Opcode_2` field.

b. Register location 9 provides access to more than one register. The register accessed depends on the value of the `CRm` field. See the register descriptions for details.

All CP15 register bits that are defined and contain state are set to 0 by Reset except:

- The V bit is set to 0 at reset if the **VINITHI** signal is LOW, or 1 if the **VINITHI** signal is HIGH.
- The B bit is set to 0 at reset if the **BIGENDINIT** signal is LOW, or 1 if the **BIGENDINIT** signal is HIGH.
- The instruction TCM is enabled at reset if the **INITRAM** pin is HIGH. This enables booting from the instruction TCM and sets the ITCM bit in the ITCM region register to 1.

2.2.1 Addresses in an ARM926EJ-S system

Three distinct types of address exist in an ARM926EJ-S system. Table 2-2 shows the address types in ARM926EJ-S processor.

Table 2-2 Address types in ARM926EJ-S

Domain	ARM9EJ-S	Caches and MMU	TCM and AMBA bus
Address type	*Virtual Address* (VA)	*Modified Virtual Address* (MVA)	*Physical Address* (PA)

This is an example of the address manipulation that occurs when the ARM9EJ-S core requests an instruction:

1. The VA of the instruction is issued by the ARM9EJ-S core.
2. The VA is translated using the FCSE PID value to the MVA. The *Instruction Cache* (ICache) and *Memory Management Unit* (MMU) detect the MVA (see *Process ID Register c13* on page 2-33).
3. If the protection check carried out by the MMU on the MVA does not abort and the MVA tag is in the ICache, the instruction data is returned to the ARM9EJ-S core.
4. If the protection check carried out by the MMU on the MVA does not abort, and the cache misses (the MVA tag is not in the cache), then the MMU translates the MVA to produce the PA. This address is given to the AMBA bus interface to perform an external access.

2.2.2 Accessing CP15 registers

You can only access CP15 registers 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 2-5.

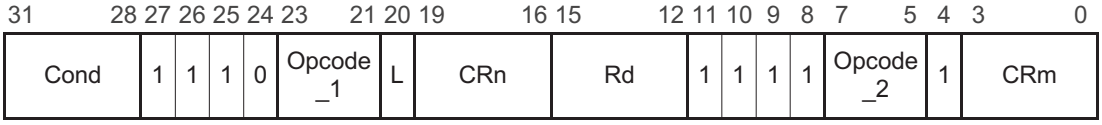


Figure 2-1 CP15 MRC and MCR bit pattern

The mnemonics for these instructions are:

```
MCR{cond} p15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
MRC{cond} p15,<Opcode_1>,<Rd>,<CRn>,<CRm>,<Opcode_2>
```

Attempting to read from a write-only register, or writing to a read-only register causes Unpredictable results. In all instructions that access CP15:

- The Opcode_1 field Should Be Zero except when the values specified are used to select the desired operations. Using other values results in Unpredictable behavior.
- The Opcode_2 and CRm fields Should Be Zero except when the values specified are used to select the desired behavior. Using other values results in Unpredictable behavior.

Table 2-3 shows the terms and abbreviations used in this chapter.

Table 2-3 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 exception.
Should Be Zero	SBZ	When writing to this location, all bits of this field Should Be Zero.

Table 2-3 CP15 abbreviations (continued)

Term	Abbreviation	Description
Should Be One	SBO	When writing to this location, all bits in this field Should Be One.
Should Be Zero or Preserved	SBZP	When writing to this location, all bits of this field Should Be Zero or preserved by writing the same value that has been previously read from the same field.

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

2.3 Register descriptions

The following registers are described in this section:

- *ID Code, Cache Type, and TCM Status Registers, c0*
- *Control Register c1* on page 2-12
- *Translation Table Base Register c2* on page 2-17
- *Domain Access Control Register c3* on page 2-17
- *Register c4* on page 2-18
- *Fault Status Registers c5* on page 2-18
- *Fault Address Register c6* on page 2-20
- *Cache Operations Register c7* on page 2-20
- *TLB Operations Register c8* on page 2-24
- *Cache Lockdown and TCM Region Registers c9* on page 2-26
- *TLB Lockdown Register c10* on page 2-32
- *Register c11 and c12* on page 2-33
- *Process ID Register c13* on page 2-33
- *Register c14* on page 2-35
- *Test and Debug Register c15* on page 2-36.

2.3.1 ID Code, Cache Type, and TCM Status Registers, c0

Register c0 accesses the ID Register, Cache Type Register, and TCM Status Registers. Reading from this register returns the device ID, the cache type, or the TCM status depending on the value of Opcode_2 used:

Opcode_2 = 0 ID value.

Opcode_2 = 1 instruction and data cache type.

Opcode_2 = 2 TCM status.

The CRm field Should Be Zero when reading from these registers. Table 2-4 shows the instructions you can use to read register c0.

Table 2-4 Reading from register c0

Function	Instruction
Read ID code	`MRC p15,0,<Rd>,c0,c0,{0, 3-7}`
Read cache type	`MRC p15,0,<Rd>,c0,c0,1`
Read TCM status	`MRC p15,0,<Rd>,c0,c0,2`

Writing to register c0 is Unpredictable.

ID Code Register c0

This is a read-only register that returns the 32-bit device ID code.

You can access the ID Code Register by reading CP15 register c0 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
```

The contents of the ID Code Register are shown in Table 2-5.

Table 2-5 Register 0, ID code

Register bits	Function	Value
[31:24]	ASCII code of implementer trademark	0x41
[23:20]	Variant	0x0
[19:16]	Architecture (ARMv5TEJ)	0x6
[15:4]	Part number	0x926
[3:0]	Revision	0x05[a]

a. The revision value can be in the range 0x0 to 0x5, depending on the layout revision you are using..

Cache Type Register c0

This is a read-only register that contains information about the size and architecture of the *Instruction Cache* (ICache) and *Data Cache* (DCache) enabling operating systems to establish how to perform such operations as cache cleaning and lockdown.

You can access the cache type register by reading CP15 register c0 with the Opcode_2 field set to 1. For example:

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

The format of the Cache Type Register is shown in Figure 2-2 on page 2-9.

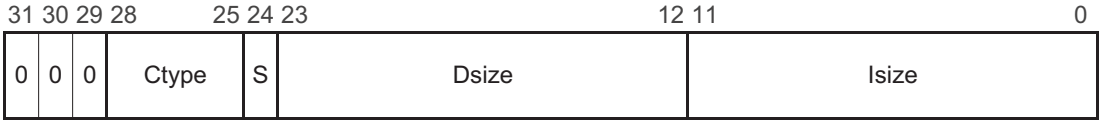


Figure 2-2 Cache Type Register format

Ctype The Ctype field determines the cache type. See Table 2-6.

S bit Specifies if the cache is a unified cache (S=0), or separate ICache and DCache (S=1). If S=0, the Isize and Dsize fields both describe the unified cache and must be identical. In the ARM926EJ-S processor, this bit is set to a 1 to denote separate caches.

Dsize Specifies the size, line length, and associativity of the DCache, or of the unified cache if the S bit is 0.

Isize Specifies the size, length, and associativity of the ICache, or of the unified cache if the S bit is 0.

The Ctype field specifies if the cache supports lockdown or not, and how it is cleaned. The encoding is shown in Table 2-6. All unused values are reserved.

Table 2-6 Ctype encoding

Value	Method	Cache cleaning	Cache lockdown
b1110	Write-back	Register 7 operations	Format C[a]

a. See *Cache Lockdown Register c9* on page 2-26 for more details on Format C for cache lockdown.

The Dsize and Isize fields in the Cache Type Register have the same format. This is shown in Figure 2-3.

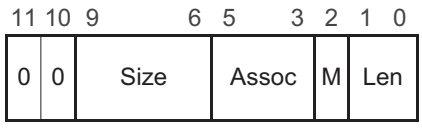


Figure 2-3 Dsize and Isize field format

Size The Size field determines the cache size in conjunction with the M bit.

Assoc The Assoc field determines the cache associativity in conjunction with the M bit.

M bit The multiplier bit determines the cache size and cache associativity values in conjunction with the Size and Assoc fields. If the cache is present, M must be set to 0. If the cache is absent, M must be set to 1. For the ARM926EJ-S processor, M is always set to 0.

Len The Len field determines the line length of the cache.

The size of the cache is determined by the Size field and the M bit. The M bit is 0 for the DCache and ICache. The Size field is bits [21:18] for the DCache and bits [9:6] for the ICache. The minimum size of each cache is 4KB, and the maximum size is 128KB. Table 2-7 shows the cache size encoding.

Table 2-7 Cache size encoding (M=0)

Size field	Cache size
b0011	4KB
b0100	8KB
b0101	16KB
b0110	32KB
b0111	64KB
b1000	128KB

The associativity of the cache is determined by the Assoc field and the M bit. The M bit is 0 for the DCache and ICache. The Assoc field is bits [17:15] for the DCache and bits [5:3] for the ICache. Table 2-8 shows the cache associativity encoding.

Table 2-8 Cache associativity encoding (M=0)

Assoc field	Associativity
b010	4-way
Other values	Reserved

The line length of the cache is determined by the Len field. The Len field is bits [13:12] for the DCache and bits [1:0] for the ICache. Table 2-9 shows the line length encoding.

Table 2-9 Line length encoding

Len field	Cache line length
b10	8 words (32 bytes)
Other values	Reserved

The cache type register values for an ARM926EJ-S processor with the following configuration are shown in Table 2-10:

- separate instruction and data caches
- DCache size = 8KB, ICache size = 16KB
- associativity = 4-way
- line length = eight words
- caches use write-back, register 7 for cache cleaning, and Format C for cache lockdown.

See *Cache Lockdown Register c9* on page 2-26 for more details on Format C for cache lockdown.

Table 2-10 Example Cache Type Register format

Function		Register bits	Value
Reserved		[31:29]	b000
Ctype		[28:25]	b1110
S		[24]	b1 = Harvard cache
Dsize	Reserved	[23:22]	b00
	Size	[21:18]	b0100 = 8KB
	Assoc	[17:15]	b010 = 4-way
	M	[14]	b0
	Len	[13:12]	b10 = 8 words per line (32 bytes)

Table 2-10 Example Cache Type Register format (continued)

Function		Register bits	Value
Isize	Reserved	[11:10]	b00
	Size	[9:6]	b0101 = 16KB
	Assoc	[5:3]	b010 = 4-way
	M	[2]	b0
	Len	[1:0]	b10 = 8 words per line (32 bytes)

TCM Status Register c0

This is a read-only register that enables operating systems to establish if TCM memories are present. See also *TCM Region Register c9* on page 2-29.

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

```
MRC p15,0,<Rd>,c0,c0,2 ;returns TCM details
```

The format of the TCM Status Register is shown in Figure 2-4.

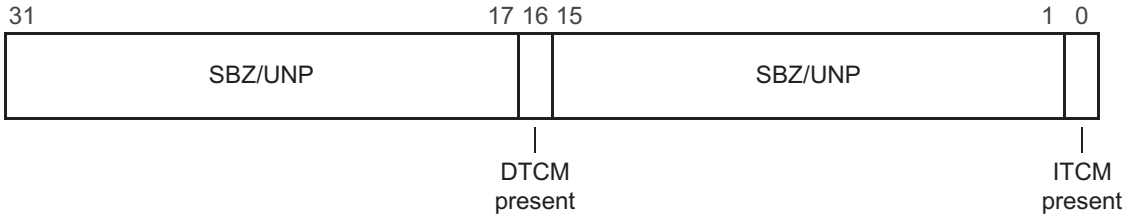


Figure 2-4 TCM Status Register format

2.3.2 Control Register c1

Register c1 is the Control Register for the ARM926EJ-S processor. This register specifies the configuration used to enable and disable the caches and MMU. It is recommended that you access this register using a read-modify-write sequence.

For both reading and writing, the CRm and Opcode_2 fields Should Be Zero. To read and write this register, use the instructions:

```
MRC p15, 0, <Rd>, c1, c0, 0 ; read control register
```

```
MCR p15, 0, <Rd>, c1, c0, 0 ; write control register
```

All defined control bits are set to zero on reset except the V bit and the B bit. The V bit is set to zero at reset if the **VINITHI** signal is LOW, or one if the **VINITHI** signal is HIGH. The B bit is set to zero at reset if the **BIGENDINIT** signal is LOW, or one if the **BIGENDINIT** signal is HIGH.

Figure 2-5 shows the format of the Control Register.

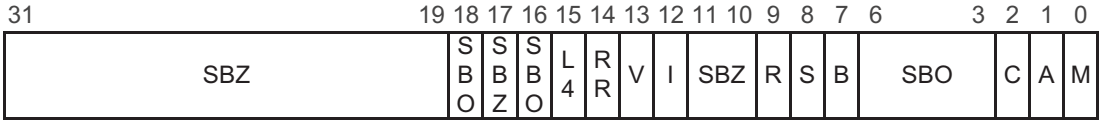


Figure 2-5 Control Register format

Table 2-11 describes the functions of the Control Register bits.

Table 2-11 Control bit functions register c1

Bit	Name	Function
[31:19]	-	Reserved. When read returns an Unpredictable value. When written Should Be Zero, or a value read from bits [31:19] on the same processor. Using a read-modify-write sequence when modifying this register provides the greatest future compatibility.
[18]	-	Reserved, SBO. Read = 1, write = 1.
[17]	-	Reserved, SBZ. Read = 0, write = 0.
[16]	-	Reserved, SBO. Read = 1, write = 1.
[15]	L4 bit	Determines if the T bit is set when load instructions change the PC: 0 = loads to PC set the T bit 1 = loads to PC do not set T bit (ARMv4 behavior). For more details see the *ARM Architecture Reference Manual*.
[14]	RR bit	Replacement strategy for ICache and DCache: 0 = Random replacement 1 = Round-robin replacement.

Table 2-11 Control bit functions register c1 (continued)

Bit	Name	Function
[13]	V bit	Location of exception vectors: 0 = Normal exception vectors selected, address range = `0x0000 0000` to `0x0000 001C` 1 = High exception vectors selected, address range = `0xFFFF 0000` to `0xFFFF 001C`. Set to the value of **VINITHI** on reset.
[12]	I bit	ICache enable/disable: 0 = ICache disabled 1 = ICache enabled.
[11:10]	-	SBZ.
[9]	R bit	ROM protection. This bit modifies the ROM protection system. See *Domain access control* on page 3-24.
[8]	S bit	System protection. This bit modifies the MMU protection system. See *Domain access control* on page 3-24.
[7]	B bit	Endianness: 0 = Little-endian operation 1 = Big-endian operation. Set to the value of **BIGENDINIT** on reset.
[6:3]	-	Reserved. SBO.
[2]	C bit	DCache enable/disable: 0 = Cache disabled 1 = Cache enabled.
[1]	A bit	Alignment fault enable/disable: 0 = Data address alignment fault checking disabled 1 = Data address alignment fault checking enabled.
[0]	M bit	MMU enable/disable: 0 = disabled 1 = enabled.

Effects of Control Register on caches

The bits of the Control Register that directly affect the ICache and DCache behavior are:

- the M bit
- the C bit
- the I bit

- the RR bit.

Assuming that TCM regions are disabled, the caches behave as shown in Table 2-12.

Table 2-12 Effects of Control Register on caches

Cache	MMU	Behavior
ICache disabled	Enabled or disabled	All instruction fetches are from external memory (AHB).
ICache enabled	Disabled	All instruction fetches are cachable, with no protection checks. All addresses are flat mapped. That is VA = MVA = PA.
ICache enabled	Enabled	Instruction fetches are cachable or noncachable, and protection checks are performed. All addresses are remapped from VA to PA, depending on the MMU page table entry. That is, VA translated to MVA, MVA remapped to PA.
DCache disabled	Enabled or disabled	All data accesses are to external memory (AHB).
DCache enabled	Disabled	All data accesses are noncachable nonbufferable. All addresses are flat mapped. That is VA = MVA = PA.
DCache enabled	Enabled	All data accesses are cachable or noncachable, and protection checks are performed. All addresses are remapped from VA to PA, depending on the MMU page table entry. That is, VA translated to MVA, MVA remapped to PA.

If either the DCache or the ICache is disabled, then the contents of that cache are not accessed. If the cache is subsequently re-enabled, the contents will not have changed. To guarantee that memory coherency is maintained, the DCache must be cleaned of dirty data before it is disabled.

Effects of the Control Register on TCM interface

The M bit of the Control Register, when combined with the En bit in the respective TCM region register c9, directly affects the TCM interface behavior, as shown in Table 2-13.

Table 2-13 Effects of Control Register on TCM interface

TCM	MMU	Cache	Behavior
Instruction TCM disabled	Disabled	ICache disabled	All instruction fetches are from the external memory (AHB).
Instruction TCM enabled	Disabled	ICache disabled	All instruction fetches are from the TCM interface, or from external memory (AHB), depending on the setting of the base address in the instruction TCM region register. No protection checks are made. All addresses are flat mapped. That is, VA = MVA= PA.
Instruction TCM enabled	Disabled	ICache enabled	All instruction fetches are from the TCM interface, or from the ICache, depending on the setting of the base address in the Instruction TCM region register. No protection checks are made. All addresses are flat mapped. That is, VA = MVA= PA.
Instruction TCM enabled	Enabled	ICache enabled	All instruction fetches are from the TCM interface, or from the ICache/AHB interface, depending on the setting of the base address in the Instruction TCM region register. Protection checks are made. All addresses are remapped from VA to PA, depending on the page entry. That is, the VA is translated to an MVA, and the MVA is remapped to a PA.
Data TCM disabled	Disabled	DCache disabled	All data accesses are to external memory (AHB).
Data TCM enabled	Disabled	DCache disabled	All data accesses are to the TCM interface, or to the external memory, depending on the setting of the base address in the data TCM region register. No protection checks are made. All addresses are flat mapped. That is, VA = MVA= PA.
Data TCM enabled	Disabled	DCache enabled	All data accesses are to the TCM interface or to external memory, depending on the setting of the base address in the data TCM region register. All addresses are flat mapped. That is, VA =MVA = PA.
Data TCM enabled	Enabled	DCache enabled	All data accesses are either from the TCM interface, or from the DCache/AHB interface, depending on the setting of the base address in the data TCM region register. Protection checks are made. All addresses are remapped from VA to PA, depending on the page entry. That is the VA is translated to an MVA, and the MVA is remapped to a PA.

Note

Read accesses on the TCM interface are not prevented when an ARM9EJ-S core memory access is aborted. All reads on the TCM interface must be treated as speculative. ARM92EJ-S processor write accesses that are aborted do not take place on the TCM interface.

2.3.3 Translation Table Base Register c2

Register c2 is the *Translation Table Base Register* (TTBR), for the base address of the first-level translation table.

Reading from c2 returns the pointer to the currently active first-level translation table in bits [31:14] and an Unpredictable value in bits [13:0].

Writing to register c2 updates the pointer to the first-level translation table from the value in bits [31:14] of the written value. Bits [13:0] Should Be Zero.

You can use the following instructions to access the TTBR:

```
MRC p15, 0, <Rd>, c2, c0, 0; read TTBR
MCR p15, 0, <Rd>, c2, c0, 0; write TTBR
```

The CRm and Opcode_2 fields Should Be Zero when writing to c2.

Figure 2-6 shows the format of the Translation Table Base Register.

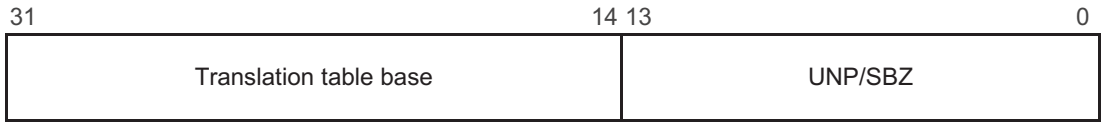


Figure 2-6 TTBR format

2.3.4 Domain Access Control Register c3

Register c3 is the Domain Access Control Register consisting of 16 two-bit fields as shown in Figure 2-7 on page 2-18.

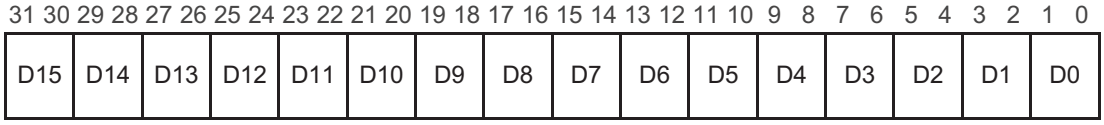


Figure 2-7 Register c3 format

Each two-bit field defines the access permissions for one of the 16 domains (D15-D0) (see Table 2-14).

Reading from c3 returns the value of the Domain Access Control Register.

Writing to c3 writes the value of the Domain Access Control Register.

Table 2-14 Domain access control defines

Value	Meaning	Description
00	No access	Any access generates a domain fault.
01	Client	Accesses are checked against the access permission bits in the section or page descriptor.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are not checked against the access permission bits so a permission fault cannot be generated.

You can use the following instructions to access the Domain Access Control Register:

```
MRC p15, 0, <Rd>, c3, c0, 0 ; read domain access permissions
MCR p15, 0, <Rd>, c3, c0, 0 ; write domain access permissions
```

2.3.5 Register c4

Accessing (reading or writing) this register causes Unpredictable behavior.

2.3.6 Fault Status Registers c5

Register c5 accesses the *Fault Status Registers* (FSRs). The FSRs contain the source of the last instruction or data fault. The instruction-side FSR is intended for debug purposes only. The FSR is updated for alignment faults, and external aborts that occur while the MMU is disabled.

The FSR accessed is determined by the value of the Opcode_2 field:

Opcode_2 = 0 *Data Fault Status Register* (DFSR).

Opcode_2 = 1 *Instruction Fault Status Register* (IFSR).

The fault type encoding is listed in Table 3-9 on page 3-22.

You can access the FSRs using the following instructions:

```
MRC p15, 0, <Rd>, c5, c0, 0 ;read DFSR
MCR p15, 0, <Rd>, c5, c0, 0 ;write DFSR
MRC p15, 0, <Rd>, c5, c0, 1 ;read IFSR
MCR p15, 0, <Rd>, c5, c0, 1 ;write IFSR
```

The format of the Fault Status Register is shown in Figure 2-8.

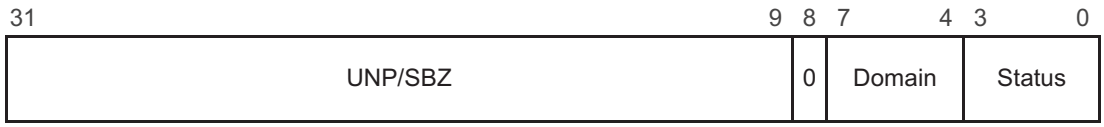


Figure 2-8 FSR format

Table 2-15 shows the bit field descriptions for the FSR.

Table 2-15 FSR bit field descriptions

Bits	Description
[31:9]	UNP/SBZP.
[8]	Always reads as zero. Writes ignored.
[7:4]	Specifies which of the 16 domains (D15-D0) was being accessed when a data fault occurred.
[3:0]	Type of fault generated (see Table 2-16 on page 2-20).

Table 2-16 shows the encodings used for the status field in the FSR, and if the Domain field contains valid information. See *Fault address and fault status registers* on page 3-21 for details of MMU aborts.

Table 2-16 FSR status field encoding

Priority	Source	Size	Status	Domain
Highest	Alignment	-	b00x1	Invalid
	External abort on translation	First level Second level	b1100 b1110	Invalid Valid
	Translation	Section Page	b0101 b0111	Invalid Valid
	Domain	Section Page	b1001 b1011	Valid Valid
	Permission	Section Page	b1101 b1111	Valid Valid
Lowest	External abort	Section or page	b10x0	Invalid

2.3.7 Fault Address Register c6

Register c6 accesses the *Fault Address Register* (FAR). The FAR contains the Modified Virtual Address of the access being attempted when a Data Abort occurred. The FAR is only updated for Data Aborts, not for Prefetch Aborts. The FAR is updated for alignment faults, and external aborts that occur while the MMU is disabled.

You can use the following instructions to access the FAR:

```
MRC p15, 0, <Rd>, c6, c0, 0 ; read FAR
MCR p15, 0, <Rd>, c6, c0, 0 ; write FAR
```

Writing c6 sets the FAR to the value of the data written. This is useful for a debugger to restore the value of the FAR to a previous state.

The CRm and Opcode_2 fields Should Be Zero when reading or writing CP15 c6.

2.3.8 Cache Operations Register c7

Register c7 controls the caches and the write buffer. The function of each cache operation is selected by the Opcode_2 and CRm fields in the MCR instruction used to write to CP15 c7. Writing other Opcode_2 or CRm values is Unpredictable.

Reading from CP15 c7 is Unpredictable, with the exception of the two test and clean operations (see Table 2-18 on page 2-22 and *Test and clean operations* on page 2-24).

You can use the following instruction to write to c7:

```
MCR p15, <Opcode_1>, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

The cache functions, and a description of each function, provided by this register are listed in Table 2-17.

Table 2-17 Function descriptions register c7

Function	Description
Invalidate cache	Invalidates all cache data, including any dirty data.
Invalidate single entry using either index or modified virtual address	Invalidates a single cache line, discarding any dirty data.
Clean single data entry using either index or modified virtual address	Writes the specified DCache line to main memory if the line is marked valid and dirty. The line is marked as not dirty. The valid bit is unchanged.
Clean and invalidate single data entry using either index or modified virtual address	Writes the specified DCache line to main memory if the line is marked valid and dirty. The line is marked not valid.
Test and clean DCache	Tests a number of cache lines, and cleans one of them if any are dirty. Returns the overall dirty state of the cache in bit 30. See *Test and clean operations* on page 2-24.
Test, clean, and invalidate DCache	As for test and clean, except that when the entire cache has been tested and cleaned, it is invalidated. See *Test and clean operations* on page 2-24.

Table 2-17 Function descriptions register c7 (continued)

Function	Description
Prefetch ICache line	Performs an ICache lookup of the specified modified virtual address. If the cache misses, and the region is cachable, a linefill is performed.
Drain write buffer	This instruction acts as an explicit memory barrier. It drains the contents of the write buffers of all memory stores occurring in program order before this instruction is completed. No instructions occurring in program order after this instruction are executed until it completes. This can be used when timing of specific stores to the level two memory system has to be controlled (for example, when a store to an interrupt acknowledge location has to complete before interrupts are enabled).
Wait for interrupt	This instruction drains the contents of the write buffers, puts the processor into a low-power state, and stops it from executing further instructions until an interrupt (or debug request) occurs. When an interrupt does occur, the MCR instruction completes and the IRQ or FIQ handler is entered as normal. The return link in R14_irq or R14_fiq contains the address of the MCR instruction plus eight, so that the typical instruction used for interrupt return (`SUBS PC,R14,#4`) returns to the instruction following the MCR.

Table 2-18 lists the cache operation functions and the associated data and instruction formats for c7.

Table 2-18 Cache operations c7

Function/operation	Data format	Instruction
Invalidate ICache and DCache	SBZ	`MCR p15, 0, <Rd>, c7, c7, 0`
Invalidate ICache	SBZ	`MCR p15, 0, <Rd>, c7, c5, 0`
Invalidate ICache single entry (MVA)	MVA	`MCR p15, 0, <Rd>, c7, c5, 1`
Invalidate ICache single entry (Set/Way)	Set/Way	`MCR p15, 0, <Rd>, c7, c5, 2`
Prefetch ICache line (MVA)	MVA	`MCR p15, 0, <Rd>, c7, c13, 1`
Invalidate DCache	SBZ	`MCR p15, 0, <Rd>, c7, c6, 0`
Invalidate DCache single entry (MVA)	MVA	`MCR p15, 0, <Rd>, c7, c6, 1`

Table 2-18 Cache operations c7 (continued)

Function/operation	Data format	Instruction
Invalidate DCache single entry (Set/Way)	Set/Way	`MCR p15, 0, <Rd>, c7, c6, 2`
Clean DCache single entry (MVA)	MVA	`MCR p15, 0, <Rd>, c7, c10, 1`
Clean DCache single entry (Set/Way)	Set/Way	`MCR p15, 0, <Rd>, c7, c10, 2`
Test and clean DCache	-	`MRC p15, 0, <Rd>, c7, c10, 3`
Clean and invalidate DCache entry (MVA)	MVA	`MCR p15, 0, <Rd>, c7, c14, 1`
Clean and invalidate DCache entry (Set/Way)	Set/Way	`MCR p15, 0, <Rd>, c7, c14, 2`
Test, clean, and invalidate DCache	-	`MRC p15, 0, <Rd>, c7, c14, 3`
Drain write buffer	SBZ	`MCR p15, 0, <Rd>, c7, c10, 4`
Wait for interrupt	SBZ	`MCR p15, 0, <Rd>, c7, c0, 4`

The MVA format for Rd for the CP15 c7 MCR operations is shown in Figure 2-9. The Tag, Set, and Word fields define the MVA. For all of the cache operations, Word Should Be Zero.

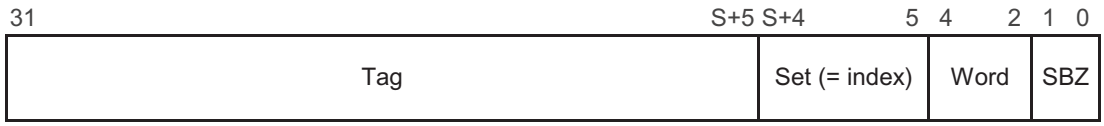


Figure 2-9 Register c7 MVA format

The Set/Way format for Rd for the CP15 c7 MCR operations is shown in Figure 2-10 on page 2-24, where A and S are the base-two logarithms of the associativity and the number of sets. The Set, Way, and Word fields define the format. For all of the cache operations, Word Should Be Zero.

For a 16KB cache, 4-way set associative, 8-word line, then:

- $A = \log_2$ associativity $= \log_2 4 = 2$
- $S = \log_2$ NSETS where:

  NSETS= cache size in bytes/associativity/line length in bytes:

  NSETS= 16384/4/32 = 128

  Therefore:

  $S = \log_2 128 = 7$

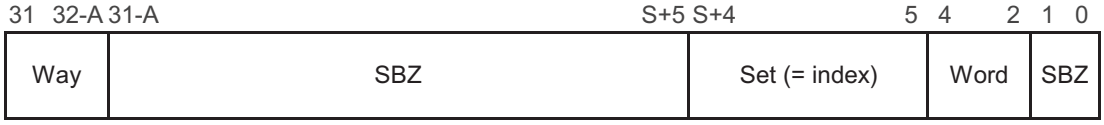


Figure 2-10 Register c7 Set/Way format

Test and clean operations

The test and clean DCache instruction provides an efficient way to clean the entire DCache using a simple loop. The test and clean DCache instruction tests a number of lines in the DCache to determine if any of them are dirty. If any dirty lines are found, then one of those lines is cleaned. The test and clean DCache instruction also returns the status of the entire DCache in bit 30.

Note

The test and clean DCache instruction, `MRC p15, 0, r15, c7, c10, 3`, is a special encoding that uses r15 as a destination operand. However, the PC is not changed by using this instruction. This MRC instruction also sets the condition code flags.

If the cache contains any dirty lines, bit 30 is set to 0. If the cache contains no dirty lines, bit 30 is set to 1. This means that you can use the following loop to clean the entire DCache:

```
tc_loop:    MRC p15, 0, r15, c7, c10, 3     ; test and clean
            BNE tc_loop
```

The test, clean, and invalidate DCache instruction is the same as test and clean DCache, except that when the entire cache has been cleaned, it is invalidated. This means that you can use the following loop to clean and invalidate the entire DCache:

```
tci_loop:   MRC p15, 0, r15, c7, c14, 3     ; test clean and invalidate
            BNE tci_loop
```

2.3.9 TLB Operations Register c8

This is a write-only register used to control the *Translation Lookaside Buffer* (TLB). There is a single TLB used to hold entries for both data and instructions. The TLB is divided into two parts:

- a set-associative part
- a fully-associative part.

The fully-associative part (also referred to as the lockdown part of the TLB) is used to store entries to be locked down. Entries held in the lockdown part of the TLB are preserved during an invalidate TLB operation. Entries can be removed from the lockdown TLB using an invalidate TLB single entry operation.

Six TLB operations are defined, and the function to be performed is selected by the Opcode_2 and CRm fields in the MCR instruction used to write CP15 c8. Writing other Opcode_2 or CRm values is Unpredictable. Reading from this register is Unpredictable.

You can use the instructions shown in Table 2-19 to perform TLB operations.

Table 2-19 Register c8 TLB operations

ARMv4/ARMv5 operation	ARM926EJ-S operation	Data	Instruction
Invalidate TLB	Invalidate set-associative TLB	SBZ	`MCR p15, 0, <Rd>, c8, c7, 0`
Invalidate TLB single entry (MVA)	Invalidate single entry	MVA	`MCR p15, 0, <Rd>, c8, c7, 1`
Invalidate instruction TLB	Invalidate set-associative TLB	SBZ	`MCR p15, 0, <Rd>, c8, c5, 0`
Invalidate instruction TLB single entry (MVA)	Invalidate single entry	MVA	`MCR p15, 0, <Rd>, c8, c5, 1`
Invalidate data TLB	Invalidate set-associative TLB	SBZ	`MCR p15, 0, <Rd>, c8, c6, 0`
Invalidate data TLB single entry (MVA)	Invalidate single entry	MVA	`MCR p15, 0, <Rd>, c8, c6, 1`

Those instructions that are intended to be used with dual TLB implementations (such as the ARM920T core or the ARM1020T core) apply to any entry, regardless of the type of access that caused the entry to be loaded into the TLB (see the *ARM Architecture Reference Manual*).

The invalidate TLB operations invalidate all the unpreserved entries in the TLB. The invalidate TLB single entry operations invalidate any TLB entry corresponding to the Modified Virtual Address given in Rd, regardless of its preserved state. See *TLB Lockdown Register c10* on page 2-32 for a description of how to preserve entries in the TLB.

Figure 2-11 on page 2-26 shows the Modified Virtual Address format used for invalidate TLB single entry operations.

Figure 2-11 Register c8 MVA format

Note

If either small or large pages are used, and these pages contain subpage access permissions that are different, then you must use four invalidate TLB single entry operations, with the MVA set to each subpage, to invalidate all information related to that page held in a TLB.

2.3.10 Cache Lockdown and TCM Region Registers c9

Register c9 accesses the Cache Lockdown and TCM Region Registers. The register accessed is determined by the value of the CRm field:

CRm = c0 selects the Cache Lockdown Register

CRm = c1 selects the TCM Region Register.

Other values of CRm are reserved.

Cache Lockdown Register c9

The Cache Lockdown Register uses a cache-way-based locking scheme (Format C) that enables you to control each cache way independently.

These registers enable you to control which cache ways of the four-way cache are used for the allocation on a linefill. When the registers are defined, subsequent linefills are only placed in the specified target cache way. This gives you some control over the cache pollution caused by particular applications, and provides a traditional lockdown operation for locking critical code into the cache.

A locking bit for each cache way determines if the normal cache allocation is allowed to access that cache way. See Table 2-21 on page 2-28.

A maximum of three cache ways of the four-way associative cache can be locked, ensuring that normal cache line replacement is performed.

Note

If no cache ways have L bits set to 0, then cache way 3 is used for all linefills.

The first four bits of this register determine the L bit for the associated cache way. The Opcode_2 field of the MRC or MCR instruction determines whether the instruction or data lockdown register is accessed:

Opcode_2 = 0 Selects the DCache lockdown register.

Opcode_2 = 1 Selects the ICache lockdown register.

You can use the instructions shown in Table 2-20 to access the Cache Lockdown Register.

Table 2-20 Cache Lockdown Register instructions

Function	Data	Instruction
Read DCache Lockdown Register	L bits	`MRC p15,0,<Rd>,c9,c0,0`
Write DCache Lockdown Register	L bits	`MCR p15,0,<Rd>,c9,c0,0`
Read ICache Lockdown Register	L bits	`MRC p15,0,<Rd>,c9,c0,1`
Write ICache Lockdown Register	L bits	`MCR p15,0,<Rd>,c9,c0,1`

You must only modify the Cache Lockdown Register using a read-modify-write sequence. For example:

```
MRC p15, 0, <Rn>, c9, c0, 1 ;
ORR <Rn>, <Rn>, 0x01 ;
MCR p15, 0, <Rn>, c9, c0, 1 ;
```

This sequence sets the L bit to 1 for way 0 of the ICache. The format of the cache lockdown register c9 is shown in Figure 2-12.

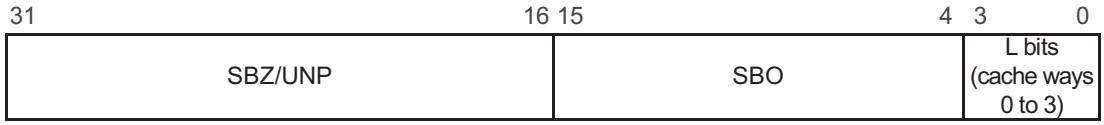


Figure 2-12 Cache Lockdown Register c9 format

The format of the Cache Lockdown Register L bits is shown in Table 2-21. All cache ways are available for allocation from reset.

Table 2-21 Cache Lockdown Register L bits

Bits	4-way associative	Notes
[31:16]	UNP/SBZP	Reserved
[15:4]	0xFFF	SBO
3	L bit for Way 3	Bits[3:0] are the L bits for each cache way: 0 = Allocation to the cache way is determined by the standard replacement algorithm (reset state) 1 = No allocation is performed to this cache way.
2	L bit for Way 2	
1	L bit for Way 1	
0	L bit for Way 0	

You can use the cache lockdown and cache unlock procedures described in:

- *Specific loading of addresses into a cache way*
- *Cache unlock procedure* on page 2-29.

Specific loading of addresses into a cache way

The procedure to lock down code and data into way *i* of a cache with *N* ways using Format C involves making it impossible to allocate to any cache way other than the target cache way:

1. Ensure that no processor exceptions can occur during the execution of this procedure, for example by disabling interrupts. If this is not possible, all code and data used by any exception handlers must be treated as code and data as in steps 2 and 3.
2. If an ICache way is being locked down, ensure that all the code executed by the lockdown procedure is in an uncachable area of memory (including TCM) or in an already locked cache way.
3. If a DCache way is being locked down, ensure that all data used by the lockdown procedure is in an uncachable area of memory (including TCM) or is in an already locked cache way.
4. Ensure that the data/instructions that are to be locked down are in a cachable area of memory.
5. Ensure that the data/instructions that are to be locked down are not already in the cache. Use the register c7 clean and/or invalidate operations to ensure this.
6. Write to register c9, CRm == 0, setting L==0 for bit *i* and L==1 for all other ways. This enables allocation to the target cache way.

7. For each of the cache lines to be locked down in cache way *i*:
   - If a DCache is being locked down, use an LDR instruction to load a word from the memory cache line to ensure that the memory cache line is loaded into the cache.
   - If an ICache is being locked down, use the register c7 MCR prefetch ICache line (CRm == c13, Opcode2 == 1) to fetch the memory cache line into the cache.
8. Write to register c9, CRm == 0 setting L == 1 for bit *i* and restoring all the other bits to the values they had before the lockdown routine was started.

Cache unlock procedure

To unlock the locked down portion of the cache, write to register c9 setting L == 0 for the appropriate bit. For example, the following sequence sets the L bit to 0 for way 0 of the ICache, unlocking way 0:

```
MRC p15, 0, <Rn>, c9, c0, 1;
BIC <Rn>, <Rn>, 0x01 ;
MCR p15, 0, <Rn>, c9, c0, 1;
```

TCM Region Register c9

The ARM926EJ-S processor supports physically-indexed, physically-tagged TCM. The TCM Region Register supports one region of instruction TCM and one region of data TCM. The minimum size of TCM region that can be supported is 4KB. The TCM Status Register indicates if TCM memories are attached (see *TCM Status Register c0* on page 2-12). The size of each TCM region is defined by the **DRSIZE** and **IRSIZE** input pins.

The data TCM is always disabled at reset. The instruction TCM is enabled at reset if the **INITRAM** pin is HIGH. This enables booting from the instruction TCM and sets the ITCM enable bit in the ITCM region register. You can use the TCM Region Register instructions listed in Table 2-22.

Table 2-22 TCM Region Register instructions

Function	Data	Instruction
Read data TCM Region Register	Base address	`MRC p15,0,<Rd>,c9,c1,0`
Write data TCM Region Register	Base address	`MCR p15,0,<Rd>,c9,c1,0`
Read instruction TCM Region Register	Base address	`MRC p15,0,<Rd>,c9,c1,1`
Write instruction TCM Region Register	Base address	`MCR p15,0,<Rd>,c9,c1,1`

The TCM Region Register format is shown in Figure 2-13.

Figure 2-13 TCM Region Register c9 format

Table 2-23 shows the bit assignments for the TCM Region Register.

Table 2-23 TCM Region Register c9

Bits	Function
[31:12]	Base address (physical address).
[11:6]	SBZ/UNP.
[5:2]	Size. The Size field reflects the value of the **IRSIZE/DRSIZE** macrocell inputs. The Size field encoding is shown in Table 2-24.
[1]	SBZ/UNP.
[0]	Enable bit: 0 = disabled 1 = enabled.

Table 2-24 TCM Size field encoding

Memory size	Value
0KB/absent	b0000
Reserved	b0001, b0010
4KB	b0011
8KB	b0100
16KB	b0101
32KB	b0110

Table 2-24 TCM Size field encoding (continued)

Memory size	Value
64KB	b0111
128KB	b1000
256KB	b1001
512KB	b1010
1MB	b1011
Reserved	b1100, b1101, b1110, b1111

If either the data or instruction TCM is disabled, then the contents of the respective TCM are not accessed. If the TCM is subsequently re-enabled, the contents will not have been changed by the ARM926EJ-S processor.

For a Harvard arrangement, the instruction-side TCM must be accessible for both reads and writes during normal operation, and for loading code, or for debug activity. This enables accesses to literal pools, undefined instruction emulation, and parameter passing for SWI operations. You must insert an *Instruction Memory Barrier* (IMB) between a write to the instruction TCM and the instructions being read from the instruction TCM. See Chapter 9 *Instruction Memory Barrier* for more details.

Note

Instruction fetches from the data TCM are not possible. An attempt to fetch an instruction from an address in the data TCM space does not result in an access to the data TCM, and the instruction is fetched from main memory. These accesses can result in external aborts, because the address range might not be supported in main memory.

The instruction TCM must not be programmed to the same base address as the data TCM. If the two TCMs are of different sizes, the regions in physical memory must not overlap. If they do overlap, it is Unpredictable which memory is accessed.

Note

The base address value setting must be aligned to the TCM size.

2.3.11 TLB Lockdown Register c10

The TLB Lockdown Register controls where hardware page table walks place the TLB entry, in the set associative region or the lockdown region of the TLB, and if in the lockdown region, which entry is written. The lockdown region of the TLB contains eight entries. See *TLB structure* on page 3-31 for a description of the structure of the TLB.

Writing the TLB Lockdown Register with the preserve bit (P bit) set to:

1 Means subsequent hardware page table walks place the TLB entry in the lockdown region at the entry specified by the victim, in the range 0 to 7.

0 Means subsequent hardware page table walks place the TLB entry in the set associative region of the TLB.

TLB entries in the lockdown region are preserved so that invalidate TLB operations only invalidate the unpreserved entries in the TLB. That is, those in the set-associative region. Invalidate TLB single entry operations invalidate any TLB entry corresponding to the Modified Virtual Address given in Rd, regardless of their preserved state. That is, if they are in the lockdown or set-associative regions of the TLB. See *TLB Operations Register c8* on page 2-24 for a description of the TLB invalidate operations.

The instructions you can use to program the TLB Lockdown Register are shown in Table 2-25.

Table 2-25 Programming the TLB Lockdown Register

Function	Instruction
Read data TLB lockdown victim	`MRC p15,0,<Rd>,c10,c0,0`
Write data TLB lockdown victim	`MCR p15,0,<Rd>,c10,c0,0`

Figure 2-14 shows the TLB Lockdown Register format.

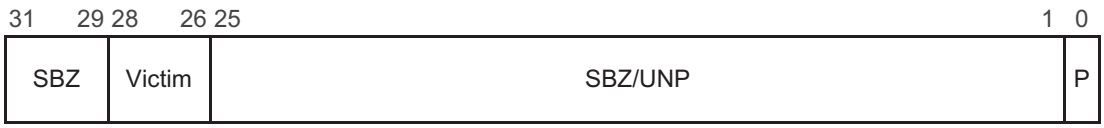


Figure 2-14 TLB Lockdown Register format

The victim automatically increments after any table walk that results in an entry being written into the lockdown part of the TLB.

Note

It is not possible for a lockdown entry to entirely map either small or large pages, unless all the subpage access permissions are identical. Entries can still be written into the lockdown region, but the address range that is mapped only covers the subpage corresponding to the address that was used to perform the page table walk.

Example 2-1 is a code sequence that locks down an entry to the current victim.

Example 2-1 Lock down an entry to the current victim

```
ADR r1,LockAddr         ; set r1 to the value of the address to be locked down
MCR p15,0,r1,c8,c7,1    ; invalidate TLB single entry to ensure that
                        ; LockAddr is not already in the TLB
MRC p15,0,r0,c10,c0,0   ; read the lockdown register
ORR r0,r0,#1            ; set the preserve bit
MCR p15,0,r0,c10,c0,0   ; write to the lockdown register
LDR r1,[r1]             ; TLB will miss, and entry will be loaded
MRC p15,0,r0,c10,c0,0   ; read the lockdown register (victim will have
                        ; incremented)
BIC r0,r0,#1            ; clear preserve bit
MCR p15,0,r0,c10,c0,0   ; write to the lockdown register
```

2.3.12 Register c11 and c12

Accessing (reading or writing) these registers causes Unpredictable behavior.

2.3.13 Process ID Register c13

Register c13 accesses the process identifier registers. The register accessed depends on the value of the Opcode_2 field:

Opcode_2 = 0 Selects the *Fast Context Switch Extension* (FCSE) *Process Identifier* (PID) Register.

Opcode_2 = 1 Selects the Context ID Register.

You can use the process ID register to determine the process that is currently running. The process identifier is set to 0 at reset.

FCSE PID Register

Addresses issued by the ARM9EJ-S core in the range 0 to 32MB are translated in accordance with the value contained in this register. Address A becomes A + (FCSE PID x 32MB). It is this modified address that is seen by the caches, MMU, and TCM interface. Addresses above 32MB are not modified. The FCSE PID is a seven-bit field, enabling 128 x 32MB processes to be mapped.

If the FCSE PID is 0, there is a flat mapping between the virtual addresses output by the ARM9EJ-S core and the modified virtual addresses used by the caches, MMU, and TCM interface. The FCSE PID is set to 0 at system reset.

If the MMU is disabled, then no FCSE address translation occurs.

FCSE translation is not applied for addresses used for entry based cache or TLB maintenance operations. For these operations VA = MVA.

Table 2-26 shows the ARM instructions that can be used to access the FCSE PID Register.

Table 2-26 FCSE PID Register operations

Function	Data	ARM Instruction
Read FCSE PID	FCSE PID	`MRC p15,0,<Rd>,c13,c0, 0`
Write FCSE PID	FCSE PID	`MCR p15,0,<Rd>,c13,c0, 0`

The format of the FCSE PID Register is shown in Figure 2-15.

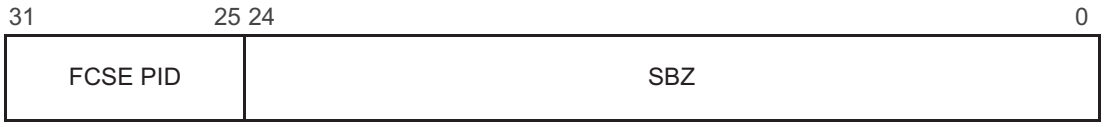


Figure 2-15 Process ID Register format

Performing a fast context switch

You can perform a fast context switch by writing to CP15 register c13 with Opcode_2 = 0. The contents of the caches and the TLB do not have to be flushed after a fast context switch because they still hold valid address tags. The two instructions after the FCSE PID has been written have been fetched with the old FCSE PID, as the following code example shows:

```
{FCSE PID = 0}
    MOV r0, #1:SHL:25       ;Fetched with FCSE PID = 0
    MCR p15,0,r0,c13,c0,0   ;Fetched with FCSE PID = 0
    A1                      ;Fetched with FCSE PID = 0
    A2                      ;Fetched with FCSE PID = 0
    A3                      ;Fetched with FCSE PID = 1
```

Where A1, A2, and A3 are the three instructions following the fast context switch.

Context ID Register

The Context ID Register provides a mechanism to allow real-time trace tools to identify the currently executing process in multi-tasking environments.

The contents of this register are replicated on the **ETMPROCID** pins of the ARM926EJ-S processor. **ETMPROCIDWR** is pulsed when a write occurs to the Context ID Register.

Table 2-27 shows the ARM instructions that you can use to access the Context ID Register.

Table 2-27 Context ID register operations

Function	Data	ARM Instruction
Read context ID	Context ID	`MRC p15,0,<Rd>,c13,c0, 1`
Write context ID	Context ID	`MCR p15,0,<Rd>,c13,c0, 1`

The format of the Context ID Register, Rd, transferred during this operation is shown in Figure 2-16.

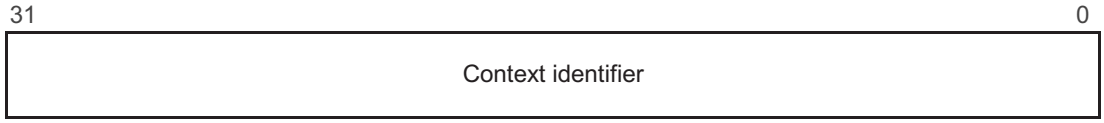


Figure 2-16 Context ID Register format

2.3.14 Register c14

Accessing (reading or writing) this register is reserved.

2.3.15 Test and Debug Register c15

You can use register c15 to provide device-specific test and debug operations in ARM926EJ-S processors. Appendix B *CP15 Test and Debug Registers* describes the registers and functions available using CP15 c15.This register is defined to be reserved for implementation-defined purposes in the *ARM Architecture Reference Manual*. If you write software that uses the device-specific facilities provided by c15, then this software is unlikely to be either backwards or forwards compatible.

Chapter 3
Memory Management Unit

This chapter describes the *Memory Management Unit* (*MMU*). It contains the following sections:

3.1 About the MMU

The ARM926EJ-S MMU is an ARM architecture v5 MMU. It provides virtual memory features required by systems operating on platforms such as Symbian OS, WindowsCE, and Linux. A single set of two-level page tables stored in main memory is used to control the address translation, permission checks, and memory region attributes for both data and instruction accesses.

The MMU uses a single unified *Translation Lookaside Buffer* (TLB) to cache the information held in the page tables.

To support both sections and pages, there are two levels of address translation. The MMU puts the translated physical addresses into the MMU Translation Lookaside Buffer TLB.

The MMU TLB has two parts:

- the main TLB
- the lockdown TLB.

The main TLB is a two-way, set-associative cache for page table information. It has 32 entries per way for a total of 64 entries. The lockdown TLB is an eight-entry fully-associative cache that contains locked TLB entries. Locking TLB entries can ensure that a memory access to a given region never incurs the penalty of a page table walk. For more details of the TLBs see *TLB structure* on page 3-31.

The MMU features are:

- standard ARM architecture v4 and v5 MMU mapping sizes, domains, and access protection scheme
- mapping sizes are 1MB (sections), 64KB (large pages), 4KB (small pages), and 1KB (tiny pages)
- access permissions for large pages and small pages can be specified separately for each quarter of the page (subpage permissions)
- hardware page table walks
- invalidate entire TLB using CP15 c8
- invalidate TLB entry selected by MVA, using CP15 c8
- lockdown of TLB entries using CP15 c10.

The following subsections are:

- *Access permissions and domains* on page 3-3
- *Translated entries* on page 3-3
- *MMU program accessible registers* on page 3-4

3.1.1 Access permissions and domains

For large and small pages, access permissions are defined for each subpage (1KB for small pages, 16KB for large pages). Sections and tiny pages have a single set of access permissions.

All regions of memory have an associated domain. A domain is the primary access control mechanism for a region of memory. It defines the conditions necessary for an access to proceed. The domain determines if:

- access permissions are used to qualify the access
- the access is unconditionally allowed to proceed
- the access is unconditionally aborted.

In the latter two cases, the access permission attributes are ignored.

There are 16 domains. These are configured using the domain access control register (see *Domain Access Control Register c3* on page 2-17).

3.1.2 Translated entries

The main TLB caches 64 translated entries. If, during a memory access, the main TLB contains a translated entry for the MVA, the MMU reads the protection data to detrmine if the access is permitted:

- if access is permitted and an off-chip access is required, the MMU outputs the appropriate physical address corresponding to the MVA
- if access is permitted and an off-chip access is not required, the cache or TCM services the access
- if access is not permitted, the MMU signals the CPU core to abort.

If the TLB misses (it does not contain an entry for the MVA) the translation table walk hardware is invoked to retrieve the translation information from a translation table in physical memory. When retrieved, the translation information is written into the TLB, possibly overwriting an existing value.

To enable use of TLB locking features, the location to be written can be specified using CP15 c10 TLB Lockdown Register.

At reset the MMU is turned off, no address mapping occurs, and all regions are marked as noncachable and nonbufferable.

3.1.3 MMU program accessible registers

Table 3-1 shows the CP15 registers that are used in conjunction with page table descriptors stored in memory to determine the operation of the MMU.

Table 3-1 MMU program-accessible CP15 registers

Register	Bits	Register description
Control register c1	M, A, S, R	Contains bits to enable the MMU (M bit), enable data address alignment checks (A bit), and to control the access protection scheme (S bit and R bit).
Translation table base register c2	[31:14]	Holds the physical address of the base of the translation table maintained in main memory. This base address must be on a 16KB boundary.
Domain access control register c3	[31:0]	Comprises 16 two-bit fields. Each field defines the access control attributes for one of 16 domains (D15 to D0).
Fault status registers, IFSR and DFSR, c5	[7:0]	Indicates the cause of a Data or Prefetch Abort, and the domain number of the aborted access, when an abort occurs. Bits [7:4] specify which of the 16 domains (D15 to D0) was being accessed when a fault occurred. Bits [3:0] indicate the type of access being attempted. The value of all other bits is Unpredictable. The encoding of these bits is shown in Table 3-9 on page 3-22.
Fault address register c6	[31:0]	Holds the MVA associated with the access that caused the Data Abort. See Table 3-9 on page 3-22 for details of the address stored for each type of fault. The ARM9EJ-S register R14_abt holds the VA associated with a Prefetch Abort.
TLB operations register c8	[31:0]	This register is used to perform TLB maintenance operations. These are either invalidating all the (unpreserved) entries in the TLB, or invalidating a specific entry.
TLB lockdown register c10	[28:26] and [0]	Enables specific page table entries to be locked into the TLB. Locking entries in the TLB guarantees that accesses to the locked page or section can proceed without incurring the time penalty of a TLB miss. This enables the execution latency for time-critical pieces of code such as interrupt handlers to be minimized.

All the CP15 MMU registers, except c8, contain state that can be read using MRC instructions, and written using MCR instructions. Registers c5 and c6 are also written by the MMU during an abort. Writing to c8 causes the MMU to perform a TLB operation, to manipulate TLB entries. This register is write-only.

The CP15 registers are described in Chapter 2 *Programmer's Model*.

3.2 Address translation

The VA generated by the CPU core is converted to a *Modified Virtual Address* (MVA) by the FCSE using the value held in CP15 c13. The MMU translates MVAs into physical addresses to access external memory, and also performs access permission checking.

The MMU table-walking hardware is used to add entries to the TLB. The translation information that comprises both the address translation data and the access permission data resides in a translation table located in physical memory. The MMU provides the logic for automatically traversing this translation table and loading entries into the TLB.

The number of stages in the hardware table walking and permission checking process is one or two depending on whether the address is marked as a section-mapped access or a page-mapped access.

There are three sizes of page-mapped accesses and one size of section-mapped access. Page-mapped accesses are for:

- large pages
- small pages
- tiny pages.

The translation process always begins in the same way, with a level one fetch. A section-mapped access requires only a level one fetch, but a page-mapped access requires an additional level two fetch.

The following subsections are:

- *Translation table base* on page 3-6
- *First-level fetch* on page 3-8
- *First-level descriptor* on page 3-8
- *Section descriptor* on page 3-10
- *Coarse page table descriptor* on page 3-11
- *Fine page table descriptor* on page 3-12
- *Translating section references* on page 3-13
- *Second-level descriptor* on page 3-14
- *Translating large page references* on page 3-16
- *Translating small page references* on page 3-18
- *Translating tiny page references* on page 3-19.

3.2.1 Translation table base

The hardware translation process is initiated when the TLB does not contain a translation for the requested MVA. The *Translation Table Base Register* (TTBR), CP15 register c2, points to the base address of a table in physical memory that contains section or page descriptors, or both. The 14 low-order bits [13:0] of the TTBR are Unpredictable on a read, and the table must reside on a 16KB boundary. Figure 3-1 shows the format of the TTBR.

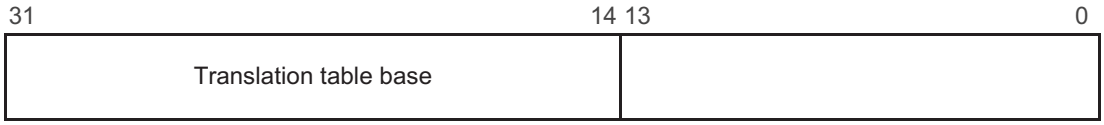


Figure 3-1 Translation Table Base Register

The translation table has up to 4096 x 32-bit entries, each describing 1MB of virtual memory. This enables up to 4GB of virtual memory to be addressed.

Figure 3-2 on page 3-7 shows the table walk process.

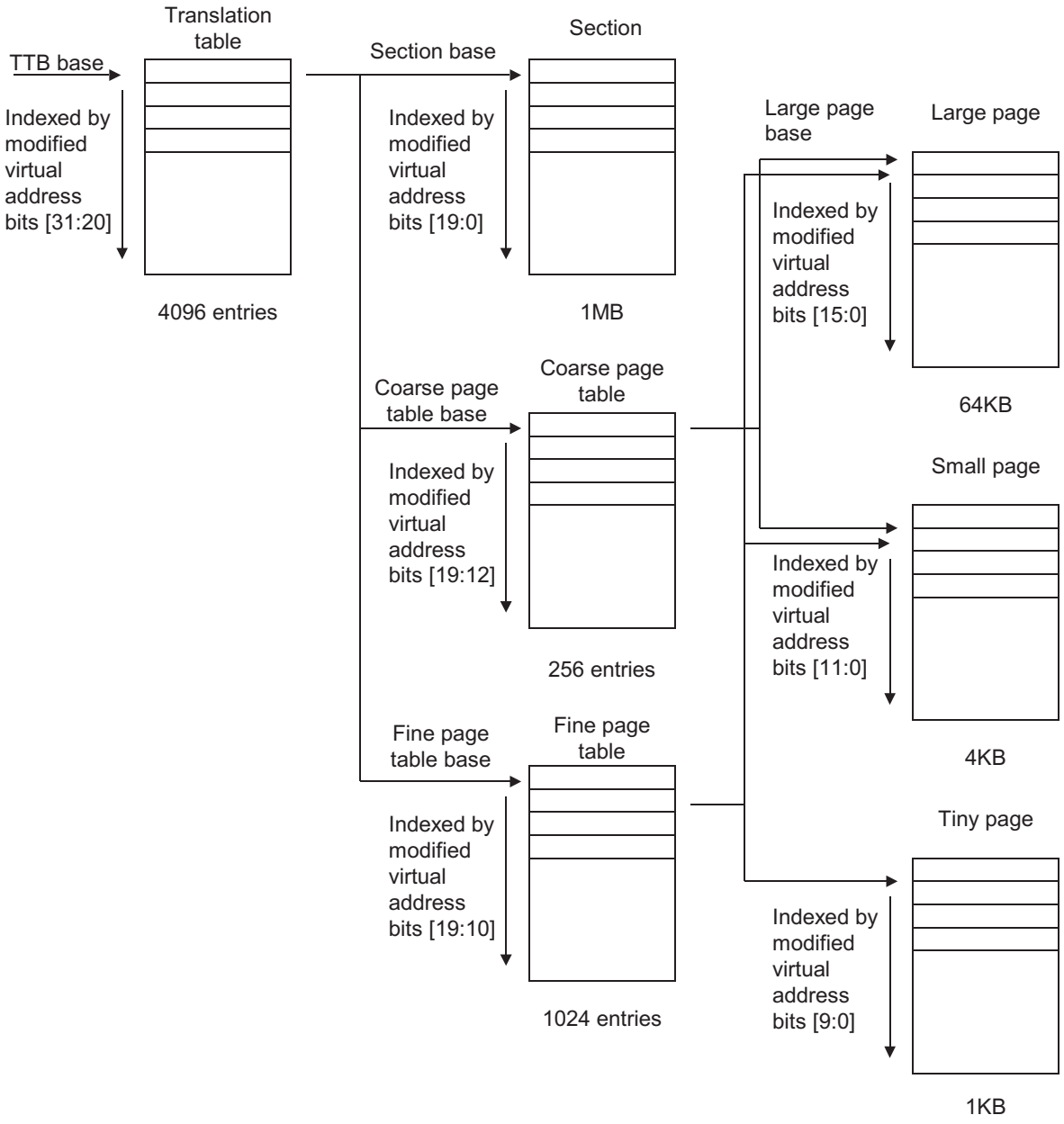


Figure 3-2 Translating page tables

3.2.2 First-level fetch

Bits [31:14] of the TTBR are concatenated with bits [31:20] of the MVA to produce a 30-bit address as shown in Figure 3-3.

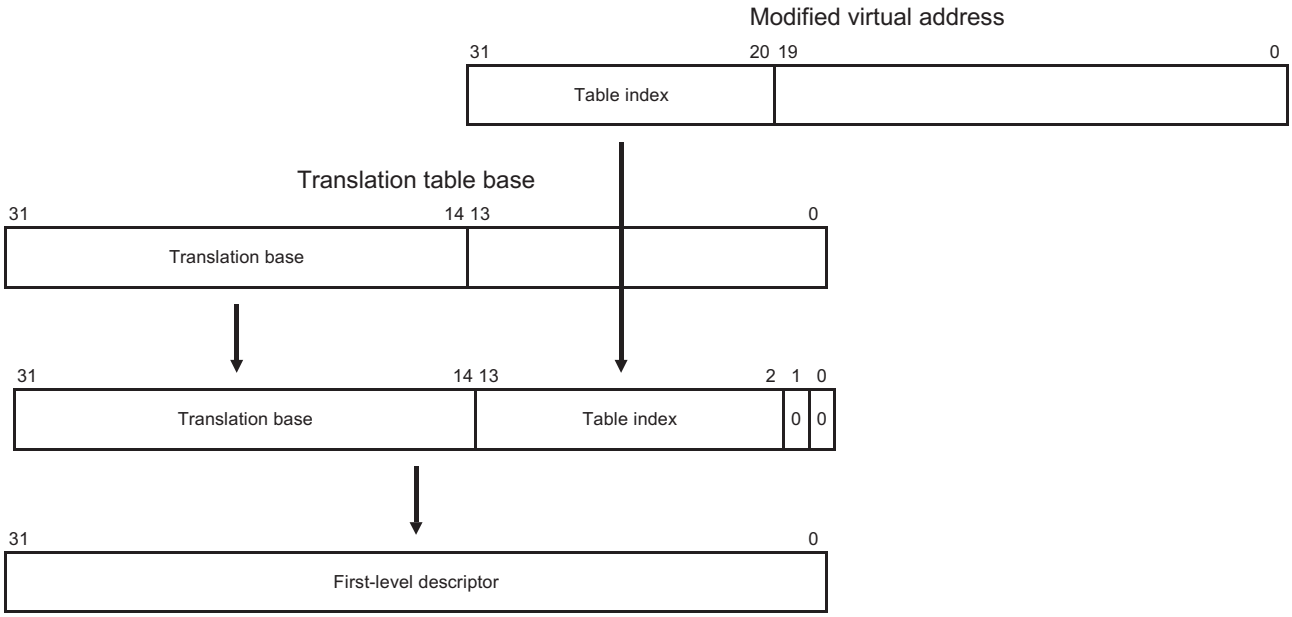


Figure 3-3 Accessing translation table first-level descriptors

This address selects a 4-byte translation table entry. This is a first-level descriptor for either a section or a page table.

3.2.3 First-level descriptor

The first-level descriptor returned is a section descriptor, a coarse page table descriptor, or a fine page table descriptor, or is invalid. Figure 3-4 on page 3-9 shows the format of a first-level descriptor.

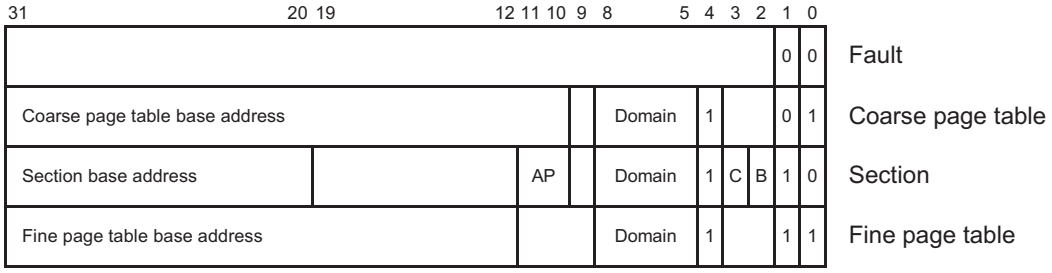


Figure 3-4 First-level descriptor

A section descriptor provides the base address of a 1MB block of memory.

The page table descriptors provide the base address of a page table that contains second-level descriptors. There are two sizes of page table:

- coarse page tables have 256 entries, splitting the 1MB that the table describes into 4KB blocks
- fine page tables have 1024 entries, splitting the 1MB that the table describes into 1KB blocks.

First-level descriptor bit assignments are shown in Table 3-2.

Table 3-2 First-level descriptor bits

Bits			Description
Section	**Coarse**	**Fine**	
[31:20]	[31:10]	[31:12]	These bits form the corresponding bits of the physical address.
[19:12]	-	-	Should Be Zero.
[11:10]	-	-	Access permission bits. *Access permissions and domains* on page 3-3 and *Fault address and fault status registers* on page 3-21 show how to interpret the access permission bits.
[9]	[9]	[11:9]	Should Be Zero.
[8:5]	[8:5]	[8:5]	Domain control bits.
[4]	[4]	[4]	Must be 1.

Table 3-2 First-level descriptor bits (continued)

Bits			Description
Section	Coarse	Fine	
[3:2]	-	-	Bits C and B indicate whether the area of memory mapped by this page is treated as write-back cachable, write-through cachable, noncached buffered, or noncached nonbuffered.
-	[3:2]	[3:2]	Should Be Zero.
[1:0]	[1:0]	[1:0]	These bits indicate the page size and validity and are interpreted as shown in Table 3-3.

The two least significant bits of the first-level descriptor indicate the descriptor type as shown in Table 3-3.

Table 3-3 Interpreting first-level descriptor bits [1:0]

Value	Meaning	Description
0 0	Invalid	Generates a section translation fault
0 1	Coarse page table	Indicates that this is a coarse page table descriptor
1 0	Section	Indicates that this is a section descriptor
1 1	Fine page table	Indicates that this is a fine page table descriptor

3.2.4 Section descriptor

A section descriptor provides the base address of a 1MB block of memory. Figure 3-5 shows the format of a section descriptor.

Figure 3-5 Section descriptor

Section descriptor bit assignments are described in Table 3-4.

Table 3-4 Section descriptor bits

Bits	Description
[31:20]	Form the corresponding bits of the physical address for a section
[19:12]	Always written as 0
[11:10]	The AP bits specify the access permissions for this section
[9]	Always written as 0
[8:5]	Specify one of the 16 possible domains (held in the domain access control register) that contain the primary access controls
[4]	Should be written as 1, for backwards compatibility
[3:2]	These bits (C and B) indicate if the area of memory mapped by this section is treated as write-back cachable, write-through cachable, noncached buffered, or noncached nonbuffered
[1:0]	These bits must be 10 to indicate a section descriptor

3.2.5 Coarse page table descriptor

A coarse page table descriptor provides the base address of a page table that contains second-level descriptors for either large page or small page accesses. Coarse page tables have 256 entries, splitting the 1MB that the table describes into 4KB blocks. Figure 3-6 shows the format of a coarse page table descriptor.

Figure 3-6 Coarse page table descriptor

Note

If a coarse page table descriptor is returned from the first-level fetch, a second-level fetch is initiated.

Coarse page table descriptor bit assignments are described in Table 3-5.

Table 3-5 Coarse page table descriptor bits

Bits	Description
[31:10]	These bits form the base for referencing the second-level descriptor (the coarse page table index for the entry is derived from the MVA)
[9]	Always written as 0
[8:5]	These bits specify one of the 16 possible domains (held in the domain access control registers) that contain the primary access controls
[4]	Always written as 1
[3:2]	Always written as 0
[1:0]	These bits must be 01 to indicate a coarse page table descriptor

3.2.6 Fine page table descriptor

A fine page table descriptor provides the base address of a page table that contains second-level descriptors for large page, small page, or tiny page accesses. Fine page tables have 1024 entries, splitting the 1MB that the table describes into 1KB blocks. Figure 3-7 shows the format of a fine page table descriptor.

Figure 3-7 Fine page table descriptor

Note

If a fine page table descriptor is returned from the first-level fetch, a second-level fetch is initiated.

Table 3-6 shows the fine page table descriptor bit assignments.

Table 3-6 Fine page table descriptor bits

Bits	Description
[31:12]	These bits form the base for referencing the second-level descriptor (the fine page table index for the entry is derived from the MVA)
[11:9]	Always written as 0
[8:5]	These bits specify one of the 16 possible domains (held in the domain access control registers) that contain the primary access controls
[4]	Always written as 1
[3:2]	Always written as 0
[1:0]	These bits must be 11 to indicate a fine page table descriptor

3.2.7 Translating section references

Figure 3-8 on page 3-14 shows the complete section translation sequence.

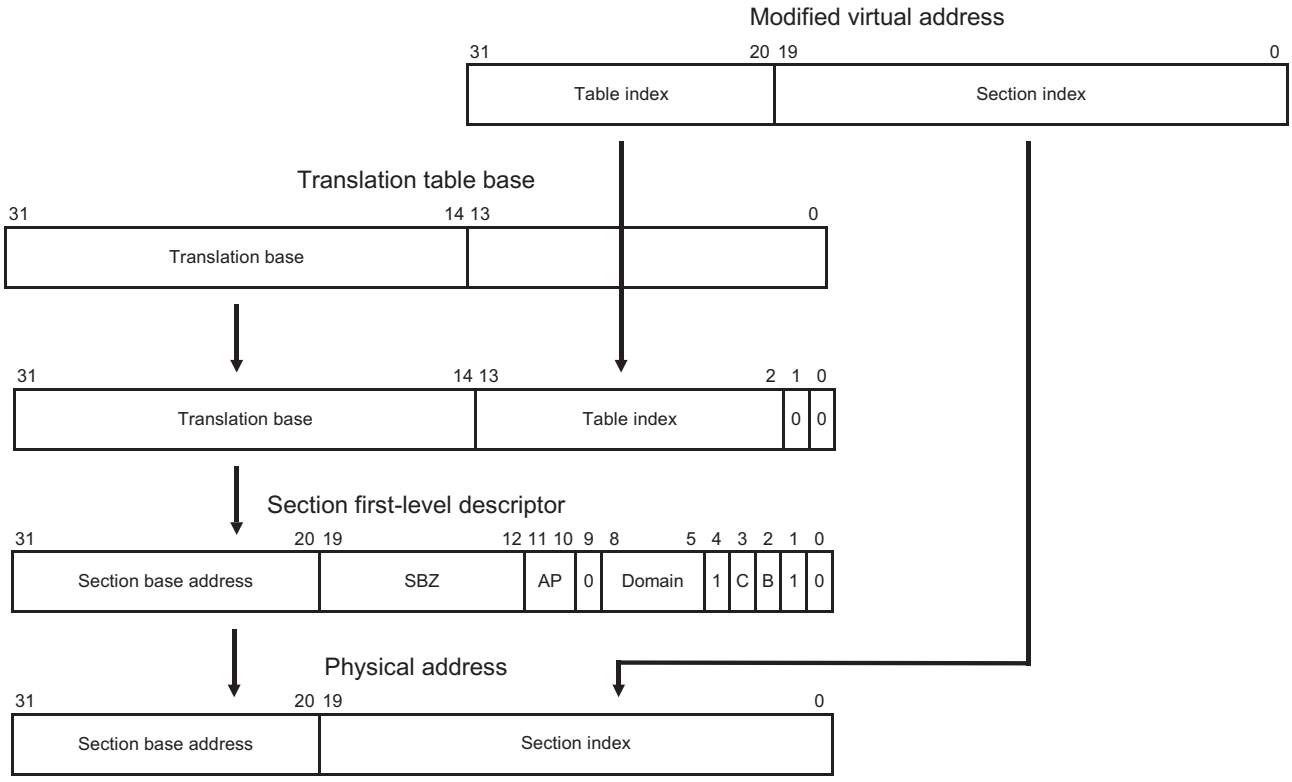


Figure 3-8 Section translation

3.2.8 Second-level descriptor

If the first-level fetch returns either a coarse page table descriptor or a fine page table descriptor, this provides the base address of the page table to be used. The page table is then accessed and a second-level descriptor is returned. Figure 3-9 on page 3-15 shows the format of second-level descriptors.

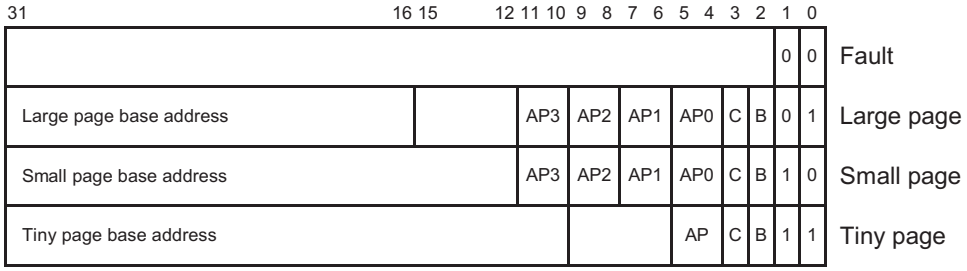


Figure 3-9 Second-level descriptor

A second-level descriptor defines a tiny, a small, or a large page descriptor, or is invalid:

- a large page descriptor provides the base address of a 64KB block of memory
- a small page descriptor provides the base address of a 4KB block of memory
- a tiny page descriptor provides the base address of a 1KB block of memory.

Coarse page tables provide base addresses for either small or large pages. Large page descriptors must be repeated in 16 consecutive entries. Small page descriptors must be repeated in each consecutive entry.

Fine page tables provide base addresses for large, small, or tiny pages. Large page descriptors must be repeated in 64 consecutive entries. Small page descriptors must be repeated in four consecutive entries and tiny page descriptors must be repeated in each consecutive entry.

Second-level descriptor bit assignments are described in Table 3-7.

Table 3-7 Second-level descriptor bits

Bits			Description
Large	Small	Tiny	
[31:16]	[31:12]	[31:10]	These bits form the corresponding bits of the physical address.
[15:12]	-	[9:6]	Should Be Zero.

Table 3-7 Second-level descriptor bits (continued)

Bits			Description
Large	Small	Tiny	
[11:4]	[11:4]	[5:4]	Access permission bits. *Domain access control* on page 3-24 and *Fault checking sequence* on page 3-26 show how to interpret the access permission bits.
[3:2]	[3:2]	[3:2]	These bits, C and B, indicate whether the area of memory mapped by this page is treated as write-back cachable, write-through cachable, noncached buffered, or noncached nonbuffered.
[1:0]	[1:0]	[1:0]	These bits indicate the page size and validity and are interpreted as shown in Table 3-8.

The two least significant bits of the second-level descriptor indicate the descriptor type as shown in Table 3-8.

Table 3-8 Interpreting page table entry bits [1:0]

Value	Meaning	Description
0 0	Invalid	Generates a page translation fault
0 1	Large page	Indicates that this is a 64KB page
1 0	Small page	Indicates that this is a 4KB page
1 1	Tiny page	Indicates that this is a 1KB page

Note

Tiny pages do not support subpage permissions and therefore only have one set of access permission bits.

3.2.9 Translating large page references

Figure 3-10 on page 3-17 shows the complete translation sequence for a 64KB large page.

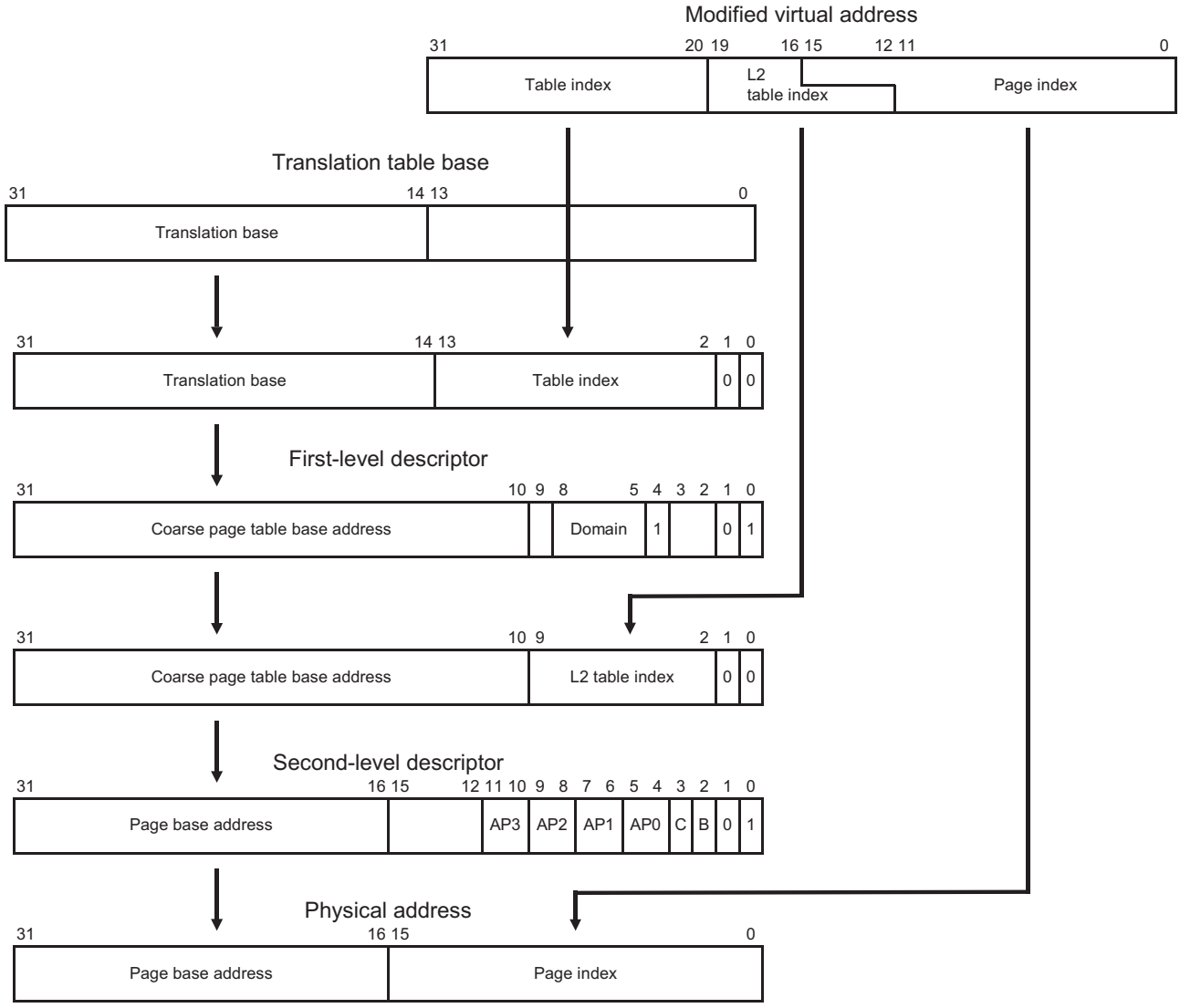


Figure 3-10 Large page translation from a coarse page table

Because the upper four bits of the page index and low-order four bits of the coarse page table index overlap, each coarse page table entry for a large page must be duplicated 16 times (in consecutive memory locations) in the coarse page table.

If a large page descriptor is included in a fine page table, the high-order six bits of the page index and low-order six bits of the fine page table index overlap. Each fine page table entry for a large page must therefore be duplicated 64 times.

3.2.10 Translating small page references

Figure 3-11 shows the complete translation sequence for a 4KB small page.

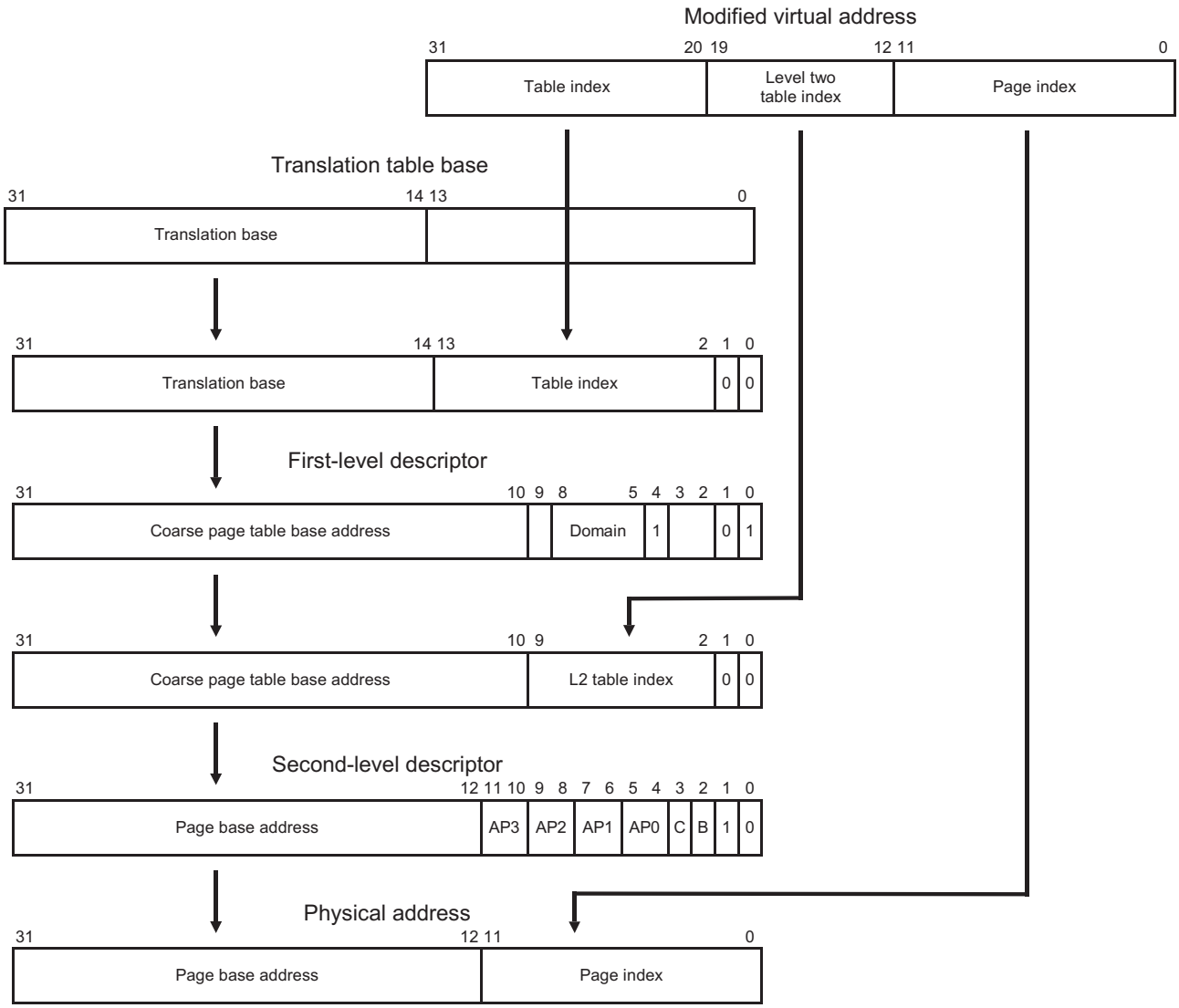


Figure 3-11 Small page translation from a coarse page table

If a small page descriptor is included in a fine page table, the upper two bits of the page index and low-order two bits of the fine page table index overlap. Each fine page table entry for a small page must therefore be duplicated four times.

3.2.11 Translating tiny page references

Figure 3-12 shows the complete translation sequence for a 1KB tiny page.

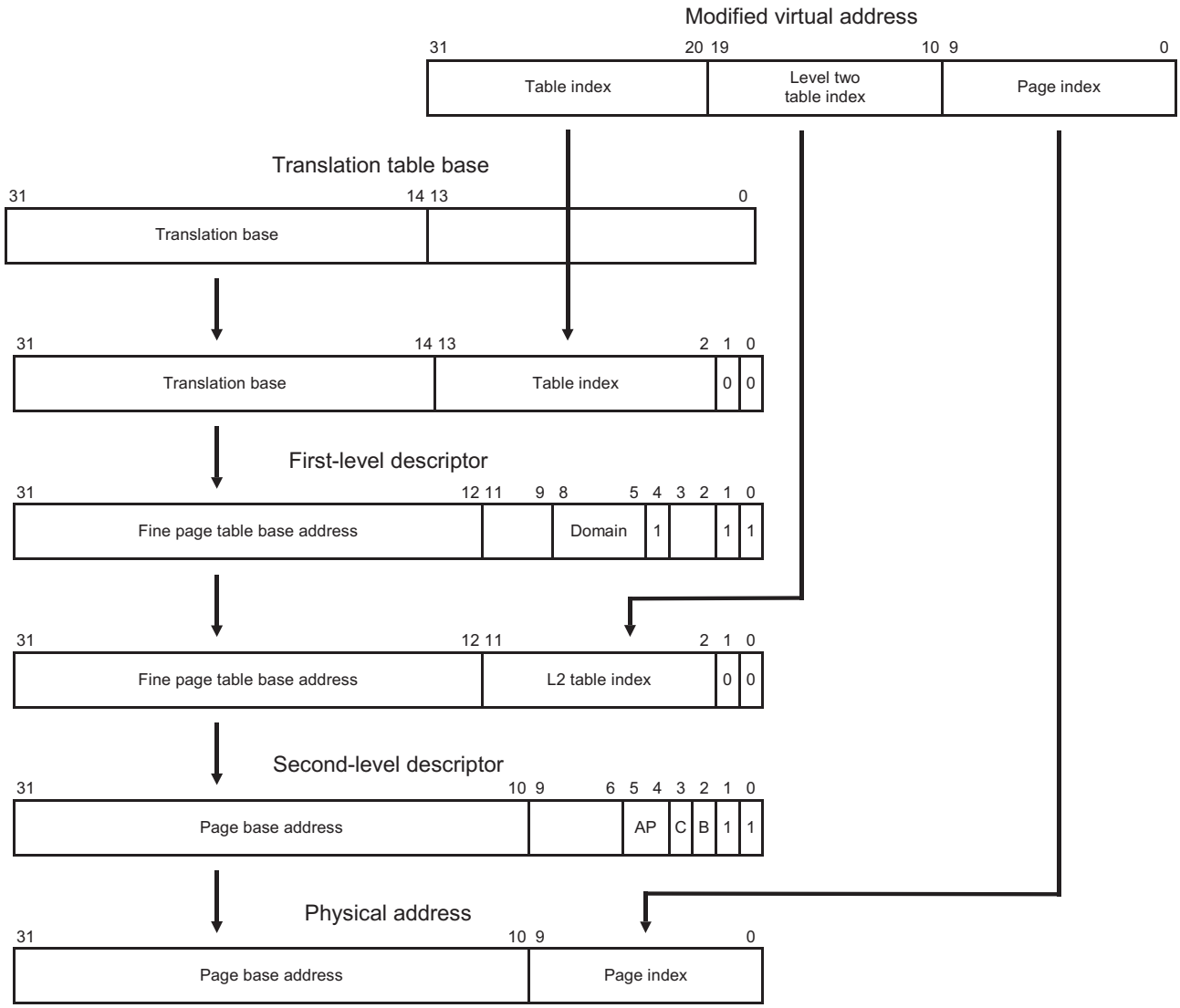


Figure 3-12 Tiny page translation from a fine page table

Page translation involves one additional step beyond that of a section translation. The first-level descriptor is the fine page table descriptor and this is used to point to the first-level descriptor.

Note

The domain specified in the first-level description and access permissions specified in the first-level description together determine whether the access has permissions to proceed. See section *Domain access control* on page 3-24 for details.

Subpages

You can define access permissions for subpages of small and large pages. If, during a page table walk, a small or large page has a different subpage permission, only the subpage being accessed is written into the TLB. For example, a 16KB (large page) subpage entry is written into the TLB if the subpage permission differs, and a 64KB entry is put in the TLB if the subpage permissions are identical.

When you use subpage permissions, and the page entry then has to be invalidated, you must invalidate all four subpages separately.

3.3 MMU faults and CPU aborts

The MMU generates an abort on the following types of faults:

- alignment faults (data accesses only)
- translation faults
- domain faults
- permission faults.

In addition, an external abort can be raised by the external system. This can happen only for access types that have the core synchronized to the external system:

- page walks
- noncached reads
- nonbuffered writes
- noncached read-lock-write sequence (SWP).

Alignment fault checking is enabled by the A bit in CP15 c1. Alignment fault checking is not affected by whether or not the MMU is enabled. Translation, domain, and permission faults are only generated when the MMU is enabled.

The access control mechanisms of the MMU detect the conditions that produce these faults. If a fault is detected as a result of a memory access, the MMU aborts the access and signals the fault condition to the CPU core. The MMU retains status and address information about faults generated by the data accesses in the data fault status register and fault address register (see *Fault address and fault status registers*).

The MMU also retains status about faults generated by instruction fetches in the instruction fault status register.

Note

The address information for an instruction side abort is contained in the core link register r14_abt.

An access violation for a given memory access inhibits any corresponding external access to the AHB interface, with an abort returned to the CPU core.

3.3.1 Fault address and fault status registers

On a Data Abort, the MMU places an encoded four-bit value, the fault status, along with the four-bit encoded domain number, in the data FSR. Similarly, on a Prefetch Abort, in the instruction FSR (intended for debug purposes only). In addition, the MVA associated with the Data Abort is latched into the FAR. If an access violation simultaneously generates more than one source of abort, they are encoded in the priority given in Table 3-9. The FAR is not updated by faults caused by instruction prefetches.

Fault status register (FSR)

Table 3-9 shows the various access permissions and controls supported by the data MMU, and how these are interpreted to generate faults.

Table 3-9 Priority encoding of fault status

Priority	Source	Size	Status	Domain
Highest	Alignment	-	b00x1	Invalid
	External abort on translation	First level Second level	b1100 b1110	Invalid Valid
	Translation	Section Page	b0101 b0111	Invalid Valid
	Domain	Section Page	b1001 b1011	Valid Valid
	Permission	Section Page	b1101 b1111	Valid Valid
Lowest	External abort	Section or page	b10x0	Invalid

Note

Alignment faults can write either b0001 or b0011 into FSR[3:0].

Invalid values can occur in the status bit encoding for domain faults. This happens when the fault is raised before a valid domain field has been read from a page table description.

Aborts masked by a higher priority abort can be regenerated by fixing the cause of the higher priority abort, and repeating the access.

Alignment faults are not possible for instruction fetches.

The instruction FSR can also be updated for instruction prefetch operations (`MCR p15,0,<Rd>,c7,c13,1`).

Fault address register (FAR)

For load and store instructions that can involve the transfer of more than one word (LDM/STM, LDRD, STRD, and STC/LDC), the value written into the FAR register depends on the type of access, and for external aborts, on whether or not the access crosses a 1KB boundary. Table 3-10 shows the FAR values for multi-word transfers.

Table 3-10 FAR values for multi-word transfers

Source	FAR
Alignment	MVA of first aborted address in transfer.
External abort on translation	MVA of first aborted address in transfer.
Translation	MVA of first aborted address in transfer.
Domain	MVA of first aborted address in transfer.
Permission	MVA of first aborted address in transfer.
External abort for noncached reads, or nonbuffered writes.	MVA of last address before 1KB boundary if any word of the transfer before 1KB boundary is externally aborted. MVA of last address in transfer if the first externally aborted word is after 1KB boundary.

Compatibility Issues

To enable code to be easily ported to ARM architecture v4 or v5 MMUs, or to future architectures, it is recommended that no reliance is made on external abort behavior.

The instruction FSR is intended for debugging purposes only. Code that is intended to be ported to other ARM architecture v4 or v5 MMUs must not use the instruction FSR.

3.4 Domain access control

MMU accesses are primarily controlled through the use of domains. There are 16 domains and each has a two-bit field to define access to it. Two types of user are supported:

- clients
- managers.

The domains are defined in the domain access control register, CP15 c3. Figure 2-7 on page 2-18 shows how the 32 bits of the register are allocated to define the 16 two-bit domains.

Table 3-11 defines how the bits within each domain are interpreted to specify the access permissions.

Table 3-11 Domain access control register, access control bits

Value	Meaning	Description
0 0	No access	Any access generates a domain fault.
0 1	Client	Accesses are checked against the access permission bits in the section or page descriptor.
1 0	Reserved	Reserved. Currently behaves like the no access mode.
1 1	Manager	Accesses are not checked against the access permission bits so a permission fault cannot be generated.

Table 3-12 shows how to interpret the *Access Permission* (AP) bits and how their interpretation is dependent on the R and S bits (Control Register c1 bits [9:8]).

Table 3-12 Interpreting access permission (AP) bits

AP	S	R	Privileged permissions	User permissions
0 0	0	0	No access	No access
0 0	1	0	Read-only	No access
0 0	0	1	Read-only	Read-only
0 0	1	1	Unpredictable	Unpredictable

Table 3-12 Interpreting access permission (AP) bits (continued)

AP	S	R	Privileged permissions	User permissions
0 1	x	x	Read/write	No access
1 0	x	x	Read/write	Read-only
1 1	x	x	Read/write	Read/write

3.5 Fault checking sequence

The sequence the MMU uses to check for access faults is different for sections and pages. The sequence for both types of access is shown in Figure 3-13.

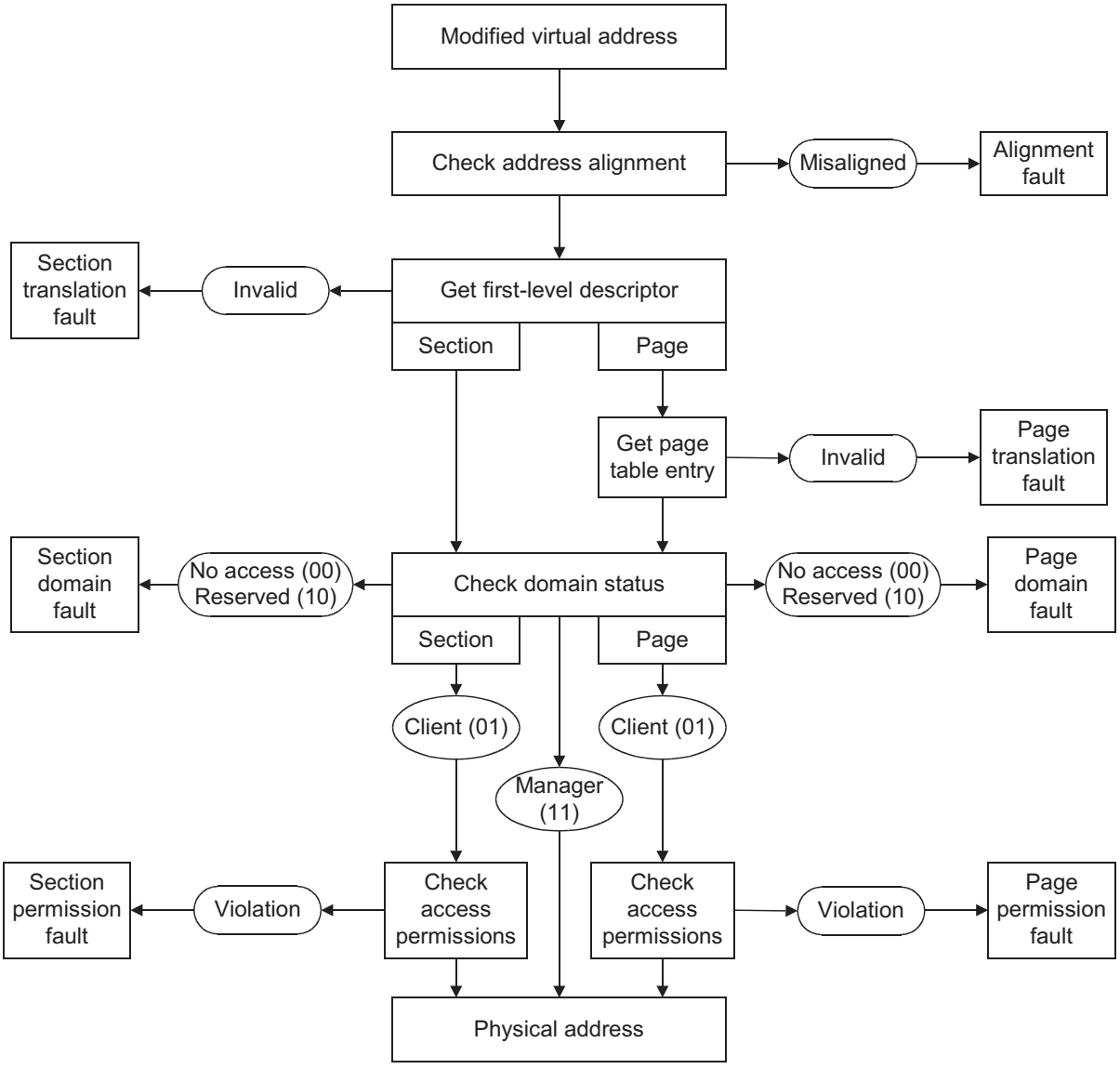


Figure 3-13 Sequence for checking faults

The conditions that generate each of the faults are described in:

- *Alignment faults* on page 3-27

- *Translation faults*
- *Domain faults*
- *Permission faults* on page 3-28.

3.5.1 Alignment faults

If alignment fault checking is enabled (the A bit in CP15 c1 is set), the MMU generates an alignment fault on any data word access if the address is not word-aligned, or on any halfword access if the address is not halfword-aligned, irrespective of whether the MMU is enabled or not. An alignment fault is not generated on any instruction fetch or any byte access.

Note

If an access generates an alignment fault, the access sequence aborts without reference to other permission checks.

3.5.2 Translation faults

There are two types of translation fault:

Section A section translation fault is generated if the level one descriptor is marked as invalid. This happens if bits [1:0] of the descriptor are both 0.

Page A page translation fault is generated if the level one descriptor is marked as invalid. This happens if bits [1:0] of the descriptor are both 0.

3.5.3 Domain faults

There are two types of domain fault:

Section The level one descriptor holds the four-bit domain field, which selects one of the 16 two-bit domains in the domain access control register. The two bits of the specified domain are then checked for access permissions as described in Table 3-12 on page 3-24. The domain is checked when the level one descriptor is returned.

Page The level one descriptor holds the four-bit domain field, which selects one of the 16 two-bit domains in the domain access control register. The two bits of the specified domain are then checked for access permissions as described in Table 3-12 on page 3-24. The domain is checked when the level one descriptor is returned.

If the specified access is either no access (00), or reserved (10), then either a section domain fault or page domain fault occurs.

3.5.4 Permission faults

If the two-bit domain field returns 01 (client), then access permissions are checked as follows:

Section If the level one descriptor defines a section-mapped access, the AP bits of the descriptor define whether or not the access is allowed, according to Table 3-12 on page 3-24. Their interpretation is dependent on the setting of the S and R bits (CP15 c1 bits 8 and 9). If the access is not allowed, a section permission fault is generated.

Large page or small page

If the level one descriptor defines a page-mapped access and the level two descriptor is for a large or small page, four access permission fields (ap3 to ap0) are specified, each corresponding to one quarter of the page. For small pages ap3 is selected by the top 1KB of the page and ap0 is selected by the bottom 1KB of the page. For large pages, ap3 is selected by the top 16KB of the page and ap0 is selected by the bottom 16KB of the page. The selected AP bits are then interpreted in exactly the same way as for a section (see Table 3-12 on page 3-24), the only difference is that the fault generated is a page permission fault.

Tiny page If the level one descriptor defines a page-mapped access, and the level two descriptor is for a tiny page, the AP bits of the level one descriptor define whether or not the access is allowed in the same way as for a section. The fault generated is a page permission fault.

3.6 External aborts

In addition to the MMU generated aborts, external aborts can be generated for certain types of access that involve transfers over the AHB bus. These can be used to flag errors on external memory accesses. However, not all accesses can be aborted in this way.

The following accesses can be externally aborted:

- page walks
- noncached reads
- nonbuffered writes
- noncached read-lock-write (SWP) sequence.

For a read-lock-write (SWP) sequence, if the read externally aborts, the write is always attempted.

A swap to an NCB region is forced to have precisely the same behavior as a swap to an NCNB region. This means that the write part of a swap to an NCB region can be externally aborted.

3.6.1 Enabling the MMU

Before enabling the MMU using CP15 c1 you must:

1. Program the TTB register (CP15 c2) and the domain access control register (Cp15 c3).
2. Program first-level and second-level page tables as required, ensuring that a valid translation table is placed in memory at the location specified by the TTB register.

When these steps have been performed, you can enable the MMU by setting CP15 c1 bit 0 HIGH.

Care must be taken if the translated address differs from the untranslated address because several instructions following the enabling of the MMU might have been prefetched with the MMU off (VA = MVA = PA).

In this case, enabling the MMU can be considered as a branch with delayed execution. A similar situation occurs when the MMU is disabled. Consider the following code sequence:

```
MRC p15, 0, R1, c1, C0, 0   ; Read control register
ORR R1, #0x1                ; Set M bit
MCR p15, 0,R1,C1, C0,0      ; Write control register and enable MMU
Fetch Flat
Fetch Flat
Fetch Translated
```


Note

Because the same register, CP15 c1, controls the enabling of the ICache, DCache, and the MMU, all three can be enabled using a single MCR instruction.

3.6.2 Disabling the MMU

To disable the MMU, clear bit 0 in CP15 c1.

Note

If the MMU is enabled, then disabled, and subsequently re-enabled, the contents of the TLB are preserved. If these are now invalid, then the TLB must be invalidated before re-enabling the MMU. See *TLB Operations Register c8* on page 2-24.

3.7 TLB structure

The MMU contains a single unified TLB used for both data accesses and instruction fetches. The TLB is divided into two parts:

- an eight-entry fully-associative part used exclusively for holding locked down TLB entries
- a set-associative part for all other entries, 2 way x 32 entry.

Whether an entry is placed in the set-associative, or lockdown part of the TLB is dependent on the state of the TLB lockdown register, when the entry is written into the TLB (see *TLB Lockdown Register c10* on page 2-32).

When an entry has been written into the lockdown part of the TLB, it can only be removed by being overwritten explicitly, or by an MVA-based TLB invalidate operation, where the MVA matches the locked down entry.

The structure of the set-associative part of the TLB does not form part of the programmer's model for the ARM926EJ-S processor. No assumptions must be made about the structure, replacement algorithm, or persistence of entries in the set-associative part. Specifically:

- Any entry written into the set-associative part of the TLB can be removed at any time. The set-associative part of the TLB must be considered as a temporary cache of translation/page table information. No reliance must be placed on an entry either residing or not residing in the set-associative TLB, unless that entry already exists in the lockdown TLB. The set-associative part of the TLB can contain entries that are defined in the page tables but do not correspond to address values that have been accessed since the TLB was invalidated.
- The set-associative part of the TLB must be considered as a cache of the underlying page table, where memory coherency must be maintained at all times. If a level one descriptor is modified in main memory, then to guarantee coherency either an invalidate TLB or invalidate TLB by entry operation must be used to remove any cached copies of the level one descriptor. This is required regardless of the type of level one descriptor (section, level two page table reference, or fault).
- If any of the subpage permissions for a given page are different, then each of the subpages are treated separately. To invalidate all the entries associated with a page with subpage permissions then four MVA-based invalidate operations are required, one for each subpage.

Chapter 4
Caches and Write Buffer

This chapter describes the *Instruction Cache* (ICache), the *Data Cache* (DCache), and the write buffer. It contains the following sections:

4.1 About the caches and write buffer

The ARM926EJ-S processor includes:

- an *Instruction Cache* (ICache)
- a *Data Cache* (DCache)
- a write buffer.

The size of the caches can be from 4KB to 128KB, in power of two increments.

The caches have the following features:

- The caches are virtual index, virtual tag, addressed using the *Modified Virtual Address* (MVA). This enables the avoidance of cache cleaning and/or invalidating on context switch.
- The caches are four-way set associative, with a cache line length of eight words per line (32 bytes per line), and with two dirty bits in the DCache.
- The DCache supports write-through and write-back (or copyback) cache operations, selected by memory region using the C and B bits in the MMU translation tables.
- Allocate on read-miss is supported. The caches perform critical-word first cache refilling.
- Pseudo-random or round-robin replacement, selectable by the RR bit in CP15 c1.
- Cache lockdown registers enable control over which cache ways are used for allocation on a linefill, providing a mechanism for both lockdown and controlling cache pollution.
- The DCache stores the *Physical Address* (PA) tag corresponding to each DCache entry in the tag RAM for use during cache line write-backs, in addition to the Virtual Address tag stored in the tag RAM. This means that the MMU is not involved in DCache write-back operations, removing the possibility of TLB misses related to the write-back address.
- The PLD data preload instruction does not cause data cache linefills. It is treated as a NOP instruction.
- Cache maintenance operations to provide efficient invalidation of:
  - — the entire DCache or ICache
  - — regions of the DCache or ICache
  - — regions of virtual memory.

  They also provide operations for efficient cleaning and invalidation of:
  - — the entire DCache
  - — regions of the DCache
  - — regions of virtual memory.

The latter allows DCache coherency to be efficiently maintained when small code changes occur, for example for self-modifying code and changes to exception vectors.

4.2 Write buffer

The write buffer is used for all writes to a noncachable, bufferable region, write-through region, and write misses to a write-back region. A separate buffer is incorporated in the DCache for holding write-back data for cache line evictions or cleaning of dirty cache lines.

The main write buffer has a 16-word data buffer and a four-address buffer.

The DCache write-back buffer has eight data word entries and a single address entry.

The MCR drain write buffer instruction enables both write buffers to be drained under software control.

The MCR wait for interrupt causes both write buffers to be drained and the ARM926EJ-S processor to be put into a low-power state until an interrupt occurs.

Write buffer behavior is described in Table 4-4 on page 4-6.

No forwarding takes place for read accesses which have corresponding pending writes in the write buffer. For such accesses the write buffer is drained and the value fetched from external memory.

4.3 Enabling the caches

On reset, the ICache and DCache entries are all invalidated and the caches are disabled. The caches are not accessed for reads or writes. The caches are enabled using the I, C, and M bits from CP15 c1, and can be enabled independently of one another. Table 4-1 gives the I and M bit settings for the ICache, and the associated behavior. The priority of the TCM and cache behavior is described in *TCM and cache access priorities* on page 4-8.

Table 4-1 CP15 c1 I and M bit settings for the ICache

CP15 c1 I bit	CP15 c1 M bit	ARM926EJ-S behavior
0	-	ICache disabled. All instruction fetches are fetched from external memory (AHB).
1	0	ICache enabled, MMU disabled. All instruction fetches are cachable, with no protection checks. All addresses are flat mapped, that is VA = MVA= PA.
1	1	ICache enabled, MMU enabled. Instruction fetches are cachable or noncachable depending on the page descriptor C bit (see Table 4-2), and protection checks are performed. All addresses are remapped from VA to PA, depending on the page entry, that is the VA is translated to an MVA, and the MVA is remapped to a PA.

Table 4-2 gives the page table C bit settings for the ICache (CP15 c1 I bit = M bit = 1).

Table 4-2 Page table C bit settings for the ICache

Page table C bit	Description	ARM926EJ-S behavior	
0	Noncachable	ICache disabled. All instruction fetches are fetched from external memory.	
1	Cachable	**Cache hit**	Read from the ICache.
		Cache miss	Linefill from external memory.

Table 4-3 gives the CP15 c1 C and M bit settings for DCache, and the associated behavior.

Table 4-3 CP15 c1 C and M bit settings for the DCache

CP15 c1 C bit	CP15 c1 M bit	ARM926EJ-S behavior
0	0	DCache disabled. All data accesses are to the external memory.
1	0	DCache enabled, MMU disabled. The C bit is overriden by the M bit setting, which means that the DCache is effectively disabled. All data accesses are noncachable, nonbufferable, with no protection checks. All addresses are flat mapped, that is VA = MVA = PA.
1	1	DCache enabled, MMU enabled. All data accesses are cachable or noncachable depending on the page descriptor C bit and B bit (see Table 4-4), and protection checks are performed. All addresses are remapped from VA to PA, depending on the MMU page table entry, that is the VA is translated to an MVA, and the MVA is remapped to a PA.

Table 4-4 gives the page table C and B bit settings for the DCache (CP15 c1 C bit = M bit = 1), and the associated behavior.

Table 4-4 Page table C and B bit settings for the DCache

Page table C bit	Page table B bit	Description	ARM926EJ-S behavior
0	0	Noncachable, nonbufferable	DCache disabled. Read from external memory. Write as a nonbuffered store(s) to external memory. DCache is not updated.
0	1	Noncachable, bufferable	DCache disabled. Read from external memory. Write as a buffered store(s) to external memory. DCache is not updated.
1	0	Write-through	DCache enabled: **Read hit** Read from DCache **Read miss** Linefill **Write hit** Write to the DCache, and buffered store to external memory **Write miss** Buffered store to external memory

Table 4-4 Page table C and B bit settings for the DCache (continued)

Page table C bit	Page table B bit	Description	ARM926EJ-S behavior	
1	1	Write-back	DCache enabled:	
			Read hit	Read from DCache
			Read miss	Linefill
			Write hit	Write to the DCache only
			Write miss	Buffered store to external memory.

4.4 TCM and cache access priorities

The priorities that apply to the ARM926EJ-S processor for instruction accesses are shown in Table 4-5. The ARM926EJ-S processor gives highest priority to an address that is in the instruction TCM region.

Table 4-5 Instruction access priorities to the TCM and cache

Address in ITCM region	Address in DTCM region	Cachable in page descriptor	ARM926EJ-S behavior
Yes	Yes	Don't care	Access ITCM
Yes	No	Cachable	Access ITCM
Yes	No	Noncachable	Access ITCM
No	Don't care	Cachable	Access ICache
No	Don't care	Noncachable	Access external memory

The priorities that apply to the ARM926EJ-S processor for data accesses are shown in Table 4-6. The Harvard arrangement for the TCM and caches requires that data reads and writes can access the Instruction TCM for both reads and writes. (The column order for Table 4-6 is deliberately the same as for instruction accesses in Table 4-5.)

Table 4-6 Data access priorities to the TCM and cache

Address in ITCM Region	Address in DTCM region	Cachable in page descriptor	ARM926EJ-S behavior
Yes	Yes	Don't care	Access DTCM
No	Yes	Cachable	Access DTCM
No	Yes	Noncachable	Access DTCM
Yes	No	Cachable	Access ITCM
Yes	No	Noncachable	Access ITCM
No	No	Cachable	Access DCache
No	No	Noncachable	Access external memory

4.5 Cache MVA and Set/Way formats

This section shows how the MVA and Set/Way formats of ARM926EJ-S caches map to a generic virtually indexed, virtually addressed cache.

Figure 4-1 shows a generic, virtually indexed, virtually addressed cache.

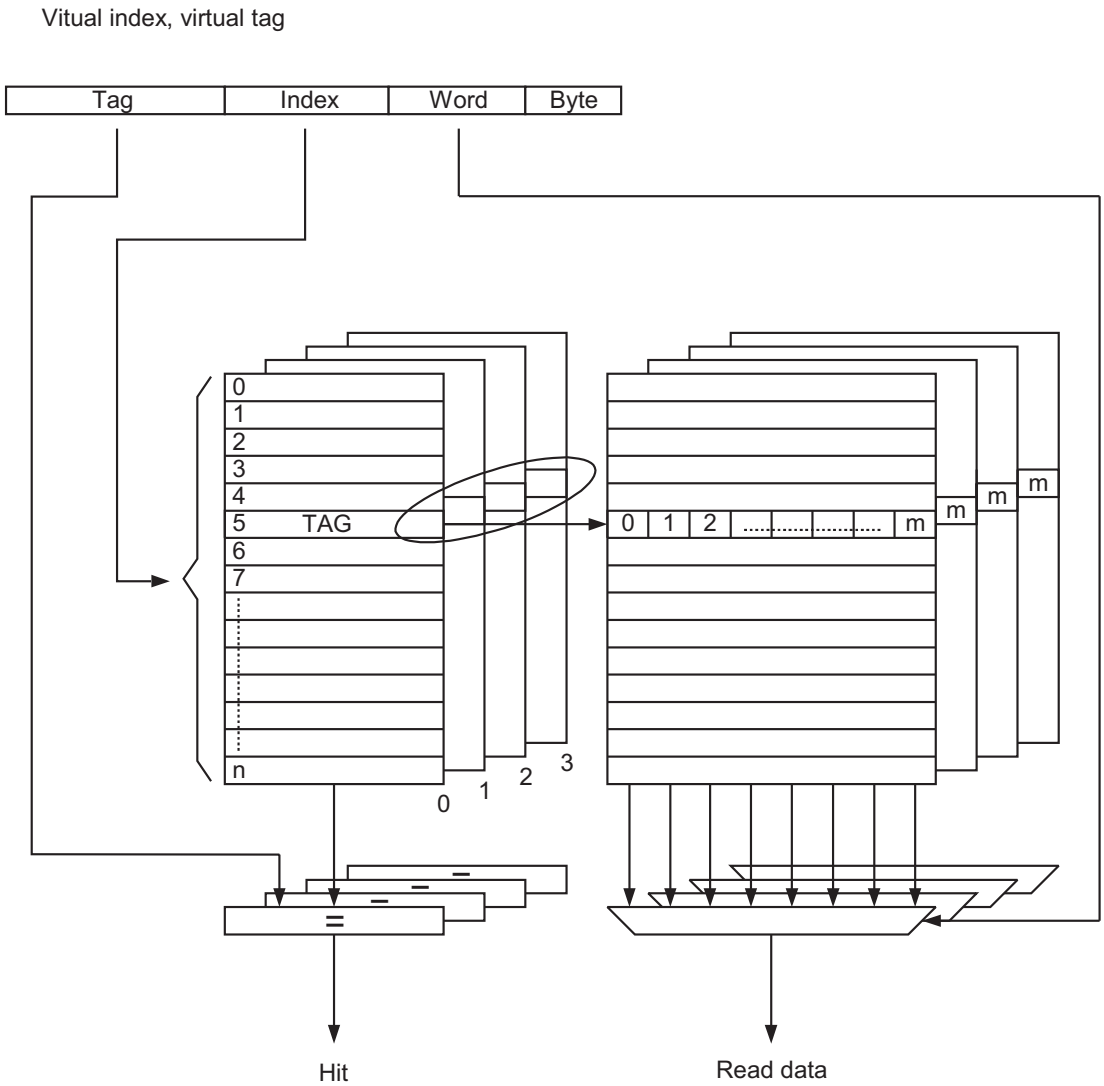


Figure 4-1 Generic virtually indexed virtually addressed cache

The ARM926EJ-S cache format is shown in Figure 4-2 on page 4-10.

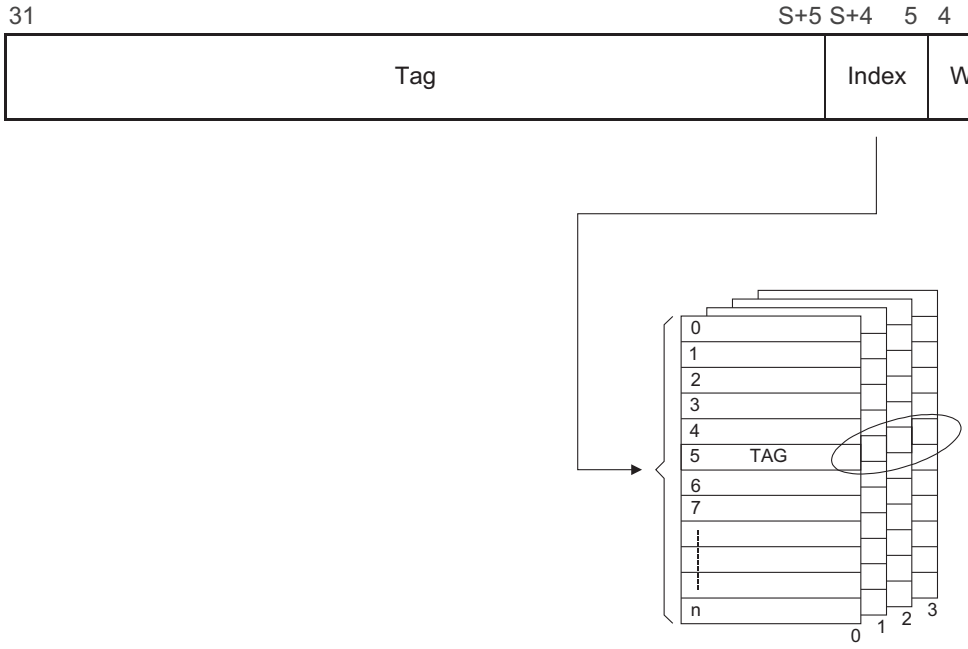


Figure 4-2 ARM926EJ-S cache associativity

Table 4-7 shows values of S and NSETS for an ARM926EJ-S cache.

Table 4-7 Values of S and NSETS

ARM926EJ-S cache size	S	NSETS
4KB	5	32
8KB	6	64
16KB	7	128
32KB	8	256
64KB	9	512
128KB	10	1024

Figure 4-2 shows the ARM926EJ-S cache associativity. In Figure 4-2, the following points apply:

- the group of tags of the same Index define a Set
- the number of tags in a Set is the Associativity

- the ARM926EJ-S caches are four-way Associative
- the range of tags addressed by the Index define a Way
- the number of tags in a Way is the number of Sets, NSETS.

The Set/Way/Word format for ARM926EJ-S caches is shown in Figure 4-3.

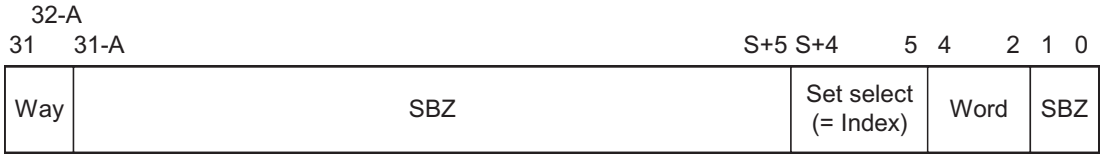


Figure 4-3 ARM926EJ-S cache Set/Way/Word format

In Figure 4-3:

A = $\log_2$ Associativity.

For example, for a four-way cache A = 2.

S = $\log_2$ NSETS.

Chapter 5
Tightly-Coupled Memory Interface

This chapter describes the ARM926EJ-S *Tightly-Coupled Memory* (TCM) interface. It contains the following sections:

5.1 About the tightly-coupled memory interface

The ARM926EJ-S processor enables low latency access to external memories using the *Tightly Coupled Memory* (TCM) interface. The term tightly coupled memory refers to the relationship between the ARM9EJ-S CPU core, and the operation of the memories, where there is a strong correlation between the instruction and data access activity of the ARM9EJ-S and the accesses made to external memory. This is in contrast to the accesses made to the AHB interfaces, which are relatively decoupled from the ARM9EJ-S core.

TCMs are intended for storing certain types of critical code or data, where low latency, deterministic access is required. TCMs are not necessarily the best choice for all types of such code or data, if code or data exhibit a high degree of spatial or temporal locality better performance may be obtained by using cache memory. (See Chapter 4 *Caches and Write Buffer*).

The ARM926EJ-S processor supports two TCM regions, one for instructions (ITCM) and one for data (DTCM). The ITCM interface can also be accessed by the data side of the ARM9EJ-S core. This is necessary for code to be loaded into the ITCM, for SWI and emulated instruction handlers, and for accesses to PC-relative literal pools.

The TCM address space is physically addressed, and the location of the TCM regions in the physical address space is controlled by the TCM Region Register (see *TCM Region Register c9* on page 2-29). The physical size of the TCM regions are defined by external inputs (**IRSIZE**, **DRSIZE**), and ranges from 4KB to 1MB. The encoding for these pins is shown in *TCM Size field encoding* on page 2-30. The TCM regions can be placed anywhere in the physical address map, with the restriction that the TCM base address must be aligned with the TCM size, and that the instruction and data TCM regions do not overlap. The TCM region size can be interrogated by software by reading the TCM Status Register (see *TCM Status Register c0* on page 2-12).

The **INITRAM** pin allows the ARM926EJ-S processor to boot from instruction TCM space after system reset. If **INITRAM** is asserted during system reset and the **VINITHI** pin is deasserted, then the ARM926EJ-S processor fetches the instruction at `0x00000000` from the instruction TCM interface. (If both **INITRAM** and **VINITHI** are asserted, the first instruction fetch after reset is from `0xFFFF0000` over the AHB).

The TCM interface supports memory accesses with zero or more wait-states. The requirement to support zero wait state accesses imposes various constraints on the TCM sub-system design that do not apply when interfacing memories with a generic bus interface such as AHB.

Because of timing restrictions, read accesses occur on the TCM interface without prior qualification by the MMU. This means that all reads on the TCM interface must be treated as being speculative, and consequently precludes the use of read-sensitive

memory. The TCM interface contains a two entry write buffer, which avoids the need for stall cycles because of the mismatch between the ARM9EJ-S native memory interface, and the requirements for standard SRAM.

TCM accesses can be extended by using the **IRWAIT/DRWAIT** inputs to generate wait states. However, the timing of these and other interface signals is such that the types of memory sub-systems that can be implemented are limited. For example schemes that require an address decode to determine if a wait-state should be inserted are not possible if operating at maximum frequency.

DMA access can be performed either by using the **IRWAIT/DRWAIT** signals to insert wait states during a DMA access, or by using the dedicated DMA interface, which avoids the need to externally multiplex critical interface signals when single cycle access memory is used.

5.2 TCM interface signals

The TCM interface is designed to be compatible the timings of standard ASIC SRAM components, allowing connection to single cycle SRAM with minimal interfacing logic required. For standard SRAM the chip-select, address, and write data/control signals are setup in one cycle, and the read or write operation takes place in the next cycle.

5.2.1 Data interface signals

The signals in the DTCM interface can be grouped by function into four categories.

- Control signals
  - **DRCS**
  - **DRWAIT**
  - **DRIDLE**
- Address and attribute signals
  - **DRSEQ**
  - **DRADDR[17:0]**
  - **DRWBL[3:0]**
  - **DRnRW**
- Data signals
  - **DRRD[31:0]**
  - **DRWD[31:0]**
- DMA signals
  - **DRDMAEN**
  - **DRDMACS**
  - **DRDMAADDR[17:0]**.

Control signals

The control signals for the data interface are:

DRCS

DRCS is used to indicate that an access will commence in the following cycle. For simple zero wait state TCM systems the **DRCS** signals corresponds directly to a memory chip select signal. For more complex systems **DRCS** corresponds to a memory request signal.

DRWAIT

DRWAIT is used to extend a TCM transfer by inserting wait states. The timing of the **DRWAIT** signal is a cycle ahead of the cycle in which the data transfer takes place, which means that if an access is to be waited, **DRWAIT** must be asserted in the same cycle as **DRCS** and deasserted one cycle before the data transfer takes place.

DRIDLE

The **DRIDLE** signal provides an early indication that no TCM access will take place in the current cycle.

Address and attribute signals

All of the address and attribute signals are valid when **DRCS** is asserted (and valid), with the exception of **DRSEQ** which also has a defined value during wait states (when **DRCS** is not valid).

DRSEQ

When **DRCS** is asserted and valid, **DRSEQ** indicates if the address for the current TCM access is sequential to the previous access. During wait states **DRSEQ** is forced HIGH.

DRADDR[17:0]

DRADDR is the word (32 bit) address for the transfer.

DRnRW

DRnRW indicates if the access is a read or a write.

DRWBL[3:0]

DRWBL is used to indicate which byte(s) of an address should be updated for write accesses. This is dependant on the address, the size of the transfer, and the current endianess setting. **DRWBL** is b0000 for reads.

Data signals

The data signals are:

DRRD[31:0]

DRRD is the read data returned by the TCM. For zero wait state systems, **DRRD** is valid in the cycle after **DRCS**. For systems with wait states, **DRRD** is valid in the cycle after **DRWAIT** is deasserted.

DRWD[31:0]

DRWD is the write data written into the TCM. It is valid in the same cycle as **DRCS** and held stable until the penultimate cycle of the access.

DMA signals

The DMA interface allows the values of **DRADDR** and **DRCS** to be generated from a source external to the ARM926EJ-S processor.

DRDMAEN

DRDMAEN is the DMA enable signal. When asserted it indicates that the DMA values should be used to produce **DRCS** and **DRADDR** rather than those from the internal ARM926EJ-S TCM controller.

DRDMACS

DRDMACS is used to generate **DRCS** when **DRDMAEN** is asserted. Because of the way the **DRDMACS** signal is combined with the internal ARM926EJ-S TCM controller, it is not valid to assert **DRDMAEN** without **DRDMACS** asserted unless the internal TCM controller is idle (**DRIDLE** asserted). The relationship between these signals is shown in Table 5-1.

Table 5-1 Relationship between DMDMAEN, DRDMACS, and DRIDLE

DRDMAEN	DRDMACS	DRIDLE	DRCS
1	1	0	1
1	0	0	Unknown
1	1	1	1
1	0	1	0

DRDMAADDR[17:0]

DRDMAADDR is used as the source for **DRADDR** whenever **DRDMAEN** is asserted.

5.2.2 Instruction TCM signals

The instruction side TCM signals are almost identical to the DTCM signals. All the signals on the DTCM have an equivalent on the instruction side.

- Control signals
  - **IRCS**
  - **IRWAIT**
  - **IRIDLE**
- Address and attribute signals
  - **IRSEQ**
  - **IRADDR[17:0]**
  - **IRWBL[3:0]**
  - **IRnRW**
- Data signals
  - **IRRD[31:0]**
  - **IRWD[31:0]**
- DMA signals
  - **IRDMAEN**
  - **IRDMACS**
  - **IRDMAADDR[17:0]**.

5.2.3 Differences between DTCM and ITCM

There are three differences between the DTCM and ITCM interfaces:

- DMA to ITCM should not occur be performed unless **IRIDLE** is asserted
- Only back-to-back transfers on the DTCM can be marked as sequential. On the ITCM idle cycles may occur before requests marked as sequential.
- Sequential write transfers will not occur on the ITCM.

5.3 TCM interface bus cycle types and timing

The TCM bus interface is pipelined to enable back-to-back accesses to TCM memory with zero wait states. For each TCM access there is one request cycle and one or more data cycles. Figure 5-1 shows a multi-cycle data side TCM access.

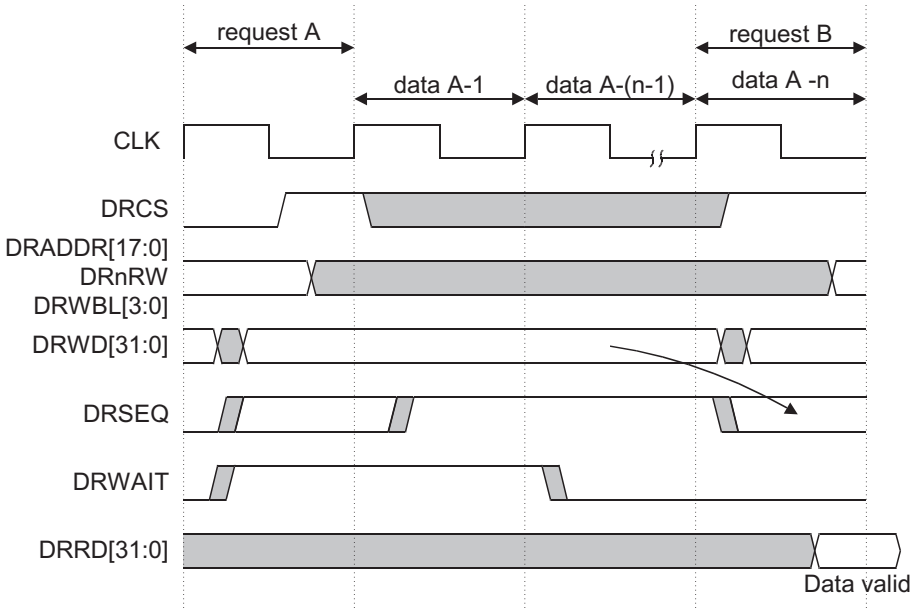


Figure 5-1 Multi-cycle data side TCM access

The first cycle is a request cycle (request A), where all of the TCM interface output signals are valid. The TCM subsystem responds on **DRWAIT**, indicating that the access will not complete in the following cycle. The cycle following the request cycle (data A-1) is the first waited data cycle. In this cycle the values of **DRADDR**, **DRnRW**, and **DRWBL** are no longer valid and their value is non-deterministic, and **DRSEQ** is asserted. The value on **DRWD** remains the same if the access is a write. As in the request cycle **DRWAIT** indicates if the access will complete in the following cycle. In the penultimate data cycle (data A-n-1) **DRWAIT** is deasserted indicating that the access will complete in the next cycle. For write accesses, this cycle is the last cycle where **DRWD** remains valid. If the last data cycle of the access (data A-n) is a read then **DRRD** contains valid read data. Because of the pipelined nature of the interface, the last data cycle of one access can overlap a request cycle of the next access.

5.3.1 Zero wait state timing

For zero wait state accesses the timing of the TCM interface corresponds to the timing of a standard SRAM component, with minimal interfacing logic required. Figure 5-2 shows examples of zero wait state accesses on the ITCM interface corresponding to instruction fetches. All accesses are reads.

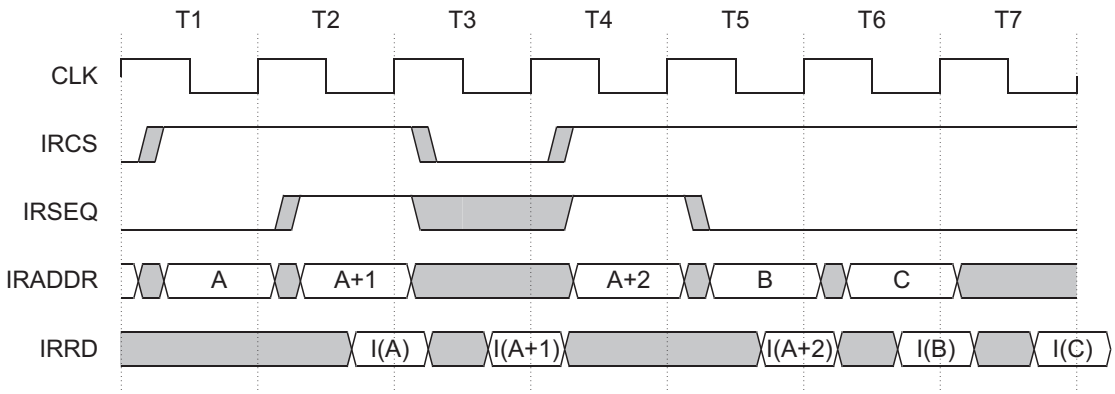


Figure 5-2 Instruction side zero wait state accesses

In cycle T1, a nonsequential request is made to address A.

In cycle T2, a sequential request is made to A+1 and data for the access to A is returned.

In cycle T3, no request is made and data is returned for the access to A+1

In cycle T4, a sequential request is made to A+2.

In cycle T5, a nonsequential request is made to address B and data is returned for the access to A+2.

In cycle T6, a nonsequential request is made to address C and data is returned for the access to B

It is important to note that, for the ITCM interface, cycles of a sequential request cycle do not necessarily occur in consecutive bus cycles. Any number of idle request cycles can occur between two requests, with the second request being marked as being sequential. The DTCM interface only produces sequential requests during consecutive bus cycles.

Figure 5-3 on page 5-10 shows examples of data side zero wait state accesses.

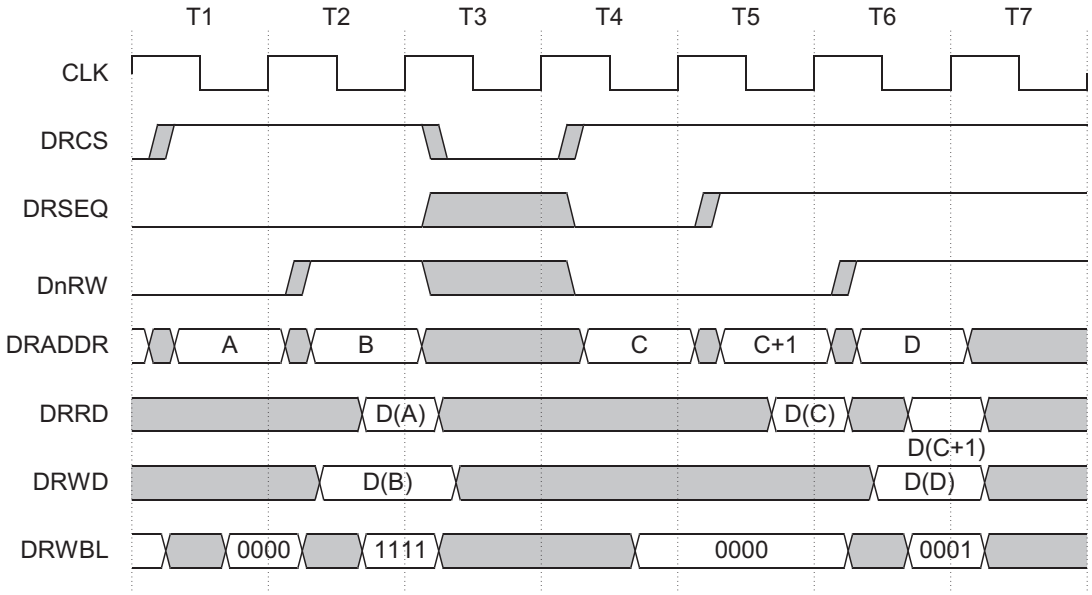


Figure 5-3 Data side zero wait state accesses

In cycle T1, a nonsequential read request is made to address A.

In cycle T2, a nonsequential word write request is made to address B and data is returned for the access to A.

In cycle T3, no request is made.

In cycle T4, a nonsequential read request is made to address C.

In cycle T5, a sequential read request is made to address C+1 and data is returned for the access to C.

In cycle T6, a nonsequential byte write request is made to address D.

5.3.2 DMA access to zero wait state TCM

For DMA accesses to zero wait state memories, the TCM DMA interface can be used which enables an alternative source of address and chip-select to be passed through to the TCM memories without impacting timing. Figure 5-4 on page 5-11 shows the relationship between **DRDMAEN**, **DRDMACS**, **DRDMAADDR**, **DRADDR** and **DRCS**.

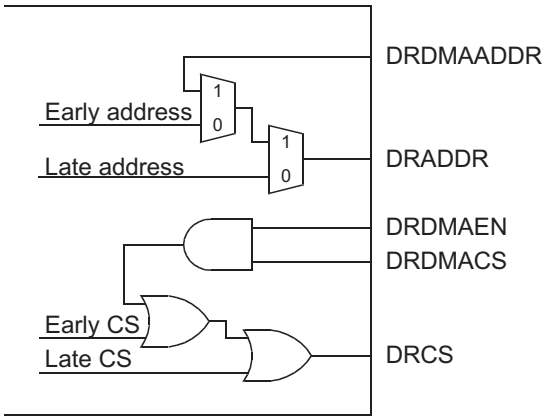


Figure 5-4 Relationship between DRDMAEN, DRDMACS, DRDMAADDR, DRADDR and DRCS

Internal to the ARM926EJ-S processor there are multiple sources for both the address and chip-select outputs. The address and chip-select outputs of the TCM interface are timing critical, however not all of the internal sources are timing critical. By combining the DMA inputs with non-critical address and chip-select signals, DMA can be done without impacting timing on these outputs. All other TCM interface outputs are non timing critical, and can be multiplexed externally.

The logic used to combine the DMA chip-select with the internal chip-select signals is designed so that if the DMA inputs are selected then the DMA chip-select is also asserted. If this is not the case then the chip-select output value is non-deterministic unless it is known that the TCM interface is an idle state, as indicated by the **DRIDLE** or **STANDBYWFI** signals.

Figure 5-5 on page 5-12 shows an example of how DMA accesses interact with normal DTCM accesses.

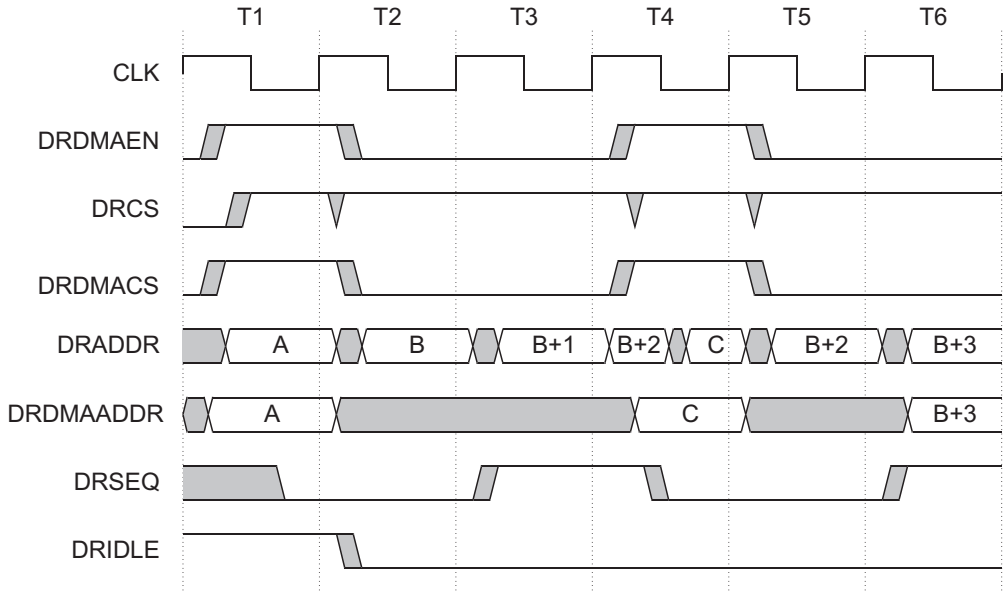


Figure 5-5 DMA access interaction with normal DTCM accesses

In cycle T1, the ARM926EJ-S internal TCM controller is idle and **DRIDLE** is asserted. **DRDMAEN** is asserted, and consequently the value of **DRDMAADDR** is propagated onto **DRADDR**, and **DRCS** is asserted (**DRDMACS** = 1). **DRSEQ** is forced LOW.

In cycle T2, the ARM926EJ-S internal TCM controller is no longer idle, and **DRIDLE** is deasserted. A nonsequential request is made to address B.

In cycle T3, a sequential request is made to address B+1 and **DRSEQ** is asserted

In cycle T4, the ARM926EJS internal TCM controller attempts to output values corresponding to a sequential request to address B+2. **DRDMAEN** is asserted, and the value of **DRADDR** and **DRSEQ** change accordingly. The ARM926EJ-S TCM controller is stalled.

In cycle T5, **DRDMAEN** is deasserted and the ARM926EJ-S TCM controller re-issues the request to address B+2. Because of the intervening DMA access, **DRSEQ** is deasserted for the repeated request.

In cycle T6, a sequential request is made to address B+3 and **DRSEQ** is re-asserted.

DMA accesses can be made to the ITCM using the **IRDMAEN**, **IRDMACS**, and **IRDMAADDR** signals but, unlike the DTCM, simultaneous access by the ARM926EJ-S and DMA is not supported. This means that ITCM DMA must not take place while executing code from the ITCM.

5.3.3 Multi-cycle access timing

If non zero wait state memory is used for TCM, then the **DRWAIT/IRWAIT** signals are used to wait the ARM926EJ-S. The wait information for a data cycle is pipelined so that the value of **DRWAIT/IRWAIT** pertains to the following data cycle, which corresponds to the request cycle for the first data cycle. If there is no active TCM access then the value on **DRWAIT/IRWAIT** is ignored. This allows the wait signals to be generated speculatively.

Figure 5-6 shows how the speculative generation of **IRWAIT** can be used to generate a single wait state for every ITCM access.

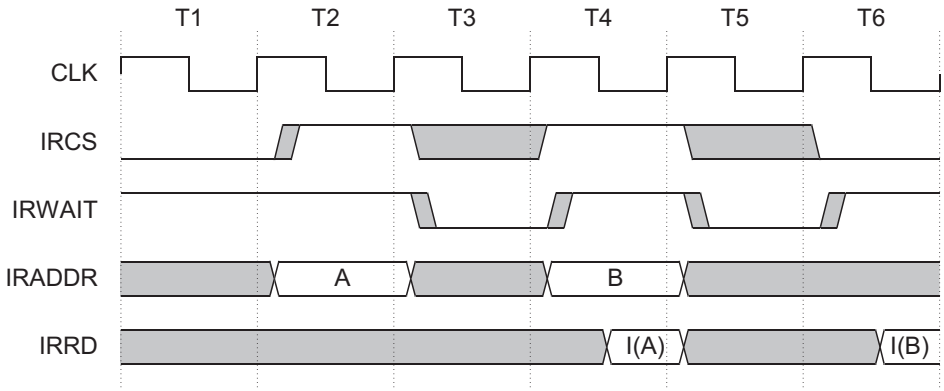


Figure 5-6 Generating a single wait state for ITCM accesses using IRWAIT

In cycle T1, **IRWAIT** is asserted but no request is made.

In cycle T2, **IRWAIT** is asserted and a request is made.

In cycle T3, **IRWAIT** is deasserted indicating that the access to A will complete in the following cycle.

In cycle T4, **IRWAIT** is asserted and a request is made. The access to A completes.

In cycle T5, **IRWAIT** is deasserted indicating that the access to B will complete in the following cycle.

In cycle T6, **IRWAIT** is asserted. No request is made. The access to B completes.

The logic required for the above example corresponds to the two-state state machine shown in Figure 5-7 on page 5-14.

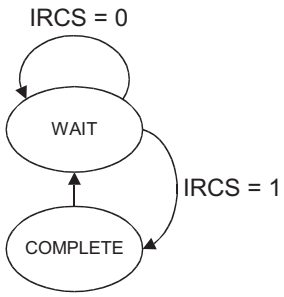


Figure 5-7 State machine for generating a single wait state

In the WAIT state **IRWAIT** is asserted. In the COMPLETE state **IRWAIT** is deasserted.

Certain types of memories can have different access penalties depending on whether an access is sequential or nonsequential. The **IRSEQ/DRSEQ** signals indicate if an access is sequential in the request cycle for an access, and are held HIGH during waited cycles. This behaviour enables a loopback arrangement, where the **SEQ** output can be fed directly back into the **WAIT** input through an inverter to produce a single cycle wait state for nonsequential accesses as shown in Figure 5-8.

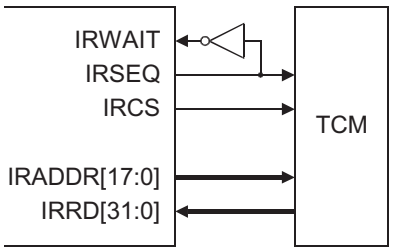


Figure 5-8 Loopback of SEQ to produce a single cycle wait state

The cycle timing of the circuit shown in Figure 5-8 is shown in Figure 5-9 on page 5-15.

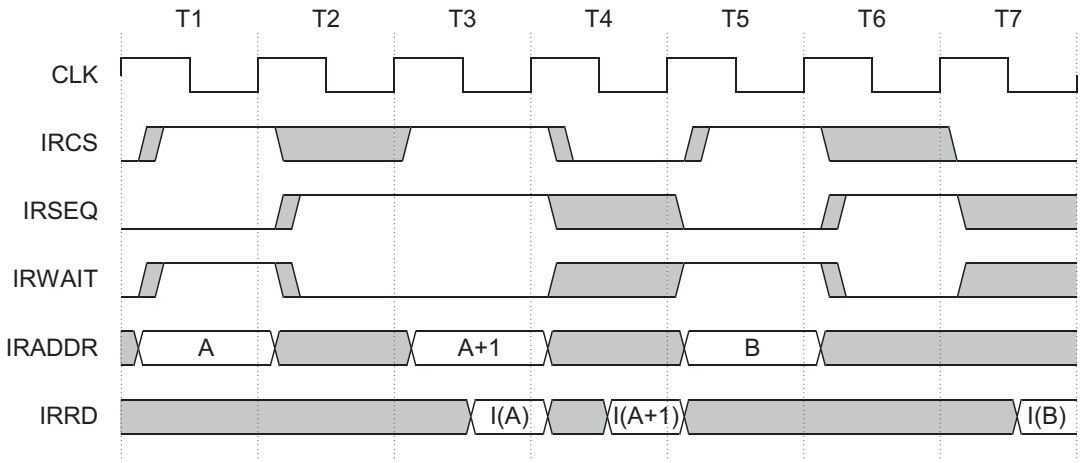


Figure 5-9 Cycle timing of loopback circuit

In cycle T1, a nonsequential request is made to address A and **IRWAIT** is asserted.

In cycle T2, **IRSEQ** is asserted because of the wait-state. **IRWAIT** is deasserted. **IRCS** is unknown.

In cycle T3, the access to A completes and a sequential request is made to A+1. **IRSEQ** is HIGH and **IRWAIT** is LOW

In cycle T4, the access to A+1 completes. No new request is issued. The values of **IRSEQ** and **IRWAIT** are unknown.

In cycle T5, a nonsequential request is made to address B and **IRWAIT** is asserted

In cycle T6, **IRSEQ** is asserted because of the wait-state. **IRWAIT** is deasserted, **IRCS** is unknown.

In cycle T7, the access to B completes.

For systems that also require DMA access to non zero wait state memories, the **WAIT** signal is used to stall the ARM92EJ-S processor for both wait states and DMA arbitration. Apart from the **DRWD/IRWD** write data signals, the information required to perform an access is only valid during the request cycle for that access. If a TCM access is postponed because of DMA, this information must be captured at the end of the request cycle.

Figure 5-10 on page 5-16 shows an example of a system where DMA access is required to a memory that has a single wait state for nonsequential accesses.

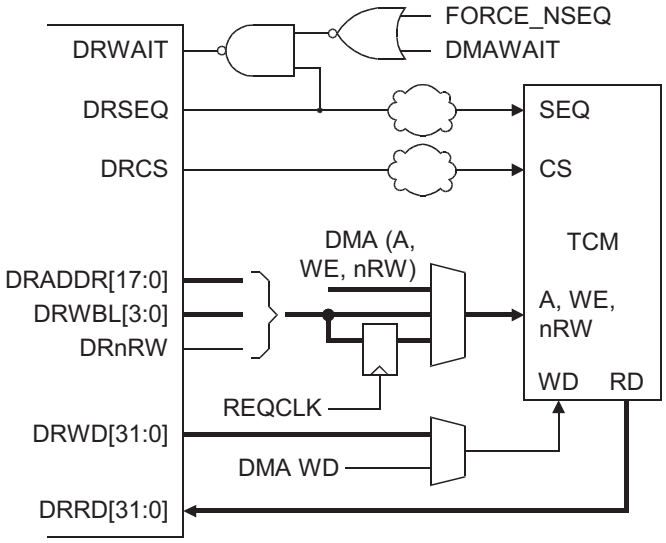


Figure 5-10 DMA with single wait state for nonsequential accesses

The logic used to generate **DRWAIT** uses both the loopback scheme using **DRSEQ** for inserting a wait state for a nonsequential request, and an additional signal **DMAWAIT**, for stalling during DMA accesses. The **FORCE_NSEQ** signal is an override signal used to force the ARM926EJ-S access to be treated as nonsequential because of an intervening DMA access.

The **A**, **WE** and **nRW** inputs to the TCM are either sourced directly from the ARM926EJ-S TCM interface, from the DMA controller, or from the capture register (clocked by **REQCLK**) if the ARM926EJ-S access is postponed because of DMA activity.

The cycle timing of the circuit shown in Figure 5-10 is shown in Figure 5-11 on page 5-17.

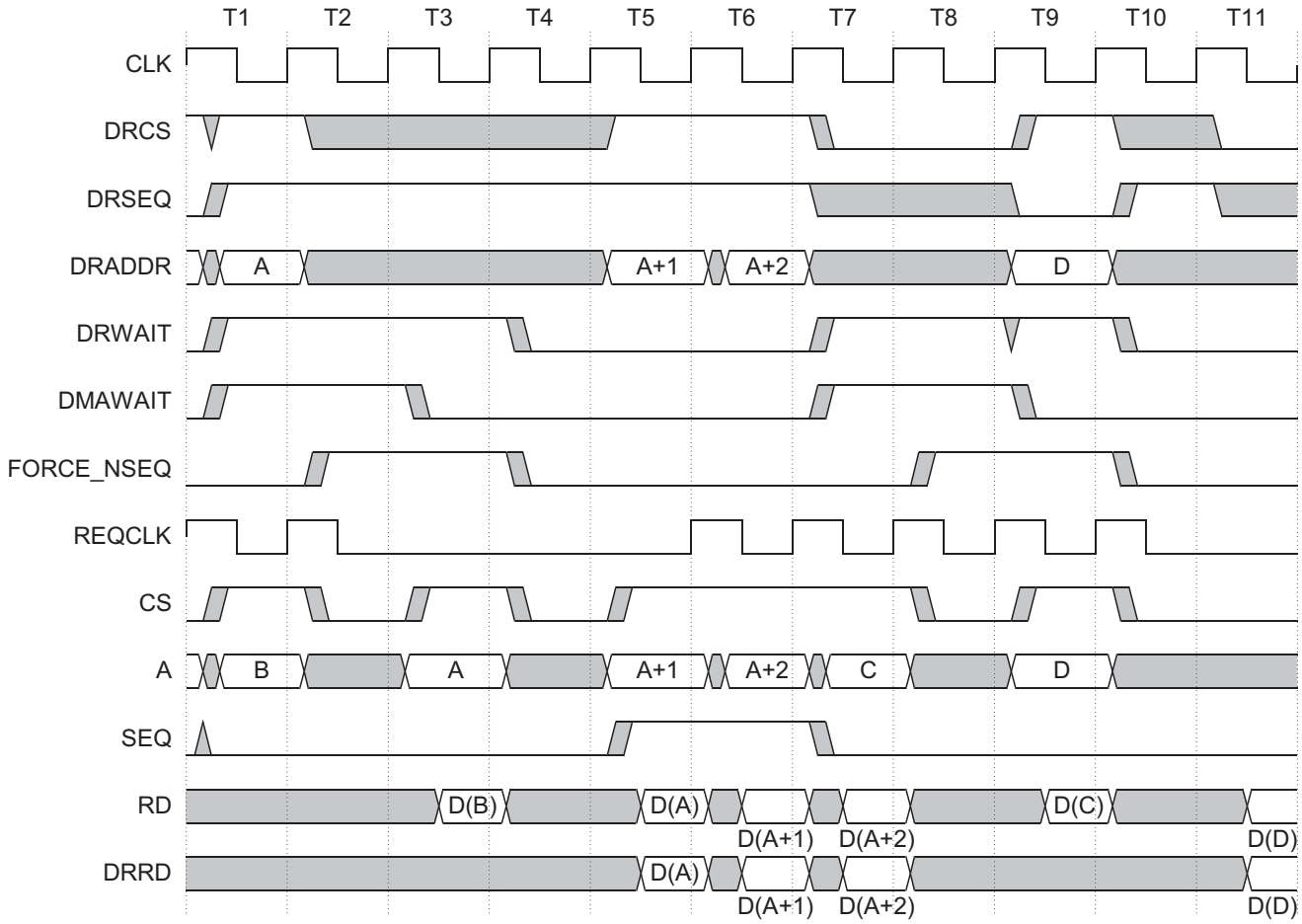


Figure 5-11 Cycle timing of circuit with DMA and single wait state for nonsequential accesses

In cycle T1, the ARM926EJ-S initiates a sequential request to address A and the DMA gains ownership of the TCM. **DRWAIT** is asserted because of **DMAWAIT**. The **CS**, **A**, **WE** signals for the TCM are sourced from the DMA. The values of **DRADDR**, **DRBWL** and **DnRW** are registered.

In cycle T2, the DMA access is still active (two cycle nonsequential access). **DRWAIT** is held HIGH because of **DMAWAIT**.

In cycle T3, the DMA access completes and **DMAWAIT** is deasserted. The access attributes captured at the end of T1 are used to generate the **CS**, **A** and **WE** signals for the TCM. **DRWAIT** is asserted because of **FORCE_NSEQ**.

In cycle T4, **FORCE_NSEQ** is deasserted causing **DRWAIT** to be deasserted indicating that the access will complete in the next cycle.

In cycle T5, the access to A completes. A sequential request is made to A+1. There is no DMA activity.

In cycle T6, the access to A+1 completes. A sequential request is made to A+2. There is no DMA activity

In cycle T7, the access to A+2 completes. No request is made and **DRCS** is deasserted. A DMA access to address C starts and **DRWAIT** is asserted using **DMAWAIT**.

In cycle T8, **DRWAIT** remains HIGH because of DMA access. No request is made, and **DRCS** remains LOW.

In cycle T9, the DMA access to C completes. A nonsequential request is made to address D.

5.4 TCM programmer's model

After reset, the behavior of the TCMs is controlled by the state of the TCM Region Register, CP15 c9.

5.4.1 Enabling the ITCM

The ITCM can automatically be enabled at reset using the **INITRAM** pin. If **INITRAM** is held HIGH during system reset, and the **VINITHI** pin is deasserted, the ITCM is enabled with the ITCM region base set to `0x0`. This allows boot code to be run from the ITCM. Boot code must be pre-loaded into the TCM for this to be useful.

If **INITRAM** is LOW during system reset and the ITCM is disabled, the ITCM can be enabled by writing to the ITCM Region Register. See *TCM Region Register c9* on page 2-29.

Note

If INITRAM = 1 and VINITHI = 1, the ITCM is enabled at system reset but the ARM926EJ-S processor boots from `0xFFFF0000`.

5.4.2 Enabling the DTCM

Unlike the ITCM there is no way of automatically enabling the DTCM at reset. The DTCM can only be enabled by writing to the DTCM Region Register. See *TCM Region Register c9* on page 2-29.

5.4.3 Disabling the ITCM

Disable the ITCM by clearing bit 0 of the ITCM Region Register, CP15 c9. This register must be written using a read-modify-write operation.

5.4.4 Disabling the DTCM

Disable the DTCM by clearing bit 0 of the DTCM Region Register, CP15 c9. This register must be written using a read-modify-write operation.

5.4.5 Cachable and bufferable attributes

All MMU page table entries used to map TCM address space must be marked noncachable. This is required for forward compatibility.

5.5 TCM interface examples

This section contains the following examples:

- *Zero-wait-state RAM example*
- *Producing byte writable memory using word writable RAM*
- *Multiple banks of RAM example* on page 5-21.

Note

Most of the examples in this section are for the DTCM interface. These are also applicable to the ITCM interface.

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

5.5.1 Zero-wait-state RAM example

Figure 5-12 shows the simplest RAM interface where the RAM block is constructed from a single word-wide RAM that has byte write control. The TCM interface can connect directly to the RAM block. This is a zero-wait-state memory so **DRWAIT** is tied LOW.

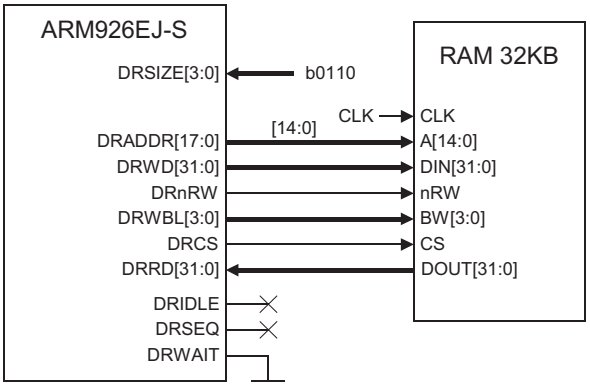


Figure 5-12 Zero wait state RAM example

5.5.2 Producing byte writable memory using word writable RAM

If byte-write RAM is not available, four banks of byte-wide RAM must be used as shown in Figure 5-13 on page 5-21.

The rules for connecting four RAM blocks are:

- Each byte-wide RAM has the same address and chip-select control as the word-wide RAM.
- The following connections must be made:
  - **DRWBL[0]**, **DRWD[7:0]**, and **DRRD[7:0]**, connect to RAM byte 0
  - **DRWBL[1]**, **DRWD[15:8]**, and **DRRD[15:8]**, connect to RAM byte 1
  - **DRWBL[2]**, **DRWD[23:16]**, and **DRRD[23:16]**, connect to RAM byte 2
  - **DRWBL[3]**, **DRWD[31:24]**, and **DRRD[31:24]**, connect to RAM byte 3.

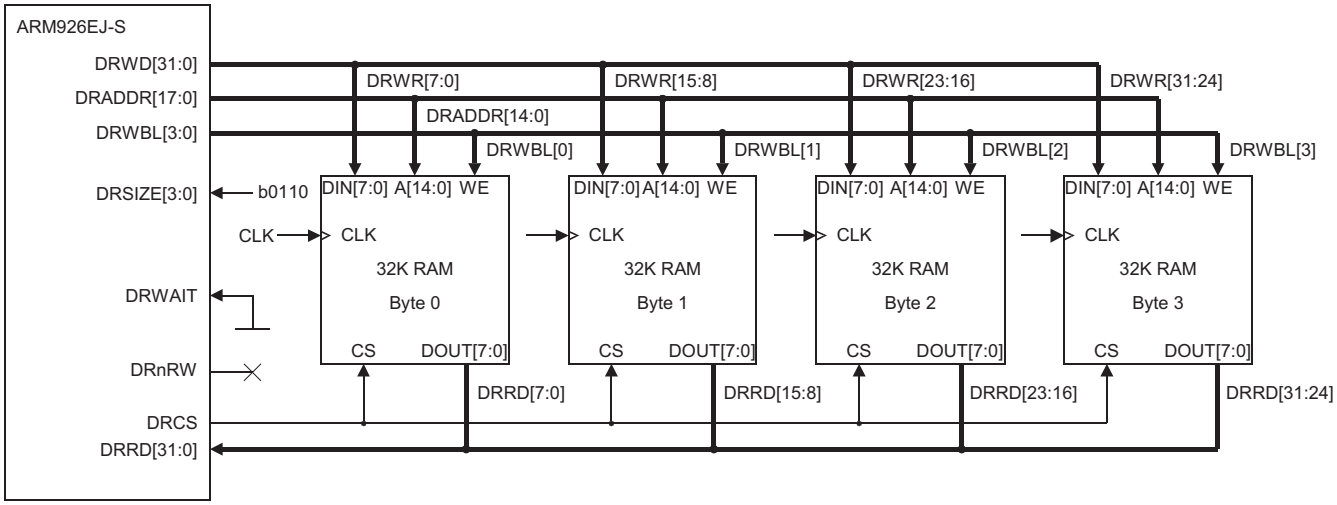


Figure 5-13 Byte-banks of RAM example

Note

In little-endian mode, **DRWBL[0]** indicates the LSB of the word and **DRWBL[3]** indicates the MSB. In big-endian mode, **DRWBL[3]** indicates the LSB of the word and **DRWBL[0]** indicates the MSB.

5.5.3 Multiple banks of RAM example

If you have to create a large memory out of smaller RAM blocks, there are two methods for doing this:

- If minimizing power consumption is more important than a fast design, you must follow the example in *Optimizing for power* on page 5-22.

- If a fast design is more important than minimizing power consumption, you must follow the example in *Optimizing for speed* on page 5-23.

The rules for producing memory out of smaller RAM blocks are:

- There must be an even number of RAM blocks b (b = 2, 4, 8, for example)
- Each RAM block must be the same size.
- If the address width of the required memory size is n bits, the address port of the smaller RAM blocks is $m = n\text{-}(\log_b/\log_2)$ bits wide.
- Address bits [m-1:0] are applied to all the RAM blocks.
- Address bits [n-1:m] are gated with **DRCS** for a power optimized solution, or with **IRnRW** for a speed optimized solution.
- Pipelined address bits [n-1:m] are used to select the correct RAM read data.

Optimizing for power

Figure 5-14 on page 5-23 shows how to produce a large memory from two smaller RAM blocks if you are optimizing for power. Separate chip select control is required for each RAM block:

CS_bank0 = ~DRADDR[14] & **DRCS**
CS_bank1 = DRADDR[14] & **DRCS**

This ensures that only the RAM being accessed is enabled, minimizing power consumption.

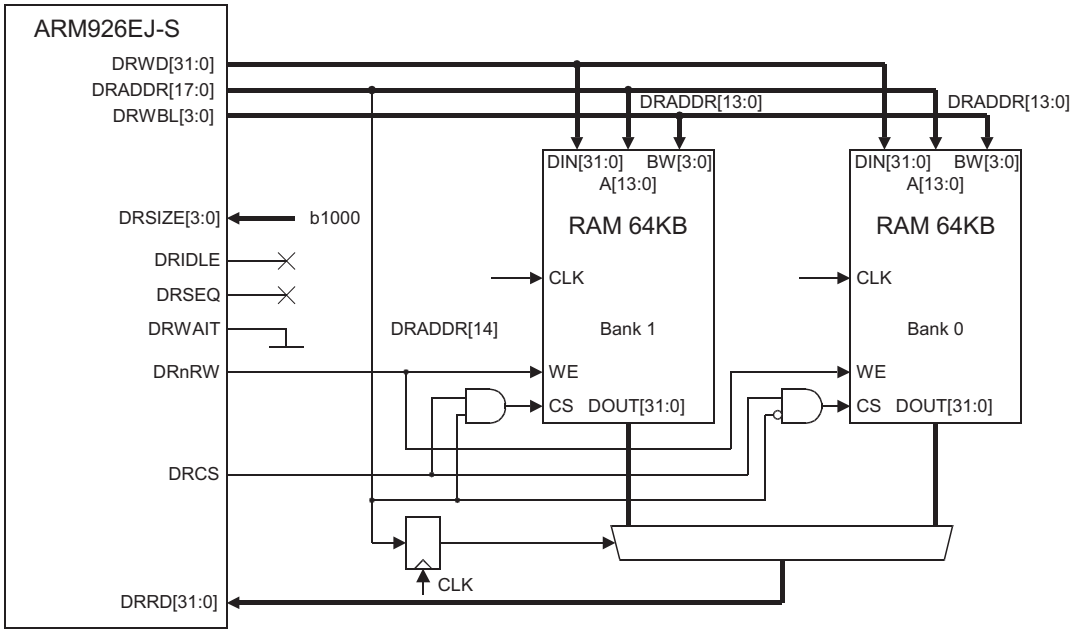


Figure 5-14 Optimizing for power

Optimizing for speed

Figure 5-15 on page 5-24 shows how to produce a large memory from two smaller RAM blocks if you are optimizing for speed. Separate write enable control is required for each RAM block:

WE_bank0 = ~DRADDR[14] & **DRnRW**

WE_bank1 = DRADDR[14] & **DRnRW**

No logic is added to the critical **DRCS** path, but both RAMs are enabled whenever **DRCS** is asserted, resulting in higher power consumption.

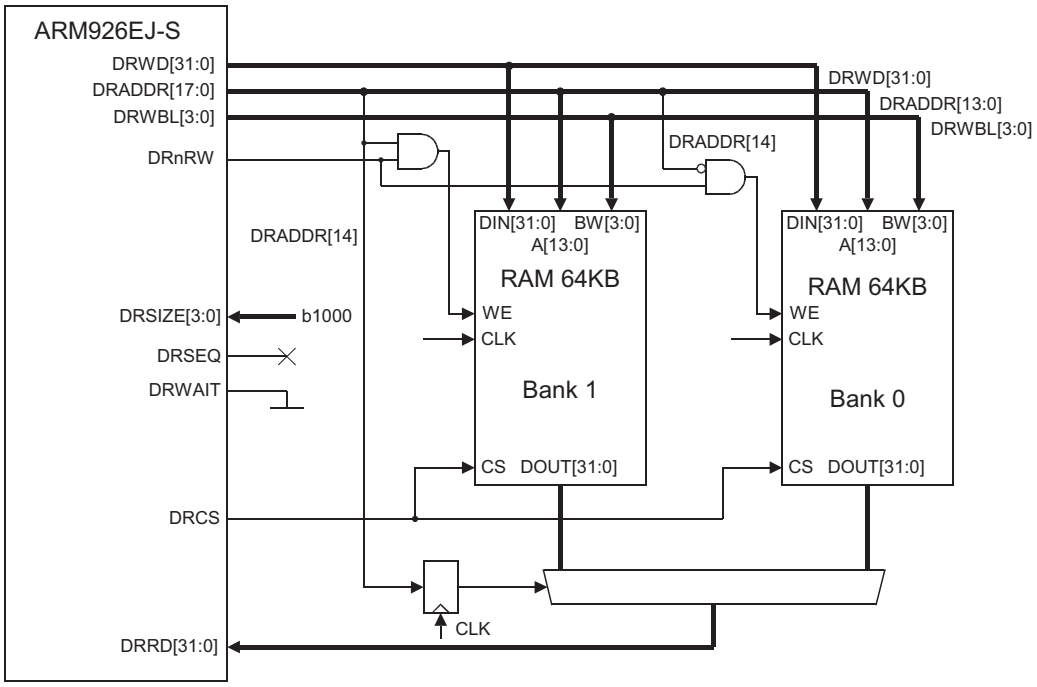


Figure 5-15 Optimizing for speed

5.5.4 Sequential ROM example

The diagram in Figure 5-16 on page 5-25 shows an example of a TCM sub-system that uses wait states for nonsequential accesses. The ROM used to hold instructions can cycle at the same frequency as the ARM926EJ-S processor it is interfaced to. However, the memory access time for the ROM (time from chip-select/address to data out) is not fast enough to be directly interfaced to the ARM926EJ-S processor.

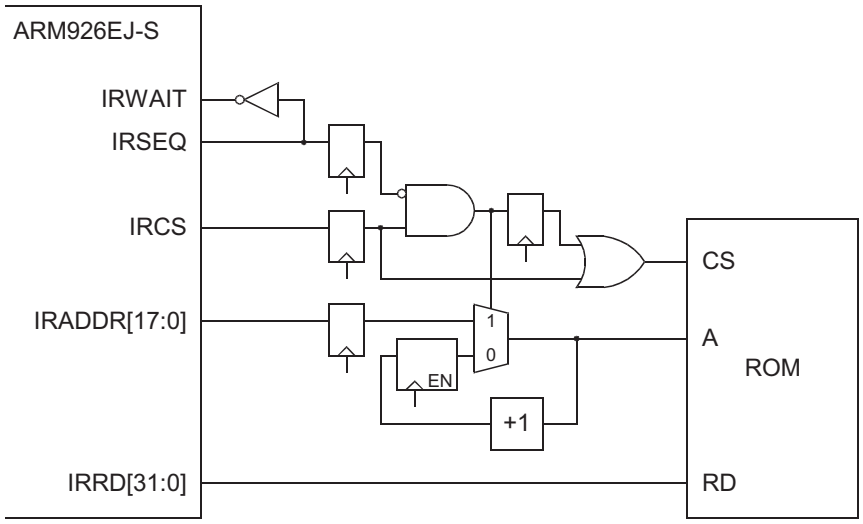


Figure 5-16 TCM subsystem that uses wait states for nonsequential accesses

The address and chip-select inputs to the ROM are pipelined with respect to the ARM926EJ-S TCM interface outputs. An address incrementer is used to generate sequential addresses. The output of the incrementer is captured at the end of every cycle where the ROM CS chip select is active. The address source for the ROM is switched over to the registered version of **IRADDR** when a nonsequential access occurs.

Figure 5-17 on page 5-26 shows the timing of the ROM address, chip-select, and read data relative to the ARM926EJ-S TCM interface signals. The address supplied to the ROM can either be behind, in sync with, or ahead of **IRADDR**, depending on the type of memory access and the presence of idle cycles.

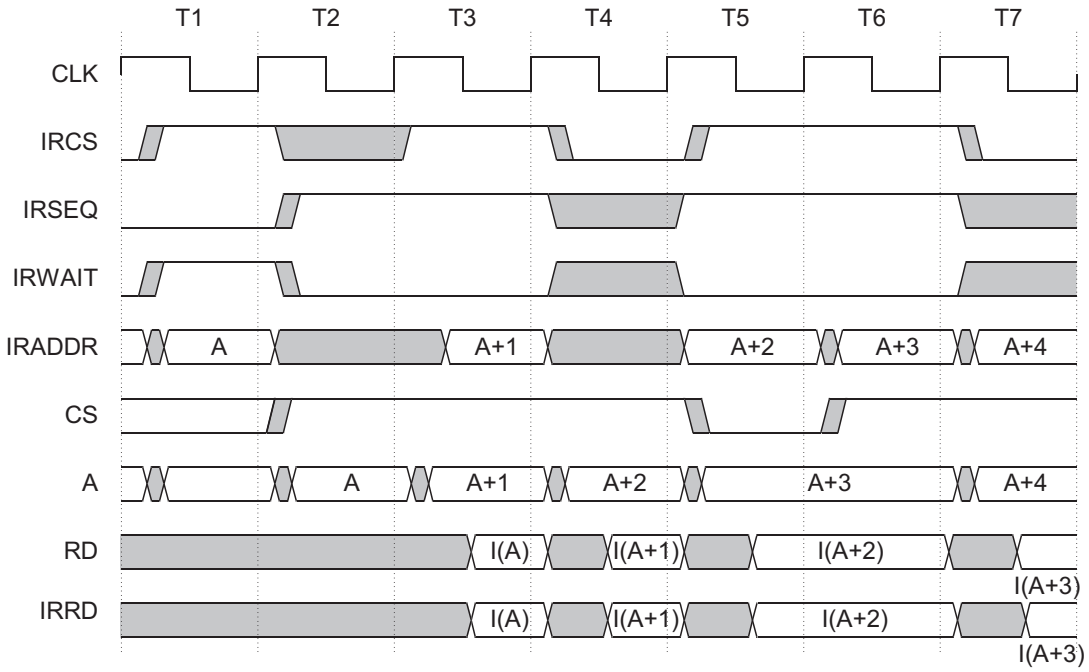


Figure 5-17 Cycle timing of circuit that uses wait states for non sequential accesses

5.5.5 DMA interface example

Figure 5-18 on page 5-27 shows an example TCM subsystem using the DMA interface. The signal driving **DRDMAEN** is connected to both the **DRDMAEN** and **DRDMACS** inputs. It is also used to control the multiplexing of the non timing critical signals (**WBL**, **nRW**, and **WD**), although this is not shown for clarity.

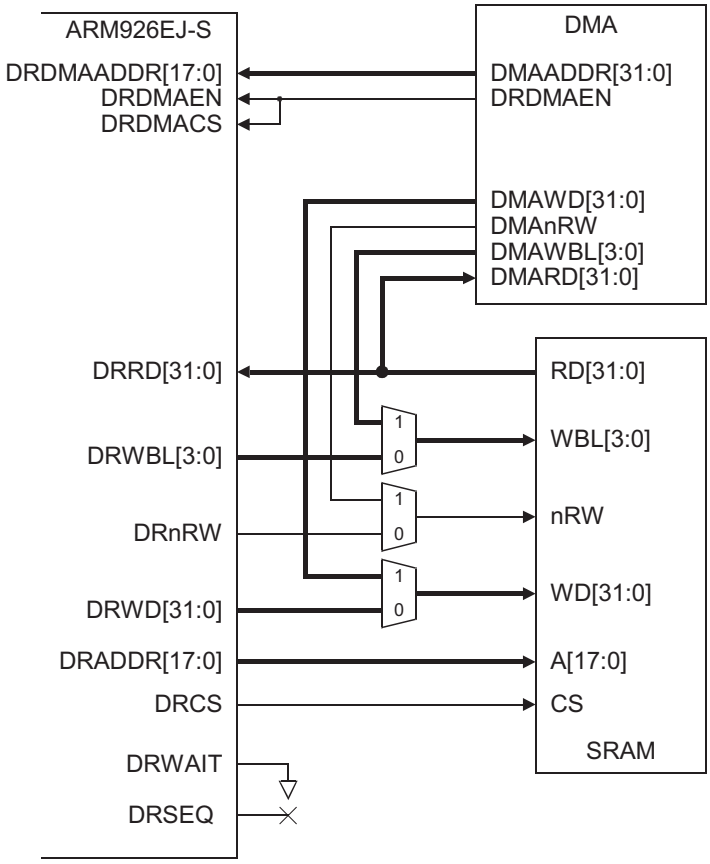


Figure 5-18 TCM subsystem that uses the DMA interface

5.5.6 Integrating RAM test logic

The memory used to implement TCM might require some form of test access, typically by a BIST controller. Generally this is done by adding a collar of multiplexors around the memory inputs. However, this method will add undesirable delays to the chip select and address signals. This can be avoided by using the DMA interface to perform the multiplexing of address and chip-select values. This is shown in Figure 5-19 on page 5-28.

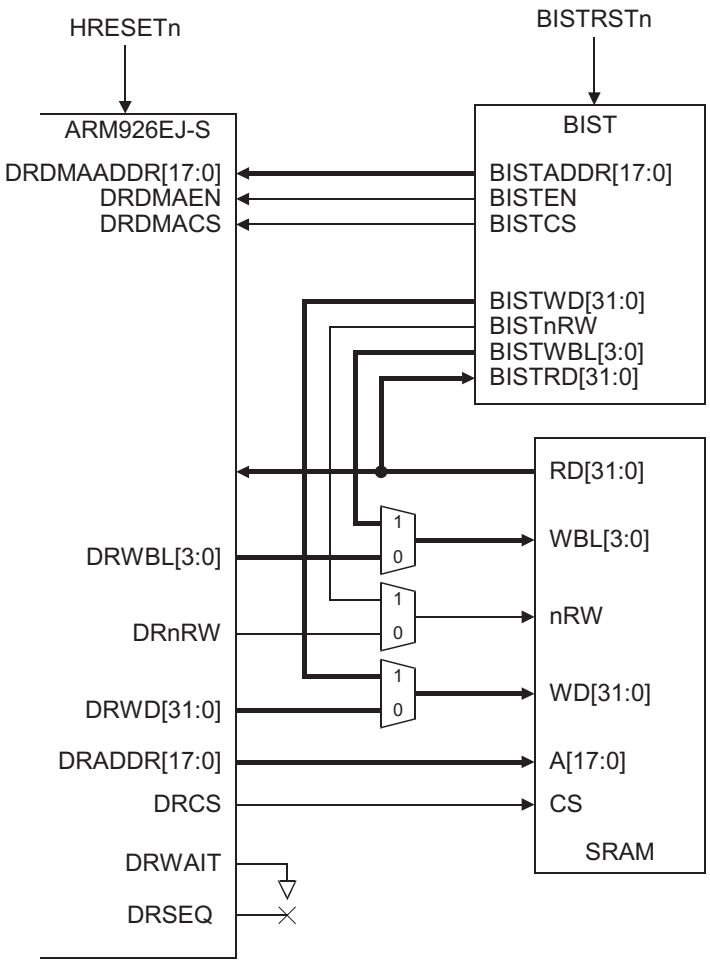


Figure 5-19 TCM test access using BIST

This is similar to the previous DMA example. However, for BIST testing it is necessary for the BIST controller to be able to force the memory chip select to both HIGH and LOW values. This requirement means that it is necessary to hold the ARM926EJ-S core in such a state that the internal value of the chip select is guranteed to be LOW. This can be done by holding the ARM926EJ-S in reset (**HRESETn** LOW) during TCM memory BIST testing. Note that this requires that **HRESETn** cannot also be used as a reset control to the BIST controller.

5.6 TCM access penalties

The data side of the ARM926EJ-S core can access the ITCM. To maximize the performance of the ITCM, data read accesses to the ITCM are pipelined. The ARM926EJ-S core is stalled for two cycles to enable the pipeline read to complete. This is the only ARM926EJ-S TCM interface stall scenario. The inclusion of a write buffer in the TCM controller has eliminated all other sources of potential stalling for zero wait state TCM.

5.7 TCM write buffer

Each TCM interface has a two word entry write buffer. This is required to de-pipeline the address and data values produced by the ARM9EJ-S core so that non-speculative writes can be made to memory with SRAM characteristics peformed without introducing stall cycles.

The ARM9EJ-S core read requests take priority over writes, and consequently TCM transactions can be out of order with respect to instruction execution. If a read access occurs to a location that also has a corresponding entry in the write-buffer, then data is forwarded from the write-buffer. If it is necessary to ensure that all outstanding writes have completed on the TCM interface then the CP15 drain write buffer instruction can be used (`MCR p15, 0, Rd, c7, c10, 4`). This instruction does not complete execution until all oustanding buffered writes (TCM and AHB) have been completed.

To guarantee that the TCM write buffers have been drained and that all outstanding requests on the TCM interface have completed, a drain write buffer instruction must be used prior to disabling either of the TCM regions.

5.8 Using synchronous SRAM as TCM memory

If you use SRAM to implement TCM memory, then your library RAM must meet the following requirements:

- It must be synchronous. All timings must be relative to the rising clock edge.
- It must have a chip select (RAM enable).
- The RAM outputs must always be valid. They must not be tristated.
- Byte write control is required.
- RAM setup times must be less than 10-15% and access times must be less than 40-50% of the target cycle time. Violation of these requirements results in a slower design. Setup and access times can be balanced by skewing the clock to the RAM.

Ideally each TCM can be constructed from single RAM blocks. However, this is not always possible for the following reasons:

- If your RAM does not have byte write control, you must construct the word-wide RAM out of four byte-wide RAMs. See *Producing byte writable memory using word writable RAM* on page 5-20.
- If your compiler cannot produce a single RAM block that is the required size, or if a single RAM block does not meet the timing requirements. In these cases, you must produce the RAM out of two or more blocks of smaller RAM. See *Multiple banks of RAM example* on page 5-21.

Ideally, your RAM block can connect directly to the ARM926EJ-S TCM interface. However, this is not always possible, and additional logic is required in the following cases:

- All TCM signals are driven as active HIGH. If your RAM requires active LOW signals, you must add inverters to create the active LOW signals.
- If power control logic is required.
- If a RAM is non single-cycle, or hardware DMA arbitration is required, logic is required to drive the appropriate wait signal.

Note

DRADDR is always a word address. **DRWBL** is used as a byte lane strobe to select the appropriate byte of the addressed word on writes. Reads are always word-wide.

5.9 TCM clock gating

If the ARM926EJ-S processor is not currently running code from a TCM region, the idle signal for that TCM (**DRIDLE** for DTCM, **IRIDLE** for ITCM) is asserted. This indicates that a TCM access will not be performed in that cycle, enabling you to stop the TCM clock. If no clock stopping is required, you can ignore the idle signals.

You can also use the idle signal to disable power to the RAMs if you require more stringent power control. Removing the RAM power invalidates the RAM contents so you must only do this if the TCMs are not being used and do not contain valid data.

Chapter 6
Bus Interface Unit

This chapter describes the ARM926EJ-S *Bus Interface Unit* (BIU). It contains the following sections:

6.1 About the bus interface unit

The ARM926EJ-S *Bus Interface Unit* (BIU) arbitrates and schedules AHB requests. The BIU contains separate masters for both instruction and data access enabling complete AHB system flexibility. Separate masters enable multi-layer AHB (see the *Multi-layer AHB Overview*) and multi-AHB systems to be implemented, giving the benefit of increased overall bus bandwidth and a more flexible system architecture. Each master is a fully compliant AHB bus master and implements the master functions as defined in the *AMBA Specification (Rev 2.0)*.

To increase system performance, write buffers are used to prevent AHB writes stalling the ARM926EJ-S system. For more details, see Chapter 4 *Caches and Write Buffer*.

The data BIU AHB signals are prefixed with D, and the instruction BIU signals are prefixed with I.

6.2 Supported AHB transfers

The ARM926EJ-S processor supports a subset of AHB transfers. The permitted AHB transfers are described in:

- *Memory map*
- *Transfer size*
- *Mapping of level one and level two (AHB) attributes* on page 6-5
- *Byte and halfword accesses* on page 6-6
- *AHB system considerations* on page 6-6
- *AHB clocking* on page 6-10.

6.2.1 Memory map

The ARM926EJ-S processor is a cached processor with two AHB interfaces. It is a key system design issue that the D side must be able to access the same memory as the I side, with the same memory map. This is required not only to load code, but to enable access to PC-relative literal pools, and for SWI and emulated instruction handlers to work.

Note

This is unlike some Harvard arrangements whereby the I-bus can be connected to the ROM and the D-bus only connected to RAM/peripherals.

6.2.2 Transfer size

The ARM926EJ-S processor performs all AHB accesses as single word, bursts of four words, or bursts of eight words. Any ARM9EJ-S core requests that are not 1, 4, or 8 words in size are split into packets of these sizes. For example, an STM of 12 words is performed on the AHB as a burst of 8 followed by a burst of 4. If a burst is interrupted because of either a Split or Retry response, or by removal of **HGRANT**, then the burst is completed as single transfers. Consequently the ARM926EJ-S processor only uses a subset of the possible **HBURST** and **HSIZE** encodings.

Table 6-1 shows the **HBURST** encodings that the ARM926EJ-S processor uses, and the operations that perform each burst size.

Table 6-1 Supported HBURST encodings

HBURST[2:0]	Description	Operation
Single	Single transfer	Single transfer of word, halfword, or byte: • data write (NCNB, NCB, WT, or WB that has missed in DCache) • data read (NCNB or NCB) • NC instruction fetch (prefetched and non-prefetched) • page table walk read • continuation of a burst that either lost grant or received a Split/Retry response.
Incr4	Four-word incrementing burst	Half-line cache write-back. Instruction prefetch, if enabled. Four-word burst NCNB, NCB, WT, or WB write.
Incr8	Eight-word incrementing burst	Full line cache write-back. Eight-word burst NCNB, NCB, WT, or WB write.
Wrap8	Eight-word wrapping burst	Cache linefill.

Note

Incr4 and Incr8 bursts can be aligned to any word boundary.

The ARM926EJ-S processor performs all Thumb instruction fetches as word-wide transfers on the AHB. See *Mapping of level one and level two (AHB) attributes* on page 6-5.

All burst reads and writes are performed by the ARM926EJ-S processor as word-wide transfers (**HSIZE[2:0]** = 010). Single reads and writes are performed as byte (**HSIZE[2:0]** = 000), halfword (**HSIZE[2:0] = 001**), or word wide transfers (**HSIZE[2:0]** = 010).

6.2.3 Mapping of level one and level two (AHB) attributes

Table 6-2 shows the **IHPROT[3:0]** and **DHPROT[3:0]** mappings for memory operations.

Table 6-2 IHPROT[3:0] and DHPROT[3:0] attributes

Operation		IHPROT[3:0] or DHPROT[3:0]	Description
DCache linefill		{1,1,Priv[a],1}	CB, data access
ICache linefill		{1,1,Priv[a],0}	CB, opcode fetch
Page table walk (data)		{1,1,1,1}	Page table walk caused by a TLB miss for a data access
Page table walk (instruction)		{1,1,1,0}	Page table walk caused by a TLB miss for an instruction fetch
Instruction fetch		{0,0,Priv[a],0}	NCNB opcode fetch
		{0,1,Priv[a],0}	NCB opcode fetch
Data access	LDR/STR	{0,0,Priv[a],1}	NCNB
		{0,1,Priv[a],1}	NCB
	STR	{1,1,Priv[a],1}	WT/WB
DCache write-back		{1,1,1,1}	-

a. Priv indicates if the access was caused by a privileged (1) or User (0) access issued by the ARM9EJ-S core.

Table walk reads that occur because of TLB misses for both data and instructions are performed using the data side bus master. The state of **DHPROT[0]** can be used to identify if a table walk is caused by an instruction fetch miss in the TLB:

DHPROT[0] = 0 Indicates that an instruction fetch miss caused the page table walk.

DHPROT[0] = 1 Indicates that a data access miss caused the page table walk.

Attributes specified for LDR instructions also apply for LDM, LDRD, and LDC operations. Similarly those for STR apply for STM, STRD, and STC operations.

A DCache write-back can be caused either by an eviction during a linefill, or an explicit cache clean operation.

6.2.4 Byte and halfword accesses

This section describes byte and halfword accesses for:

- *Address alignment*
- *Thumb instruction fetches*
- *Endianness and byte lane indication*.

Address alignment

The ARM926EJ-S BIU performs address alignment checking and aligns AHB addresses to the necessary boundary. 16-bit accesses are aligned to halfword boundaries, and 32-bit accesses to word boundaries.

Thumb instruction fetches

All instruction fetches, irrespective of the state of the ARM9EJ-S core, are made as 32-bit accesses on the AHB. If the ARM9EJ-S core is in Thumb state, then two instructions can be fetched at a time.

Endianness and byte lane indication

The *AMBA Specification (Rev 2.0)* does not specify any explicit support for endianness. The ARM926EJ-S processor provides a supplementary signal, **DHBL**, that indicates which bytes are to be updated for write transfers and which bytes should contain valid data for reads. This is created using the address, and the endianness of the access.

The **CFGBIGEND** signal indicates the current endianness setting used by the ARM9EJ-S core, and reflects the value held in CP15 c1 (see *Control Register c1* on page 2-12).

Because writes are buffered, the value of the **CFGBIGEND** signal might be inconsistent with **DHBL** if the write-buffer is not drained before changing the endianness setting in the control register.

DHBL is encoded in little-endian format. For example, a value of b0001 indicates byte 0 in little-endian mode, and byte 3 in big-endian mode.

6.2.5 AHB system considerations

This section describes AHB considerations for:

- *Single-layer AHB systems* on page 6-7
- *Multi-layer AHB systems* on page 6-7
- *Multi-AHB systems* on page 6-8

- *Memory coherency* on page 6-9.

Single-layer AHB systems

If the ARM926EJ-S processor is to be used in a single-layer AHB system, each of the two BIU masters must be treated as being unique.

The simplest way of integrating the two ARM926EJ-S bus masters into a single-layer AHB system is for each master to be a separate requestor into the AHB arbiter, the same as for any multi-master system. The data master normally has higher arbitration priority than the instruction master.

Note

The ARM926EJ-S instruction AHB interface does not perform locked transfers so **IHLOCK** is always driven LOW.

DHCLKEN and **IHCLKEN** must be tied together, as described in *AHB clocking* on page 6-10. If **HCLK** and **CLK** are the same frequency, **DHCLKEN** and **IHCLKEN** must be tied HIGH.

Because of the handover cycle when transferring ownership of the bus, a nongranted bus master incurs an extra cycle of latency to get onto the bus if the bus is currently idle. This means that if the data BIU is the default bus master, it can start AHB transactions a cycle earlier than the instruction BIU (nondefault bus master), which must wait for ownership of the bus to be handed over.

This cycle of latency only exists for the first transaction. If the instruction BIU is prefetching instructions, for example, it can perform back-to-back requests and maintain ownership of the bus until a higher priority bus master is granted.

Multi-layer AHB systems

Figure 6-1 on page 6-8 shows an example of a Multi-layer AHB system. In this example the I-interface labeled I-side, and the D-interface labeled D-side are configured through an interconnect matrix to have access to four slave ports. If the two AHB interfaces, known as layers, require access to the same slave at the same time, then an arbitration process within the interconnect matrix determines the layer that has the highest priority. Under this system D-side can have access to any slave port that I-side is not using at that time, which increases the overall bus bandwidth available.

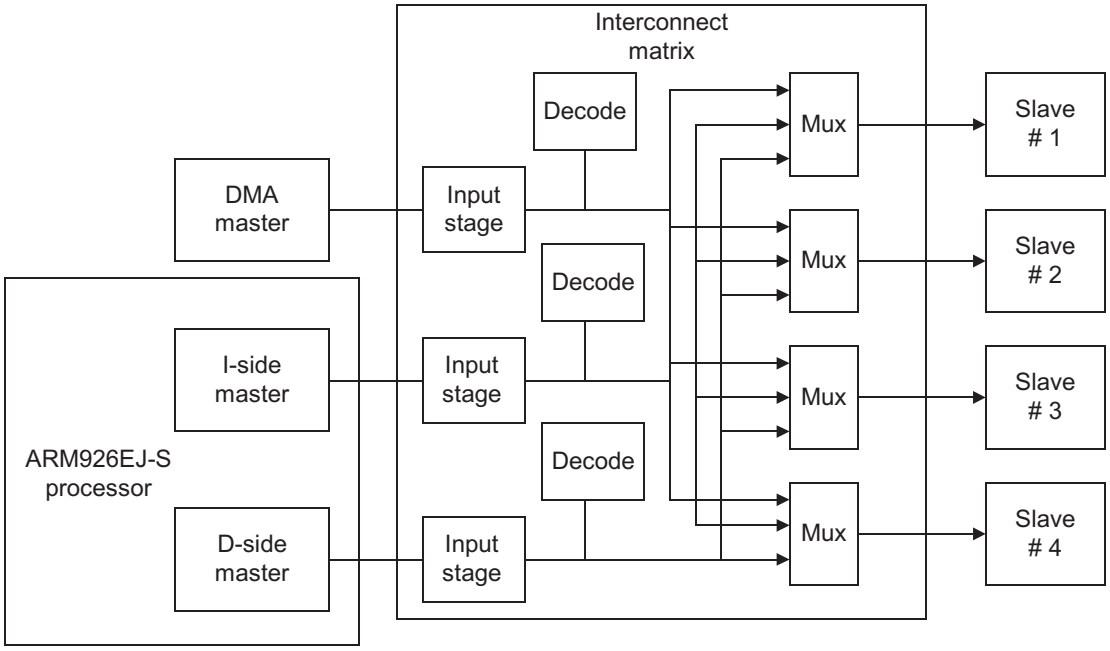


Figure 6-1 Multi-layer AHB system example

Multi-layer AHB is described in more detail in the *Multi-layer AHB Overview*.

Multi-AHB systems

It is possible that the ARM926EJ-S instruction and data AHB interfaces can be connected to separate AHB systems, although there must be a mechanism to support data side access to the instruction memory. Each AHB system can be running at different frequencies. The ARM926EJ-S processor is able to cope with this by providing two **HCLKEN** inputs:

- **DHCLKEN** is used to specify the rising **HCLK** edge for the system in which the data BIU is the master
- **IHCLKEN** is used to specify the rising **HCLK** edge for the system in which the instruction BIU is the master.

Figure 6-2 on page 6-9 shows an example of a Multi-AHB system.

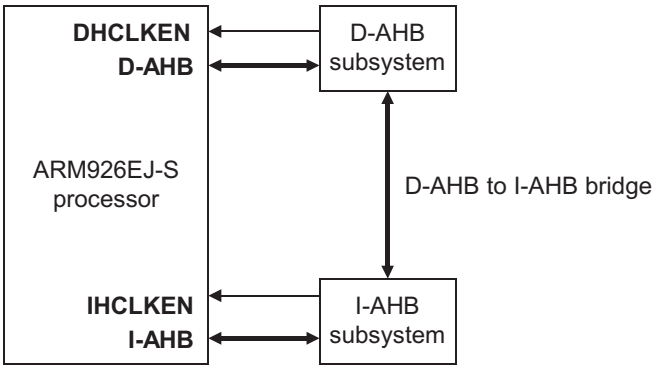


Figure 6-2 Multi-AHB system example

If both AHB systems operate at the same frequency, **DHCLKEN** and **IHCLKEN** must be tied together. See *AHB clocking* on page 6-10 for more details.

The AHB clock for each system, **HCLK1** and **HCLK2**, must be synchronized to the ARM926EJ-S clock signal **CLK**.

Memory coherency

Because of the Harvard nature of the ARM926EJ-S processor, instruction and data flow order cannot be guaranteed, and the arbitration order of the two masters can be considered to be arbitrary.

For single and multi-layer AHB systems:

- the arbitration priority of the two masters determines which of the masters is granted the bus, if both make a simultaneous request
- if the granted master receives a Split or Retry response, the other master can be granted the bus and complete its transaction before the split master completes.

For multi-AHB systems:

- the two systems can be operating at different frequencies
- the memory slaves can insert waits and/or issue Split or Retry responses.

If the sequence of flow is critical, in self-modifying code for example, an *Instruction Memory Barrier* (IMB) must be used to force coherency. See Chapter 9 *Instruction Memory Barrier* for more details.

6.2.6 AHB clocking

The ARM926EJ-S design uses a single clock, **CLK**. To run the ARM926EJ-S processor at a higher frequency than the AHB system bus, a separate AHB clock enable for each of the two bus masters is required (in a multi-AHB system each AHB system can be running at a different frequency):

DHCLKEN Is used to signify the rising edge of **HCLK** for the system data BIU bus master.

IHCLKEN Is used to signify the rising edge of **HCLK** for the system instruction BIU bus master.

Figure 6-3 shows the relationships between **CLK**, **HCLK**, **DHCLKEN**, and **IHCLKEN**.

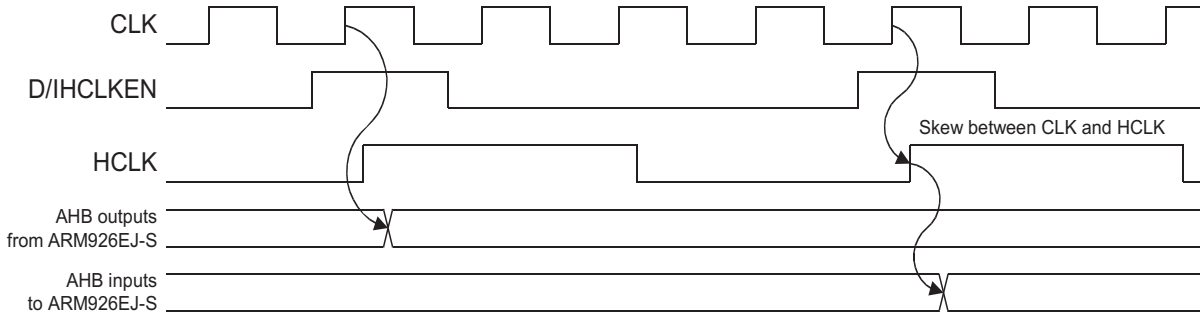


Figure 6-3 AHB clock relationships

For single and multi-layer AHB systems, **DHCLKEN** and **IHCLKEN** must be tied together. If **HCLK** and **CLK** are the same frequency, the relevant **HCLKEN** input (or inputs) must be tied HIGH.

CLK and **HCLK** must be synchronous. The skew between **CLK** and **HCLK** must be minimized.

6.2.7 External Abort limitations

Only certain types of accesses cause an External Abort if an Error response is returned for an AHB transfer. These are:

- page table walk
- noncached read
- nonbuffered write
- noncached read-lock-write (SWP).

For all other types of access (cache linefills, writeback evictions, buffered writes), an Error response is ignored.

If the ARM926EJ-S processor is to be used in a system which has to be tolerant to soft errors in external memory, then both soft error detection and correction must be done in hardware at the time the AHB transfer is made. The **DHREADY** and **IHREADY** signals can be used to extend the transfer until corrected data is available.

Chapter 7
Noncachable Instruction Fetches

This chapter describes noncachable instruction fetches in the ARM926EJ-S processor. It contains the following section:

7.1 About noncachable instruction fetches

The ARM926EJ-S processor performs speculative noncachable instruction fetches to increase performance. Speculative instruction fetching is enabled at reset. This can be disabled using bit 16 in the debug state register CP15 c15 (see *Test and Debug Register c15* on page 2-36). If prefetching is disabled only instruction fetches issued directly by the ARM9EJ-S core result in instruction fetches on the AHB interface.

The following subsection is divided into:

- *Uses of noncachable code*
- *Self modifying code*
- *AHB behavior* on page 7-3.

7.1.1 Uses of noncachable code

Although noncachable code performance has been improved compared with other ARM9 family cached cores, it is still recommended that the ICache is used in preference, where practical.

Noncachable code has previously been used for boot loaders of operating systems and for preventing cache pollution. It is worth noting that the ICache can be enabled without the MMU being enabled (see Chapter 4 *Caches and Write Buffer*), and that cache pollution can be controlled using the cache lockdown register (see *Cache Lockdown and TCM Region Registers c9* on page 2-26).

7.1.2 Self modifying code

A four-word buffer is used to hold speculatively fetched instructions. Only sequential instructions are fetched speculatively, and in the event of the ARM9EJ-S core issuing a nonsequential instruction fetch, the contents of the buffer are discarded (flushed). In situations where the contents of the prefetch buffer might become invalid during a sequence of sequential instruction fetches by the ARM9EJ-S core (for example, turning the MMU on or off, or turning on the ICache), the prefetch buffer is also flushed. This avoids the requirement for an explicit *Instruction Memory Barrier* (IMB) operation to be performed, except when self-modifying code is used. Because the prefetch buffer is flushed when the ARM9EJ-S core issues a nonsequential instruction fetch, a branch instruction (or equivalent) can be used to implement the required IMB behavior. This is illustrated by the following code sequence:

```
LDMIA   R0,{R1-R5}          ; load code sequence into R1-R5
ADR     R0,self_mod_code
STMIA   R0,{R1-R5}          ; store code sequence (nonbuffered region)
B       self_mod_code       ; branch to modified code
self_mod_code:
```


This IMB implementation only applies to the ARM926EJ-S processor running code from a noncachable region of memory. If code is run from a cachable region of memory, or a different device is used then a different IMB implementation is required. IMBs are described in Chapter 9 *Instruction Memory Barrier*.

7.1.3 AHB behavior

If instruction prefetching is disabled, all instruction fetches appear on the AHB interface as single, nonsequential fetches.

If prefetching is enabled then instruction fetches either appear as bursts of four instructions, or as single, nonsequential fetches. No speculative instruction fetching is done across a 1KB boundary.

All instruction fetches, including those made in Thumb state, are word transfers (32 bits). In Thumb state a single-word instruction fetch reads two Thumb instructions, and a four-word burst reads eight instructions.

Chapter 8
Coprocessor Interface

This chapter describes the ARM926EJ-S coprocessor interface. It contains the following sections:

8.1 About the ARM926EJ-S external coprocessor interface

The ARM926EJ-S supports the connection of on-chip coprocessors to the ARM9EJ-S core through an external coprocessor interface. All types of coprocessor instructions are supported.

8.1.1 Overview

Coprocessors determine the instructions that they have to execute by using a *pipeline follower* in the coprocessor. As each instruction arrives from memory it enters both the ARM9EJ-S pipeline and the coprocessor pipeline. To avoid a critical path for the instruction being latched by the coprocessor, the coprocessor pipeline must operate one clock cycle behind the ARM9EJ-S core pipeline.

The two pipelines are synchronized by stalling the ARM9EJ-S core pipeline in its first Execute cycle whenever an external coprocessor instruction moves from the Decode to the Execute stage.

To enable coprocessors to continue execution of coprocessor data operations while the ARM9EJ-S core pipeline is stalled (for example, while waiting for a cache linefill to occur), the coprocessor receives the clock **CLK**, and a clock enable signal **CPCLKEN**. You can use these to produce a gated coprocessor clock with the circuit shown in Figure 8-1.

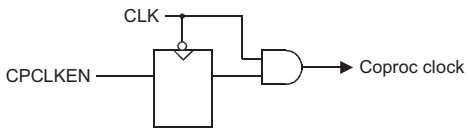


Figure 8-1 Producing a coprocessor clock

Figure 8-2 indicates the timing for these signals and when the coprocessor pipeline must advance its state.

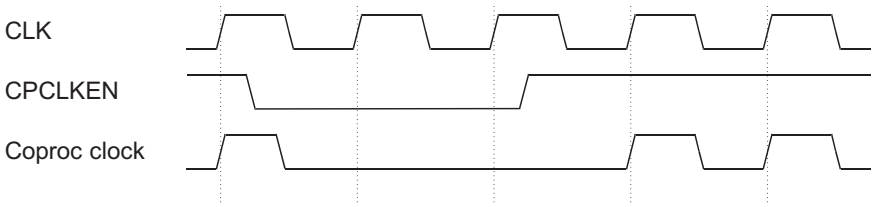


Figure 8-2 Coprocessor clocking

This is one technique for generating a clock that reflects the ARM9EJ-S core pipeline advancing. If **CPCLKEN** is LOW on the rising edge of **CPCLK** then the ARM9EJ-S core pipeline is stalled and the coprocessor pipeline should not advance.

Coprocessor instructions

There are three classes of coprocessor instructions:

LDC or STC Load coprocessor register from memory or store coprocessor register to memory.

MCR/MCRR or MRC/MRRC

Register transfer between the coprocessor and the ARM processor core.

CDP Coprocessor data operation.

Examples of how a coprocessor must execute these instruction classes are given in:

- *LDC/STC* on page 8-4
- *MCR/MRC* on page 8-6
- *CDP* on page 8-8.

8.2 LDC/STC

The cycle timing for this operation is shown in Figure 8-3.

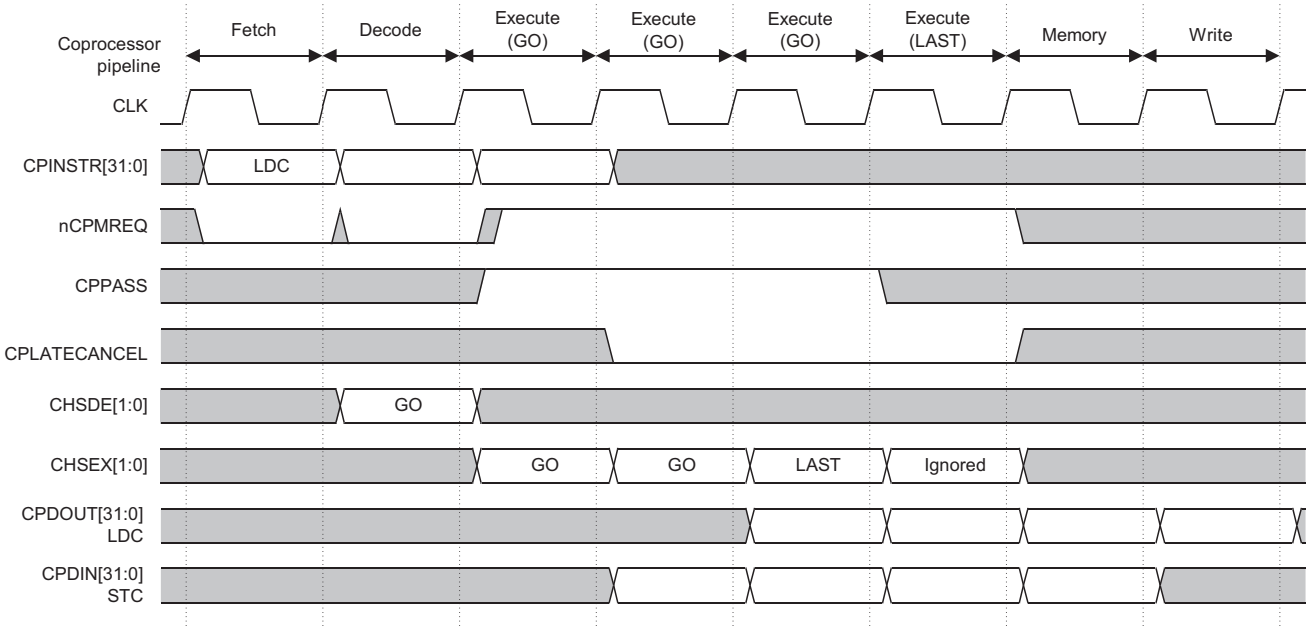


Figure 8-3 LDC/STC cycle timing

In Figure 8-3 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 ARM9EJ-S core performs the main decode off the rising edge of the clock during the Decode stage. From this, the core commits to executing the instruction and so performs an instruction fetch. The coprocessor instruction pipeline keeps in step with the ARM9EJ-S core by monitoring **nCPMREQ**. **nCPMREQ** is an active LOW signal that indicates if the ARM9EJ-S pipeline has advanced. **CPINSTR** is updated with the fetched instruction in the next cycle. This means that the instruction currently on **CPINSTR** must enter the Decode stage of the coprocessor pipeline, and that the instruction in the Decode stage of the coprocessor pipeline must enter its Execute stage.

During the Execute stage, the condition codes are combined with the flags to determine if the instruction 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 then **CPPASS** is asserted on every cycle until the coprocessor instruction is executed. If an interrupt occurs during busy-waiting then **CPPASS** is driven LOW and the coprocessor should stop the coprocessor instruction execution.

Another output, **CPLATECANCEL** is used to cancel a coprocessor instruction when the instruction preceding it caused a Data Abort. This is valid on the rising edge of **CLK** on the cycle after the first coprocessor Execute cycle of a coprocessor instruction.

On the rising edge of the clock the ARM9EJ-S core examines the coprocessor handshake signals **CHSDE[1:0]** and **CHSEX[1:0]**:

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

The handshake signals encode one of four states, as shown in Table 8-1.

Table 8-1 Handshake signal encoding

State	Value	Description
WAIT	00	If there is a coprocessor attached that can handle the instruction, but not immediately, then the coprocessor handshake signals are driven to indicate that the ARM9EJ-S core has stalled. This is known as the busy-wait condition. In the busy-wait condition, the ARM9EJ-S 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 then the undefined instruction trap is taken. If **CHSEX[1:0]** changes to GO or LAST then the instruction proceeds as described in GO. If an interrupt occurs then the ARM9EJ-S core is forced out of the busy-wait state. This is indicated to the coprocessor by the **CPPASS** signal going LOW. When the instruction is restarted the coprocessor must not commit to the instruction (that is, change any of the coprocessor state) until the coprocessor has seen **CPPASS** HIGH when the handshake signals indicate the GO or LAST condition.
GO	01	The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires another cycle of execution. Both the ARM9EJ-S core and the coprocessor must consider the state of the **CPPASS** signal before committing to the instruction. For an LDC or STC instruction, then the coprocessor instruction drives the handshake signals with GO when two or more words still have to be transferred. When only one further word is required the coprocessor drives the handshake signals with LAST.
ABSENT	10	If there is no coprocessor attached that can execute the coprocessor instruction, then the handshake signals indicate the ABSENT state and the ARM9EJ-S core takes the undefined instruction trap.
LAST	11	An LDC or STC instruction might transfer more than one word of data. If this is the case then, possibly after busy waiting, the coprocessor drives the coprocessor handshake signals with a number of GO states, followed by a LAST cycle. The LAST indicates that the next transfer is the final one. If there was only one transfer then the sequence would be [WAIT,[WAIT,...]],LAST.

8.3 MCR/MRC

These cycles look very similar to STC/LDC. An example with a busy-wait state is shown in Figure 8-4.

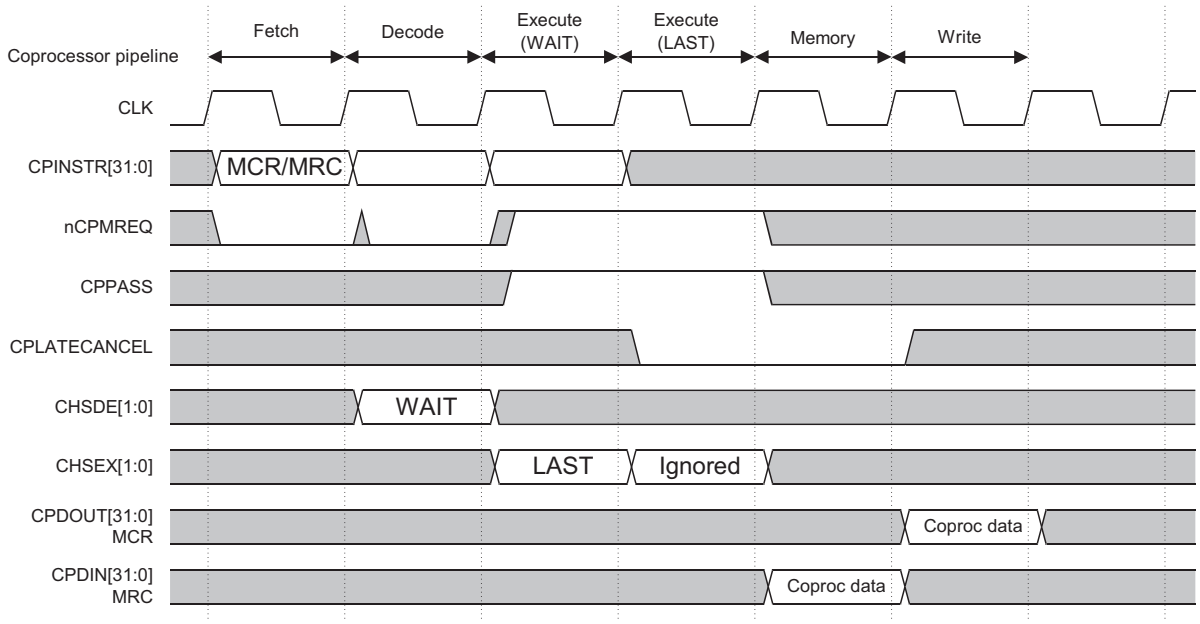


Figure 8-4 MCR/MRC cycle timing

First, **nCPMREQ** is driven LOW to indicate that the instruction on **CPINSTR** is entering the Decode stage of the pipeline. This coprocessor decodes the new instruction and drives **CHSDE[1:0]** as required.

In the next cycle, **nCPMREQ** is driven LOW to indicate that the instruction has now been issued to the Execute stage. If the condition codes pass and the instruction is to be executed, the **CPPASS** signal is driven HIGH and the **CHSDE[1:0]** handshake bus is examined (it is ignored in all other cases).

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 register data during the coprocessor Write stage. In the case of an MRC, **CPDIN[31:0]** is sampled at the end of the ARM9EJ-S memory stage and written to the destination register during the next cycle.

8.3.1 Interlocked MCR

If the data for an MCR operation is not available inside the ARM9EJ-S core pipeline during its first Decode cycle, then the ARM9EJ-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 remains there for a number of cycles before entering the Execute stage.

Figure 8-5 shows an example of an interlocked MCR.

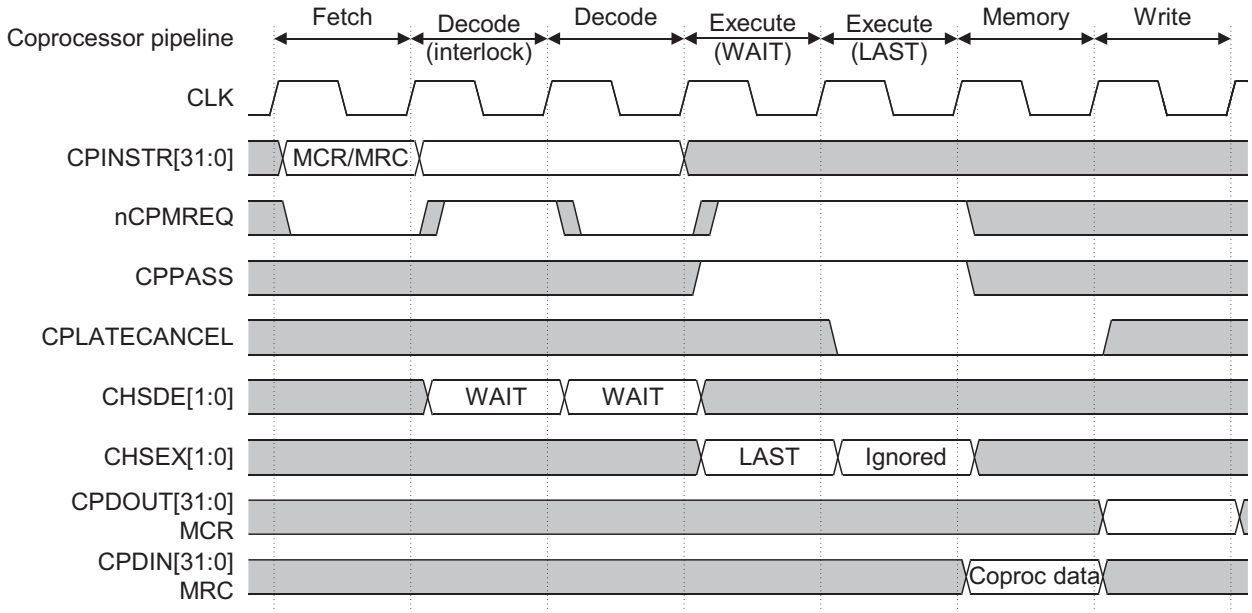


Figure 8-5 Interlocked MCR

8.4 CDP

CDP instructions usually execute in a single cycle. Like all the previous cycles, **nCPMREQ** is driven LOW to signal when an instruction is entering the Decode and then the Execute stage of the pipeline. If the instruction is to be executed then the **CPPASS** signal is driven HIGH during Execute. If the coprocessor can execute the instruction immediately it drives **CHSDE[1:0]** with LAST. If the instruction requires a busy-wait cycle, then the coprocessor drives **CHSDE[1:0]** with WAIT and then **CHSEX[1:0]** with LAST. Figure 8-6 shows a CDP that is canceled due to the previous instruction causing a Data Abort.

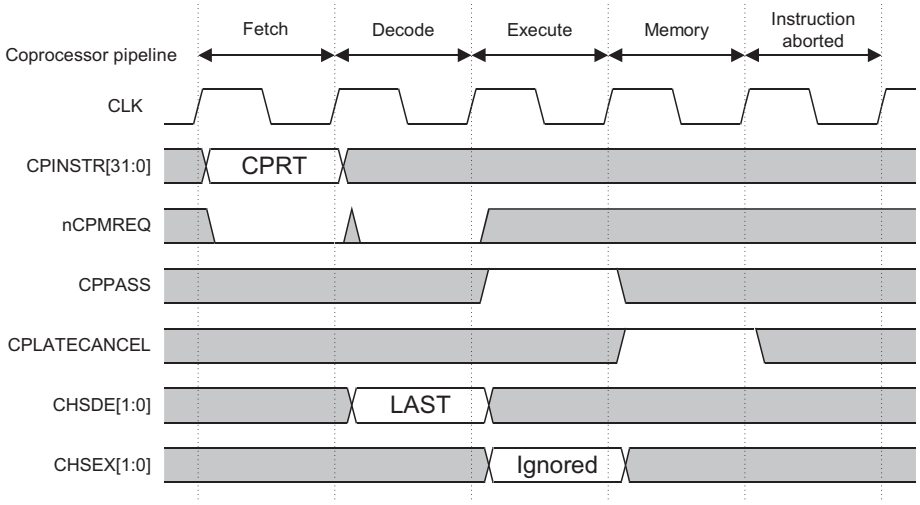


Figure 8-6 Latecanceled CDP

The CDP instruction enters the Execute stage of the pipeline and is signaled to execute by **CPPASS**. In the following phase **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.

Note

CPLATECANCEL can be asserted during the Memory cycle or during the Execute cycle. The coprocessor must be able to handle instruction aborts during these two stages.

8.5 Privileged instructions

The coprocessor might restrict certain instructions for use in privileged modes only. To do this, the coprocessor has to track the **nCPTRANS** output.

Figure 8-7 shows how **nCPTRANS** changes after a mode change.

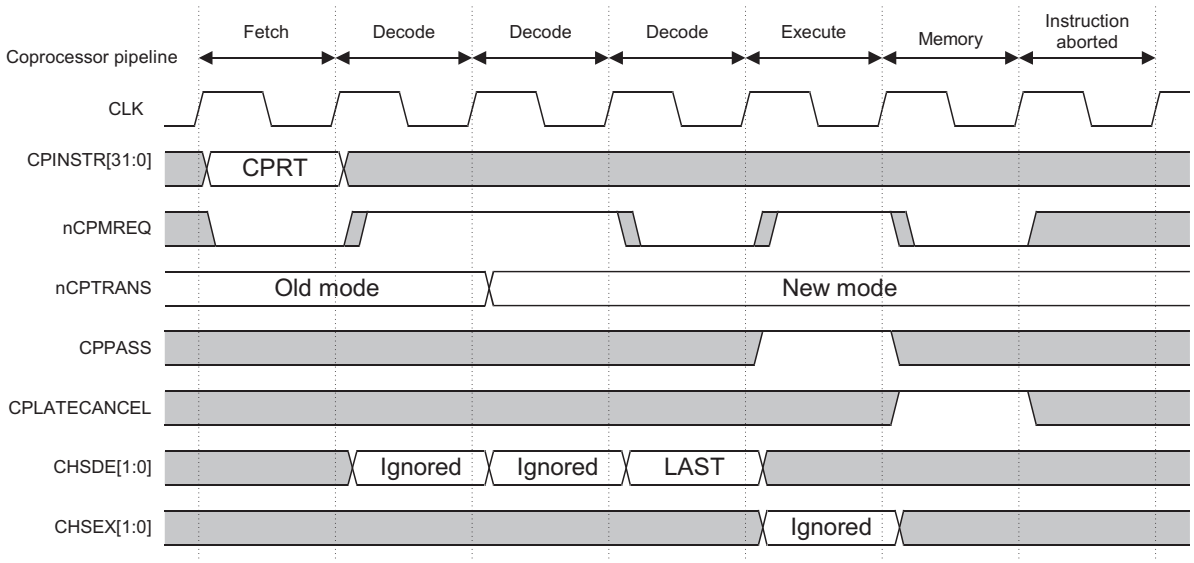


Figure 8-7 Privileged instructions

8.6 Busy-waiting and interrupts

The coprocessor is permitted to stall (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 on **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 required to keep the instruction in the busy-wait loop.

For interrupt latency reasons the coprocessor might be interrupted while busy-waiting, causing the instruction to be abandoned using **CPPASS**. The coprocessor must monitor the state of **CPPASS** during every busy-wait cycle. If it is HIGH the instruction must be executed. If it is LOW the instruction must be abandoned.

Figure 8-8 shows a busy-waited coprocessor instruction being abandoned due to an interrupt.

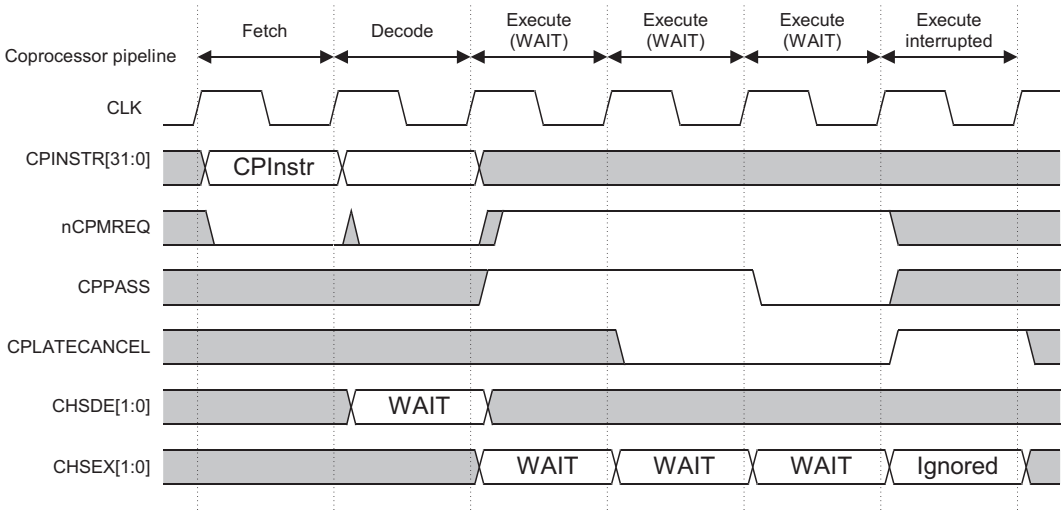


Figure 8-8 Busy waiting and interrupts

In Figure 8-8, **CPLATECANCEL** is also asserted as a result of the Execute interruption.

8.7 CPBURST

The **CPBURST** signal is used by the external coprocessor to indicate the number of words to be transferred in an LDC or STC operation. **CPBURST** is used by the ARM926EJ-S memory system to optimize LDC/STC instructions that access either noncachable or nonbufferable regions of memory. The encoding of **CPBURST** is shown in Table 8-2.

Table 8-2 CPBURST encoding

CPBURST[3:0]	Number of words to transfer
b0000	1 word or unknown
b0001	2 words
b0010	3 words
…	…
b1110	15 words
b1111	16 words

The encoding for a single word transfer and an unknown number of transfers is the same. If **CPBURST** is set to b0000 for an STC or LDC operation, and this results in an access to either a noncached or nonbuffered region of memory, then any resultant AHB bus transfers are performed as individual nonsequential accesses.

CPBURST is driven by external coprocessors in the same cycle as the **CHSDE** response. This must be driven to b0000 at all other times. An example of a transfer that uses **CPBURST** is shown in Figure 8-9 on page 8-12.

8.8 CPABORT

The **CPABORT** signal being asserted HIGH indicates that an LDC/STC instruction has aborted. **CPABORT** is asserted in the cycle after the Memory stage of the aborting LDC/STC instruction. This is shown in Figure 8-9.

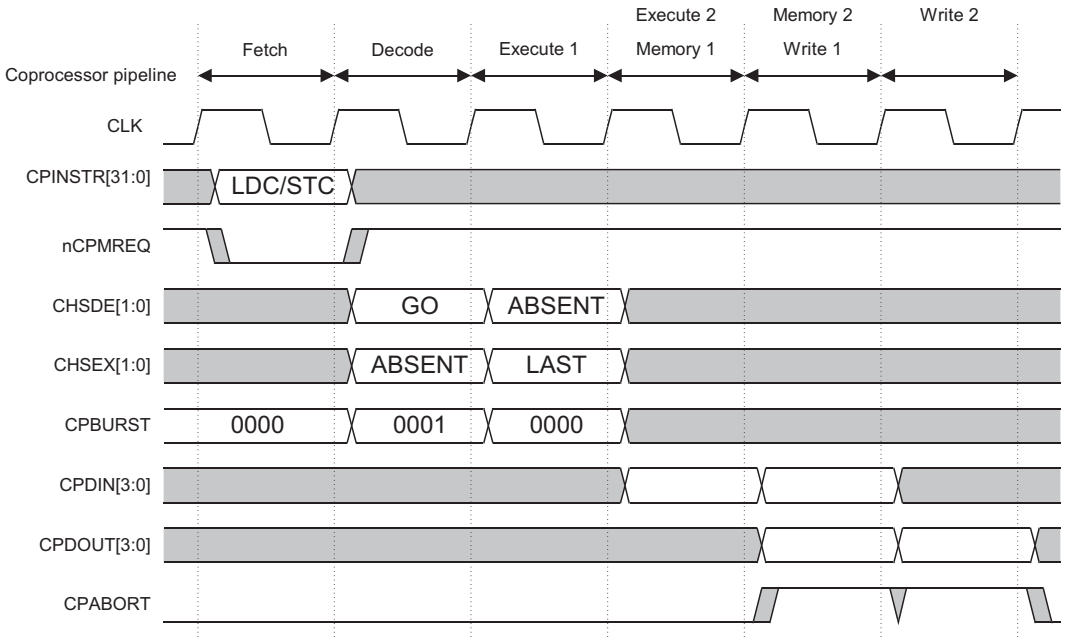


Figure 8-9 CPBURST and CPABORT timing

8.9 nCPINSTRVALID

The **nCPINSTRVALID** signal indicates if the instruction currently on the **CPINSTR** bus is valid, and should be decoded by the coprocessor. If **nCPINSTRVALID** is 1, then the instruction should not be decoded by the coprocessor and an ABSENT response should be made for all corresponding Decode cycles for this instruction.

nCPINSTRVALID is the equivalent of the **CPTBIT** signal in the ARM946E-S and ARM966E-S processors.

8.10 Connecting multiple external coprocessors

If multiple coprocessors are connected to the ARM926EJ-S processor, then outputs of the various coprocessors must be combined to form a single set of coprocessor inputs. The coprocessor handshake signals are combined together by ANDing the top bit and ORing the bottom bit. This enables a coprocessor to produce a fixed response of `b10` (Absent), when it is inactive. The other external coprocessor inputs, **CPDIN** and **CPBURST**, are combined by ORing. This is shown in Figure 8-10.

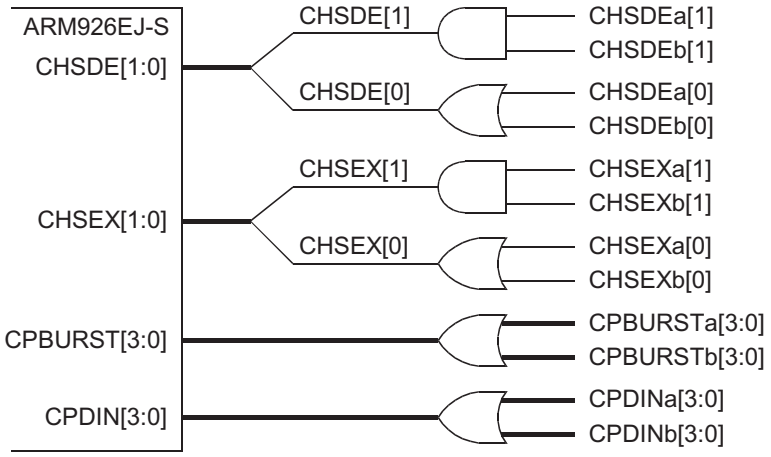


Figure 8-10 Arrangement for connecting two coprocessors

The OR arrangement for **CPBURST** and **CPDIN** means that coprocessors must drive zero values onto their **CPBURST** and **CPDIN** outputs when they are inactive, or do not own the corresponding coprocessor pipeline stage associated with these signals.

Chapter 9
Instruction Memory Barrier

This chapter describes the ARM926EJ-S *Instruction Memory Barrier* (IMB) operation. It contains the following sections:

9.1 About the instruction memory barrier operation

Whenever code is treated as data, for example self-modifying code, or loading code into memory, then a sequence of instructions called an *Instruction Memory Barrier* (IMB) operation must be used to ensure consistency between the data and instruction streams processed by the ARM926EJ-S processor.

Usually the instruction and data streams are considered to be completely independent by the ARM926EJ-S processor memory system, and any changes in the data side are not automatically reflected in the instruction side. For example if code is modified in main memory then the ICache might contain stale entries. To remove these stale entries part or all of the ICache must be invalidated.

9.2 IMB operation

To ensure consistency between data and instruction sides, you must take the following steps:

1. *Clean the DCache*
2. *Drain the write buffer*
3. *Synchronize data and instruction streams in level two AHB subsystems*
4. *Invalidate the ICache* on page 9-4
5. *Flush the prefetch buffer* on page 9-4.

9.2.1 Clean the DCache

If the cache contains cache lines corresponding to write-back regions of memory, then it might contain dirty entries. These entries must be cleaned to make external memory consistent with the DCache. If only a small part of the cache has to be cleaned, then this can be done by using a sequence of clean DCache single entry instructions, or if the entire cache has to be cleaned, then this can be done efficiently using the test and clean instruction. See *Cache Operations Register c7* on page 2-20 for details of cache maintenance operations.

9.2.2 Drain the write buffer

Executing a drain write buffer instruction causes the ARM9EJ-S core to wait until outstanding buffered writes have completed on the AHB interface. This includes writes that occur as a result of data being written back to main memory because of clean operations, and data for store instructions.

9.2.3 Synchronize data and instruction streams in level two AHB subsystems

The level two AHB subsystem might also require explicit synchronization between data and instruction sides. It is possible for the data and instruction AHB masters to be attached to different AHB subsystems. Even if both masters are present on the same bus, some form of separate ICache might exist for performance reasons, and this has to be invalidated to ensure consistency.

The process of synchronizing instructions and data in level two memory must be invoked using some form of fully blocking operation. This is to ensure that the end of the operation can be determined using software. It is recommended that either a nonbuffered store (`STR`) or a noncached load (`LDR`) is used to trigger external synchronization.

9.2.4 Invalidate the ICache

The ICache must be invalidated to remove any stale copies of instructions that are no longer valid. If the ICache is not being used, or the modified regions are not in cachable areas of memory, then this might not be required.

9.2.5 Flush the prefetch buffer

To ensure consistency, the prefetch buffer should be flushed before self-modifying code is executed. See *Self modifying code* on page 7-2.

9.3 Example IMB sequences

The following sequence corresponds to steps 1-4 in *IMB operation* on page 9-3:

```
clean_loop
    MRC p15, 0, r15, c7, c10, 3 ; clean entire dcache using test and clean
    BNE clean_loop

    MCR p15, 0, r0, c7, c10, 4  ; drain write buffer
    STR rx,[ry]                 ; nonbuffered store to signal L2 world to
                                ; synchronize
    MCR p15, 0, r0, c7, c5, 0   ; invalidate icache
```

The following sequence illustrates an IMB sequence used after modifying a single instruction (for example, setting a software breakpoint), with no external synchronization required:

```
STR rx,[ry]                  ; store that modifies instruction at address ry
MCR p15, 0, ry, c7, c10, 1   ; clean dcache single entry (MVA)
MCR p15, 0, r0, c7, c10, 4   ; drain write buffer
MCR p15, 0, ry, c7, c5,  1   ; invalidate icache single entry (MVA)
```


Chapter 10
Embedded Trace Macrocell Support

This chapter describes the *Embedded Trace Macrocell* (ETM) support for the ARM926EJ-S processor. It contains the following section:

- *About Embedded Trace Macrocell support* on page 10-2.

10.1 About Embedded Trace Macrocell support

To support real-time trace, the ARM926EJ-S processor provides an interface to enable connection of an *Embedded Trace Macrocell* (ETM). For more information on the ETM, see the *ETM9 Technical Reference Manual*.

The ETM consists of two parts:

Trace port A trace protocol has been developed to provide a real-time trace capability for processor cores that are deeply embedded in larger ASIC designs. Because the ASIC normally includes significant amounts of on-chip memory, it is not possible to determine how the processor core is operating by only observing the pins of the ASIC. A trace port is required to understand the operation of the processor.

Triggering facilities

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

The ETM is used to compress the trace information and export it through a narrow trace port. An external *Trace Port Analyzer* (TPA) is used to capture the trace information.

The ARM926EJ-S ETM interface exports the required signals for the ETM to perform trace. The interface is enabled and disabled by the **ETMEN** input signal. Where an ETM module is not required, the **ETMEN** input can be tied LOW to disable the trace outputs and save power.

10.1.1 FIFOFULL

Whenever the ETM FIFO fills up, the ETM asserts its **FIFOFULL** signal. To prevent loss in trace coverage, the ARM926EJ-S processor stalls until **FIFOFULL** is deasserted.

The ARM926EJ-S processor only stalls on instruction boundaries, to allow any AHB transfers to complete. Programming of the ETM FIFO watermark must take this into consideration. If the current instruction is either an `LDM` or an `STM`, then the FIFO might have to accept up to 16 words after **FIFOFULL** has been asserted.

Interrupts (FIQ or IRQ) prevent the ARM926EJ-S processor from stalling when **FIFOFULL** is asserted, unless they are masked. See *Test and Debug Register c15* on page 2-36 for details of how interrupts can be masked during trace.

Note

Stalling the core with **FIFOFULL** affects real-time operating performance. If connected, an ETM must be disabled during normal ARM926EJ-S processor operation to prevent **FIFOFULL** adversely affecting the ARM926EJ-S processor performance.

Chapter 11
Debug Support

This chapter describes the debug support for the ARM926EJ-S processor. It contains the following section:

- *About debug support* on page 11-2.

11.1 About debug support

Debug support is implemented by using the ARM9EJ-S core embedded within the ARM926EJ-S processor. Full details of the debug support provided by the ARM9EJ-S core are described in the *ARM9EJ-S Technical Reference Manual*.

Debug support for the ARM926EJ-S memory system is implemented by extending the debug facilities providing access to CP15 using an ARM9EJ-S external scan chain (scan chain 15). This scan chain is external to the ARM9EJ-S core but internal to the ARM926EJ-S processor.

11.1.1 Debug clocks

The system and test clocks must be synchronized externally to the ARM926EJ-S macrocell. To synchronize off-chip debug clocking with the ARM926EJ-S macrocell requires a three-state synchronizer. This is described in the debug chapter of the *ARM9EJ-S Technical Reference Manual*.

11.1.2 Scan chain 15

Scan chain 15 enables access to the CP15 registers. Scan chain 15 is 48 bits long. Table 11-1 shows the bit assignments for scan chain 15.

Table 11-1 Scan chain 15 format

Bits	Function
[47]	Write, not read (W/R)
[46:33]	Register address
[32]	Initiate access/access complete When written: 1 = initiate new access 0 = NOP When read: 1 = access complete 0 = access incomplete
[31:0]	Data value

With scan chain 15 selected, **TDI** is connected to bit 47 and **TDO** is connected to bit 0.

To perform an access using scan chain 15, you must:

1. During the SHIFT-DR state of the TAP state machine, shift in the read/write bit, register address, and register data value for writing, with bit 32 set to 1. For read operations the data value field does not have to be written.
2. Move through UPDATE-DR. The operation specified by the register address and write not read bits does not start.
3. Return to SHIFT-DR and perform a shift operation so that bits 32, and [31:0] are read, and a NOP instruction (bit 32 = 0) is shifted in.
4. Move through UPDATE-DR. No operation is performed because bit 32 is 0.
5. Check the access complete value that is shifted out. If it is 1, the operation has completed and bits [31:0] contain valid data for reads. If it is 0, the access has not completed and you must go back to step 3.

Note

If Multi-ICE is used, then this has the restriction that a maximum of 40 bits of any scan chain can be written at a time. Because scan chain 15 is 48 bits long, CP15 register writes require two operations to write all the required bits, and initiate the access. This can be done by first writing bits [31:0] with the required data value, and bit 32 to 0. This has the effect of presetting the data value field for the next operation. The second operation sets bits [47:33] to the required values, and bit 32 to 1 to initiate the access. This relies on the specific behavior of scan chain 15, which enables data to be recirculated if a value is scanned in with bit 32 set to 0, and there is no pending access. In this case the transition through UPDATE-DR does not modify the contents of the scan chain, and the value written in can safely be read back out in a subsequent CAPTURE-DR, SHIFT-DR sequence.

The mapping of scan chain 15 to CP15 registers is done in the same way as a CP15 MRC/MCR operation. Bits [46:33] of the scan chain are mapped onto Opcode_1, Opcode_2, CRn, and CRm.

The mapping of the register address field to the CP15 registers is shown in Table 11-2.

Table 11-2 Scan chain 15 mapping to CP15 registers

MRC/MCR instruction field	Scan chain 15 mapping
Opcode_1	[46:44]
Opcode_2	[43:41]
CRn	[40:37]
CRm	[36:33]

Writes to either the cache operations register (CRn = c7) or the TLB operations register (CRn = c8), which require a form of address to select an entry to be manipulated, use the data value part of the scan chain to provide the address information. The format of the address field is identical to that used for the value of Rd, for the equivalent MCR instruction.

Memory system debug operations (CRn = c15), which require an address to be used to select an entry, use the value held in the debug address register (see *Debug and Test Address Register* on page B-4). The format of the address field is identical to that used for the value of Rd, for the equivalent MCR instruction.

If an invalid instruction is scanned into scan chain 15, it is translated into a read of the ID register. This means that you can check the output data for ID register reads to indicate that an invalid instruction has been scanned in.

Chapter 12
Power Management

This chapter describes the power management facilities provided by the ARM926EJ-S processor. It contains the following section:

- *About power management* on page 12-2.

12.1 About power management

The power management facilities provided by the ARM926EJ-S processor are:

- *Dynamic power management (wait for interrupt mode)*
- *Static power management (leakage control)* on page 12-3.

12.1.1 Dynamic power management (wait for interrupt mode)

The ARM926EJ-S processor can be put into a low-power state by the wait for interrupt instruction:

```
MCR p15,0,<Rd>,c7,c0,4
```

This instruction switches the ARM926EJ-S processor into a low-power state until either an interrupt (IRQ or FIQ) or a debug request occurs. The debug request can either be an external debug request **EDBGRQ** or a debug request made by the debugger by writing to the DBGRQ bit of the ARM9EJ-S debug control register using scan chain 2.

In wait for interrupt mode, all internal ARM926EJ-S clocks can be stopped. The switch into the low-power state is delayed until all write buffers have been drained, and the ARM926EJ-S memory system is in a quiescent state.

The switch into low-power state is indicated by the assertion of the **STANDBYWFI** signal. If **STANDBYWFI** is asserted then it is guaranteed that all of ARM926EJ-S external interfaces (AHB, TCM, and external coprocessor) are in an idle state. The **STANDBYWFI** signal is intended to be used to shut down clocks to other parts of the system, such as external coprocessors, that do not have to be clocked if the ARM926EJ-S processor is idle.

The **STANDBYWFI** signal is deasserted in the second cycle following an interrupt or a debug request. It is guaranteed that no form of access on any external interface is started until the cycle after **STANDBYWFI** is deasserted. Figure 12-1 shows the deassertion of the **STANDBYWFI** signal after an IRQ interrupt.

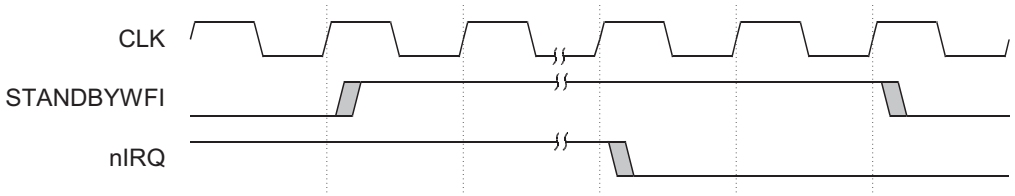


Figure 12-1 Deassertion of STANDBYWFI after an IRQ interrupt

When the ARM926EJ-S has entered a low-power state, all of the main internal clocks are stopped, including the clock for the ARM9EJ-S core. However, the ARM9EJ-S is active if **DBGTCKEN** is asserted. This enables values to be written in the ARM9EJ-S debug control register so that a debugger can force an exit from wait for interrupt mode. This means that you can safely stop the ARM926EJ-S **CLK** if **STANDBYWFI** is HIGH and **DBGTCKEN** is LOW.

Figure 12-2 shows the recommended logic for stopping the main ARM926EJ-S clock during wait for interrupt.

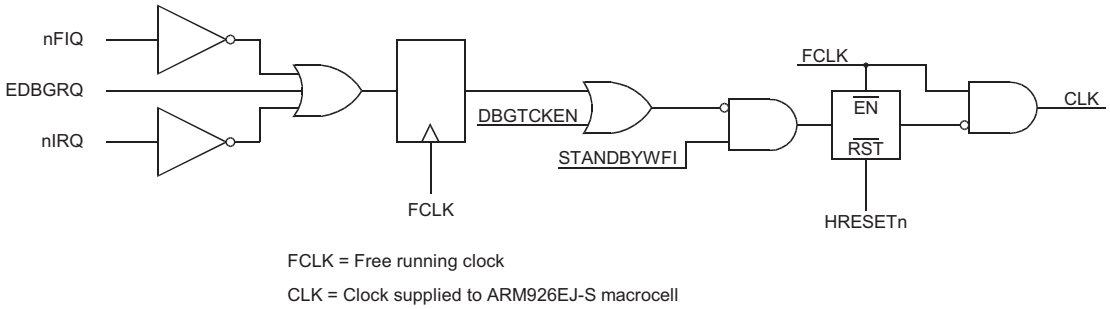


Figure 12-2 Logic for stopping ARM926EJ-S clock during wait for interrupt

The nature of the **nFIQ**, **nIRQ**, and **EDBGRQ** signals enables them to be registered prior to being used in the gating logic. **DBGTCKEN** must be used combinationally to maintain the relationship between the ARM926EJ-S JTAG logic and the **RTCK** signal used by the debugger. See the *ARM9EJ-S Technical Reference Manual* for details of how **DBGTCKEN** is generated and used.

12.1.2 Static power management (leakage control)

The ARM926EJ-S design is partitioned so that the SRAM blocks that are used for the caches and the MMU can be powered down under certain conditions.

Cache RAMs

The RAMs for either of the caches can be safely powered down if the respective cache has been disabled (using CP15 control register c1) and it contains no valid entries. While a cache is disabled, only explicit CP15 operations can cause the cache RAMs to be accessed (c7 cache maintenance operations). These instructions must not be executed while any of the cache RAMs are powered down. If any of the RAMs for a cache have been powered down, then they must be powered up prior to re-enabling the relevant cache.

MMU RAMs

The RAM used to implement the MMU can be safely powered down if the MMU has been disabled (using CP15 control register c1) and it contains no valid entries.While the MMU is disabled, only explicit CP15 operations can cause the MMU RAM to be accessed (c8 TLB maintenance operations, and c15 MMU test/debug operations). These instructions must not be executed while the MMU RAM is powered down.The MMU RAM must be powered up prior to re-enabling the MMU.

Appendix A
Signal Descriptions

This appendix describes the ARM926EJ-S processor input and output signals. It contains the following sections:

A.1 Signal properties and requirements

To ensure ease of integration of the ARM926EJ-S processor into embedded applications, and to simplify synthesis flow, the following design techniques have been used:

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

These techniques simplify the definition of the top-level ARM926EJ-S processor signals because all outputs change from the rising edge and all inputs are sampled with the rising edge of the clock. In addition, all signals are either input or output only. Bidirectional signals are not used.

Note

You must use external logic to synchronize asynchronous signals (for example interrupt sources) before applying them to the ARM926EJ-S processor.

A.2 AHB related signals

Table A-1 describes the ARM926EJ-S processor AHB related signals.

Table A-1 AHB related signals

Signal name	Direction	Description
DHADDR[31:0]	Output	AHB address (data).
DHBL[3:0]	Output	Byte lane indicator for current transfer.
DHBURST[2:0]	Output	AHB burst size (data).
DHBUSREQ	Output	AHB bus request (data).
DHCLKEN	Input	Signifies the rising edge of **HCLK** for the data AHB. If **CLK** and **HCLK** are the same frequency, **DHCLKEN** must be tied HIGH.
DHGRANT	Input	AHB bus grant signal (data).
DHLOCK	Output	AHB bus lock signal (data).
DHPROT[3:0]	Output	AHB bus access information (data).
DHRDATA[31:0]	Input	AHB read data (data).
DHREADY	Input	AHB transfer complete signal (data).
DHRESP[1:0]	Input	AHB transfer response (data).
DHSIZE[2:0]	Output	AHB transfer size (data), indicating byte, halfword, or word. **DHSIZE[2]** is tied LOW.
DHTRANS[1:0]	Output	AHB transfer type (data).
DHWDATA[31:0]	Output	AHB write data (data).
DHWRITE	Output	AHB transfer direction (data).
HRESETn	Input	AHB reset signal.
IHADDR[31:0]	Output	AHB address (instruction).
IHBURST[2:0]	Output	AHB burst size. (instruction).
IHBUSREQ	Output	AHB bus request (instruction).
IHCLKEN	Input	Signifies the rising edge of **HCLK** for the data AHB. If **CLK** and **HCLK** are the same frequency, **IHCLKEN** must be tied HIGH.

Table A-1 AHB related signals (continued)

Signal name	Direction	Description
IHGRANT	Input	AHB bus grant signal (instruction).
IHLOCK	Output	AHB bus lock signal (instruction).
IHPROT[3:0]	Output	AHB bus access information (instruction).
IHREADY	Input	AHB transfer complete signal (instruction).
IHRDATA[31:0]	Input	AHB read data (instruction).
IHRESP[1:0]	Input	AHB transfer response (instruction).
IHSIZE[2:0]	Output	AHB transfer size (instruction), indicating byte, halfword, or word. **IHSIZE[2]** is tied LOW.
IHTRANS[1:0]	Output	AHB transfer type (instruction).
IHWRITE	Output	AHB transfer direction (instruction).

A.3 Coprocessor interface signals

Table A-2 describes the ARM926EJ-S processor coprocessor interface signals.

Table A-2 Coprocessor interface signals

Name	Direction	Description
CPABORT	Output	Indicates STC/LDC operation aborted. Asserted in WB stage of coprocessor pipeline.
CPBURST[3:0]	Output	Indicates number of words to be transferred for LDC/STC operation. If no external coprocessors are attached, this must be tied to b0000.
CPCLKEN Coprocessor clock enable	Output	Coprocessor clock enable. When HIGH on the rising edge of **CLK** the pipeline follower logic can advance.
CPDIN[31:0] Coprocessor write data	Input	The coprocessor data bus for transferring data from the coprocessor.
CPDOUT[31:0] Coprocessor read data	Output	The coprocessor data bus for transferring data to the coprocessor.
CPEN Coprocessor enable	Input	When LOW disables the external coprocessor interface. If **CPEN** is LOW then **CHSDE** and **CHSEX** must both be driven to b10 (ABSENT response).
CPINSTR[31:0] Coprocessor instruction data	Output	The coprocessor instruction bus that instructions are transferred over to the pipeline follower in 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.
CHSDE[1:0] Coprocessor handshake decode	Output	The handshake signals from the Decode stage of the coprocessor pipeline follower. Indicates ABSENT (b10), WAIT (b00), GO (b01), or LAST (b11). If no external coprocessors are attached this must be tied to b10 (ABSENT response).

Table A-2 Coprocessor interface signals (continued)

Name	Direction	Description
CHSEX[1:0] Coprocessor handshake execute	Input	The handshake signals from the Execute stage of the coprocessors pipeline follower. Indicates ABSENT (10), WAIT (00), GO (01), or LAST (11). If no external coprocessors are attached these must be tied to b10 (ABSENT response).
nCPINSTRVALID Coprocessor valid instruction	Output	Valid instruction indicator for **CPINSTR** (replaces CPTBIT).
nCPMREQ Not coprocessor instruction request	Output	If this signal is 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 the coprocessor interface is in a nonprivileged state. When HIGH the coprocessor interface is in a privileged state.

A.4 Debug signals

Table A-3 describes the ARM926EJ-S processor debug signals.

Table A-3 Debug signals

Name	Direction	Description
COMMRX Communications channel receive	Output	When HIGH, this signal denotes that the comms channel receive buffer contains valid data waiting to be read.
COMMTX Communications channel transmit	Output	When HIGH, this signal 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 ARM926EJ-S processor 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	Inputs to the EmbeddedICE-RT logic that enable breakpoints or 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 ARM926EJ-S processor to enter debug state if that instruction reaches the Execute stage of the processor pipeline.
DBGINSTREXEC Instruction executed	Output	Indicates that the instruction in the Execute stage of the processor pipeline has been executed.

Table A-3 Debug signals (continued)

Name	Direction	Description
DBGRNG[1:0] EmbeddedICE-RT range out	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.

A.5 JTAG signals

Table A-4 describes the ARM926EJ-S processor JTAG signals.

Table A-4 JTAG signals

Name	Direction	Description
DBGIR[3:0] TAP controller instruction register	Output	These four bits reflect the current instruction loaded into the TAP controller instruction register. These bits change when the TAP controller is in the UPDATE-IR state.
DBGnTRST Not test reset	Input	This is the active LOW reset signal for the EmbeddedICE-RT internal state. This signal is a level-sensitive asynchronous reset input.
DBGnTDOEN Not **DBGTDO** enable	Output	When LOW, indicates that the serial data is being driven out of the **DBGTDO** output. Normally used as an output enable for a **DBGTDO** pin in a packaged part.
DBGSCREG[4:0]	Output	These five bits reflect the ID number of the scan chain currently selected by the TAP controller. These bits change when the TAP controller is in the UPDATE-DR state.
DBGSDIN 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 test clock enable.
DBGTDI	Input	Test data input for debug logic.
DBGTDO	Output	Test data output from debug logic.
DBGTMS	Input	Test mode select for TAP controller.

A.6 Miscellaneous signals

Table A-5 describes the miscellaneous signals on the ARM926EJ-S processor.

Table A-5 Miscellaneous signals

Name	Direction	Description
BIGENDINIT	Input	Determines the setting of the B bit in CP15 c1 after a system reset. When HIGH the reset state of the B bit is 1 (big-endian). When LOW the reset state of the B bit is 0 (little-endian).
CLK	Input	This clock times all operations of the ARM926EJ-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. Through the use of the **DHCLKEN** and **IHCLKEN** signals, this clock also times AHB operations. Through the use of the **DBGTCKEN** signal, this clock also controls JTAG and debug operations.
CFGBIGEND ARM9EJ-S core endianness configuration	Output	This signal reflects the setting of the B bit in CP15 c1. When HIGH, the processor treats bytes in memory as being in big-endian format. When LOW, memory is treated as little-endian.
EXTEST	Input	EXTEST mode test signal. This signal must be LOW during normal operation.
INTEST	Input	INTEST mode test signal. This signal must be LOW during normal operation.
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**.
SCANENABLE	Input	Scan enable test signal. This signal must be LOW during normal operation.
STANDBYWFI	Output	When HIGH indicates that the ARM926EJ-S processor is in wait for interrupt mode.

Table A-5 Miscellaneous signals (continued)

Name	Direction	Description
TAPID[31:0]	Input	This is the ARM926EJ-S device identification (ID) code test data register, accessible from the scan chains. It must be tied to 0x07926F0F for an ARM926EJ-S processor when the device is instantiated.
TESTMODE	Input	Test mode test signal. This signal must be LOW during normal operation.
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`.

A.7 ETM interface signals

Table A-6 describes the ARM926EJ-S processor ETM interface signals.

Table A-6 ETM interface signals

Name	Direction	Description
ETMBIGEND	Output	ETM big-endian configuration indication.
ETMCHSD[1:0]	Output	ETM coprocessor handshake decode signals.
ETMCHSE[1:0]	Output	ETM coprocessor handshake execute signals.
ETMDA[31:0]	Output	ETM data address.
ETMDABORT	Output	ETM data abort.
ETMDBGACK	Output	ETM debug mode indication.
ETMDMAS[1:0]	Output	ETM data size indication.
ETMDMORE	Output	ETM more sequential data indication.
ETMDnMREQ	Output	ETM data memory request.
ETMDnRW	Output	ETM data not read/write.
ETMDSEQ	Output	ETM sequential data indication.
ETMEN	Input	Synchronous ETM interface enable. This signal must be tied LOW if an ETM is not used.
ETMHIVECS	Output	ETM exception vectors configuration.
ETMIA[31:0]	Output	ETM instruction address.
ETMIABORT	Output	ETM instruction abort.
ETMID15TO11[15:11]	Output	ETM instruction data field bits [15:11].
ETMID31TO25[31:25]	Output	ETM instruction data field bits [31:25].
ETMIJBIT	Output	ETM Jazelle state indication.
ETMInMREQ	Output	ETM instruction memory request.
ETMINSTREXEC	Output	ETM instruction execute indication.
ETMINSTRVALID	Output	ETM instruction valid indication.
ETMISEQ	Output	ETM sequential instruction access.

Table A-6 ETM interface signals (continued)

Name	Direction	Description
ETMITBIT	Output	ETM Thumb state indication.
ETMLATECANCEL	Output	ETM coprocessor late cancel indication.
ETMnWAIT	Output	ETM clock stall signal.
ETMPASS	Output	ETM coprocessor instruction execute indication.
ETMPROCID[31:0]	Output	ETM process identifier.
ETMPROCIDWR	Output	**ETMPROCID** write strobe.
ETMRDATA[31:0]	Output	ETM read data.
ETMRNGOUT[1:0]	Output	ETM watchpoint register match indication.
ETMWDATA[31:0]	Output	ETM write data.
ETMZIFIRST	Output	Indicates the current Decode cycle is the first being traced for the current Java instruction.
ETMZILAST	Output	Indicates the current Decode cycle is the last being traced for the current Java instruction.
FIFOFULL	Input	ETM FIFO full. This signal must be tied LOW if an ETM is not used.

A.8 TCM interface signals

Table A-7 describes the ARM926EJ-S TCM interface signals.

Table A-7 TCM interface signals

Signal	Direction	Function
DRADDR[17:0]	Output	Data TCM address. This is the word address for the access. Valid during request cycles.
DRCS	Output	Chip select. Indicates if an access will take place in the following cycle. Not valid during wait cycles.
DRDMAADDR[17:0]	Input	Direct memory access address for DTCM memory. If **DRDMAEN** is set to 1, then the value of **DRDMAADDR** is routed directly through to **DRADDR**.
DRDMAEN	Input	DMA access cycle. If asserted, **DRADDR** is directly sourced from **DRDMAADDR**, and **DRCS** is the result of logically ORing **DRDMACS** with the chip select value for the current TCM access.
DRDMACS	Input	Direct memory access chip-select for DTCM.
DRIDLE	Output	Data TCM interface idle: 0 = TCM access 1 = no access will take place in the current cycle or TCM disabled. Not valid for DMA accesses.
DRnRW	Output	Data TCM read not write: 0 = read 1 = write. Indicates if the access is a read or write. Valid during request cycles.
DRRD[31:0]	Input	Data TCM read data. Valid during non-waited data cycles.
DRSEQ	Output	Request sequential. Valid during request cycles, asserted during wait cycles. Indicates that the address in the current cycle is sequential to the address used during the previous request cycle.

Table A-7 TCM interface signals (continued)

Signal	Direction	Function
DRSIZE[3:0]	Input	Data TCM size. Static configuration input that specifies the physical size of TCM memories attached. 0000 = absent 0011 = 4KB 0100 = 8KB … 1010 = 512KB 1011 = 1MB Values 0001, 0010, and 1100 to 1111 are reserved.
DRWAIT	Input	Data TCM wait state input. If HIGH, the DTCM cannot service the request in that cycle. Valid in request cycle and subsequent wait cycles. Ignored if not a request or wait cycle.
DRWBL[3:0]	Output	Data TCM write data byte lane indicator. Valid during request cycles. For reads, set to b0000 For writes indicates which byte(s) are to be written, depending on the address and the size of the access (word, halfword, or byte). Bits of **DRWBL** are set only when a write is taking place, so when **DnRW** is unset all the bits of **DRWBL** are also unset.
DRWD[31:0]	Output	Data TCM write data. Valid during request cycles when **DRnRW** is 0. Valid during waited write cycles.
INITRAM	Input	Enables instruction TCM at system reset. Enables booting from the instruction TCM if **VINITHI** is LOW.
IRADDR[17:0]	Output	Instruction TCM address. This is the word address for the access. Valid during request cycles.
IRCS	Output	Chip select. Indicates if an access will take place in the following cycle. Not valid during wait cycles.

Table A-7 TCM interface signals (continued)

Signal	Direction	Function
IRDMAADR[17:0]	Input	DMA access cycle. If asserted, **IRADDR** is directly sourced from **IRDMAADDR**, and **IRCS** is the result of logically ORing **IRDMACS** with the chip select value for the current TCM access.
IRDMAEN	Input	Enables direct memory access to the ITCM memory using the **IRDMAADDR** and **IRDMACS** inputs.
IRDMACS	Input	Direct memory access chip-select for ITCM.
IRIDLE	Output	Instruction TCM interface idle: 0 = TCM access 1 = no access will take place in the current cycle or TCM disabled. Not valid for DMA accesses.
IRnRW	Output	Instruction TCM read not write: 0 = read 1 = write. Indicates if the access is a read or write. Valid during request cycles.
IRRD[31:0]	Input	Instruction TCM read data. Valid during non-waited data cycles.
IRSEQ	Output	Request sequential. Valid during request cycles, asserted during wait cycles. Indicates that the address in the current cycle is sequential to the address used during the previous request cycle. **IRSEQ** is not valid following ITCM DMA accesses.
IRSIZE[3:0]	Input	Instruction TCM size. Static configuration input that specifies the physical size of TCM memories attached. 0000 = absent 0011 = 4KB 0100 = 8KB … 1010 = 512KB 1011 = 1MB Values 0001, 0010, and 1100 to 1111 are reserved.

Table A-7 TCM interface signals (continued)

Signal	Direction	Function
IRWAIT	Input	Instruction TCM wait state input. If HIGH, the ITCM cannot service the request in that cycle. Valid in request cycle and subsequent wait cycles. Ignored if not a request or wait cycle.
IRWBL[3:0]	Output	Instruction TCM write data byte lane indicator. Valid during request cycles. For reads, set to b0000 For writes indicates which byte(s) are to be written, depending on the address and the size of the access (word, halfword, or byte). Bits of **IRWBL** are set only when a write is taking place, so when **IRnRW** is unset all the bits of **IRWBL** are also unset.
IRWD[31:0]	Output	Instruction TCM write data. Valid during request cycles when **IRnRW** is 0. Valid during waited write cycles.

Appendix B
CP15 Test and Debug Registers

This appendix describes the ARM926EJ-S CP15 Test and Debug Registers. It contains the following section:

- *About the Test and Debug Registers* on page B-2.

B.1 About the Test and Debug Registers

The ARM926EJ-S Test and Debug Registers, CP15 c15, provide additional device-specific test operations. You can use the registers to access and control the following:

- *Debug Override Register*
- *Debug and Test Address Register* on page B-4
- *Trace Control Register* on page B-5
- *MMU test operations* on page B-5
- *Cache Debug Control Register* on page B-12
- *MMU Debug Control Register* on page B-13
- *Memory Region Remap Register* on page B-15.

You must only use these operations for test. The *ARM Architecture Reference Manual* describes this register as implementation-defined.

The format of the CP15 test and debug operations is:

```
MCR/MRC  p15, <Opcode_1>, <Rd>, c15, <CRm>, <Opcode_2>
```

The MRC and MCR bit pattern is shown in Figure B-1.

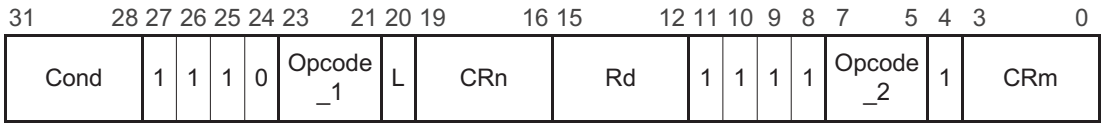


Figure B-1 CP15 MRC and MCR bit pattern

The L bit distinguishes between an MCR (L = 1) and an MRC (L = 0).

B.1.1 Debug Override Register

You can use the Debug Override Register to modify the behavior of the ARM926EJ-S core from the default behavior.

The function of each ARM926EJ-S Debug Override Register bit is shown in Table B-1 on page B-3.

The Debug Override Register can be accessed by using the following instructions:

```
MRC{cond} p15,0,<Rd>,c15,c0,0 ; Read Debug Override Register
MCR{cond} p15,0,<Rd>,c15,c0,0 ; Write Debug Override Register
```

The reset state of the Debug Override Register is `0x0`.

Table B-1 Debug Override Register

Bits	Function or name	Description
[31:20]	Reserved	Read = Unpredictable Write = Should Be Zero
[19]	Test and clean all	0 = Default behavior for test and clean instructions 1 = Modifies the behavior of test and clean, and test, clean, and invalidate instructions so that they act on the complete cache
[18]	Abort data TLB miss	0 = Do not abort DTLB miss 1 = Abort DTLB miss
[17]	Abort instruction TLB miss	0 = Do not abort ITLB miss 1 = Abort ITLB miss
[16]	Disable NC instruction prefetching	0 = Enable prefetching 1 = Disable prefetching
[15]	Disable block-level clock gating	0 = Enable block-level clock gating 1 = Disable block-level clock gating
[14]	Disable NCB stores (force NCNB)	0 = Enable NCB stores 1 = Disable NCB stores (force NCNB)
[13]	MMU disabled, DCache enabled behavior	0 = If MMU disabled. level one access NCNB 1 = If MMU disabled and DCache enabled level one access WT
[12:0]	Reserved	Read = Unpredictable Write = Should Be Zero

Bit 13, MMU disabled, DCache enabled behavior

This bit changes the behavior when the MMU is disabled but the DCache is enabled. During normal operation, if the MMU is disabled, all data accesses are treated as being NCNB. If Bit 13 is set with the MMU disabled, and the DCache is enabled, all data accesses are treated as WT.

Note

This behavior can be overridden using the memory region register.

Bit 14, disable NCB stores (force NCNB)

You can use this bit to force all NCB stores to be treated as NCNB stores at level one. This bit overrides the settings in both the MMU page tables and the memory region remap register.

Bit 15, disable block-level clock gating

You can use this bit to disable block-level clock gating with the ARM926EJ-S processor. This bit does not affect the functionality of the ARM926EJ-S processor. It allows the benefits of block-level clock gating to be evaluated without the requirement to build two different implementations of the ARM926EJ-S macrocell, one with block-level clock gating, one without.

Bit 16, disable NC instruction prefetching

You can use this bit to disable speculative prefetching for instructions in noncachable areas of memory. The default behavior of ARM926EJ-S processor is to perform speculative sequential instruction fetches on the AHB interface. Disabling prefetching prevents any speculative noncachable instruction prefetches by the ARM926EJ-S memory system, and only instruction requests issued by the ARM9EJ-S core result in instruction fetches on the AHB interface.

Bits 17 & 18, abort instruction TLB miss

You can use the abort data TLB miss and abort instruction TLB miss bits to prevent page table walks occurring as the result of a TLB miss. When set, a TLB miss results in the access being aborted as if the access has resulted in a translation fault, and a value of `0000` being written into the status field of the appropriate FSR.

Bit 19, test and clean all

You can use the test-and-clean-all bit to modify the behavior of the test and clean, and test clean and invalidate instructions so that a single instruction can be used to clean or clean and invalidate the entire cache. This is only intended for use by a debugger, to provide an efficient way to clean the data cache using scan chain 15.

B.1.2 Debug and Test Address Register

This register defines the address used for debug and test operations, and for MMU test operations using the MMU Test Register.

You can access the Debug and Test Address Register using the following instructions:

```
MRC{cond} p15,0,<Rd>,c15,c1,0 ; Read Debug and Test Address Register
MCR{cond} p15,0,<Rd>,c15,c1,0 ; Write Debug and Test Address Register
```

B.1.3 Trace Control Register

You can access the Trace Control Register by using the following instructions:

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

You can use the Trace Control Register to determine under what conditions the ARM9EJ-S core is stalled when the **FIFOFULL** signal is asserted.

Usually, non-invasive real-time trace requires the presence of an **nFIQ** or **nIRQ** interrupt to prevent the ARM9EJ-S core being stalled by **FIFOFULL** being asserted.

The Trace Control Register enables you to modify this behavior, so that the presence of an interrupt does not prevent the ARM9EJ-S core being stalled if **FIFOFULL** is asserted.

Table B-2 shows the bit assignments for the Trace Control Register. Bits [2:1] of this register are reset to 0.

Table B-2 Trace Control Register bit assignments

Bits	Content
[31:3]	Reserved (Should Be Zero)
[2]	1 = FIQ interrupt does not prevent **FIFOFULL** from stalling the ARM9EJ-S core 0 = FIQ interrupt prevents **FIFOFULL** from stalling the ARM9EJ-S core
[1]	1 = IRQ interrupt does not prevent **FIFOFULL** from stalling the ARM9EJ-S core 0 = IRQ interrupt prevents **FIFOFULL** from stalling the ARM9EJ-S core
[0]	Reserved (Should Be Zero)

B.1.4 MMU test operations

The MMU test operations support accessing TLB structures in the MMU and are used in conjunction with the Debug and Test Address Register.

You can access the MMU test operations using the instructions in Table B-3.

Table B-3 MMU test operation instructions

Instruction	Operation
`MRC p15, 4/5, <Rd>, c15, c2, 0`	Read tag in main TLB entry
`MCR p15, 4/5, <Rd>, c15, c3, 0`	Write tag in main TLB entry

Table B-3 MMU test operation instructions (continued)

Instruction	Operation
`MRC p15, 4/5, <Rd>, c15, c4, 0`	Read PA and access permission data in main TLB entry
`MCR p15, 4/5, <Rd>, c15, c5, 0`	Write PA and access permission data data in main TLB entry
`MCR p15, 4/5, <Rd>, c15, c7, 0`	Transfer main TLB entry into RAM
`MRC P15, 4/5, <Rd>, c15, c2, 1`	Read tag in lockdown TLB entry
`MCR P15, 4/5, <Rd>, c15, c3, 1`	Write tag in lockdown TLB entry
`MRC P15, 4/5, <Rd>, c15, c4, 1`	Read PA and access permission data in lockdown TLB entry
`MCR P15, 4/5, <Rd>, c15, c5, 1`	Write PA and access permission data in lockdown TLB entry
`MCR P15, 4/5, <Rd>, c15, c7, 1`	Transfer lockdown TLB entry into RAM

Inserting or reading entries in the main TLB

Use this procedure to access entries in the main TLB:

1. Use the following Debug and Test Address Register instruction to access a main TLB entry:

   ```
   MCR p15, 0, <Rd>, c15, c1, 0 ; select TLB entry
   ```

   The Rd register selects the main TLB entry as Figure B-2 shows.

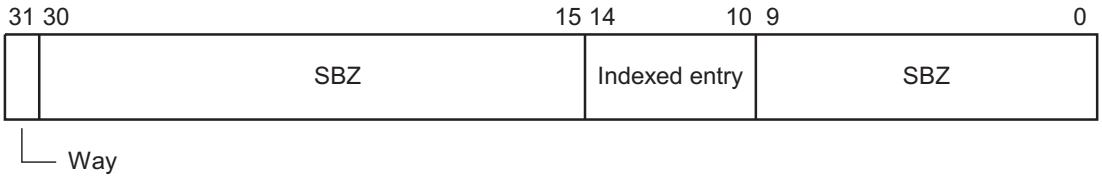


Figure B-2 Rd format for selecting main TLB entry

Table B-4 describes the Rd register entry-select bit fields.

Table B-4 Encoding of the main TLB entry-select bit fields

Bit	Name	Definition
[31]	Way	Way select: 1 = way 1 0 = way 0.

Table B-4 Encoding of the main TLB entry-select bit fields

Bit	Name	Definition
[30:15]	-	Should Be Zero.
[14:10]	Indexed entry	Indexed entry in main TLB.
[9:0]	-	Should Be Zero.

2. Use the following MMU test operation instructions to access the MVA tag:

```
MRC p15, 4/5, <Rd>, c15, c2, 0 ; read tag in main TLB
MCR p15, 4/5, <Rd>, c15, c3, 0 ; write tag in main TLB
```

The Rd register contains the read or write data as Figure B-3 shows.

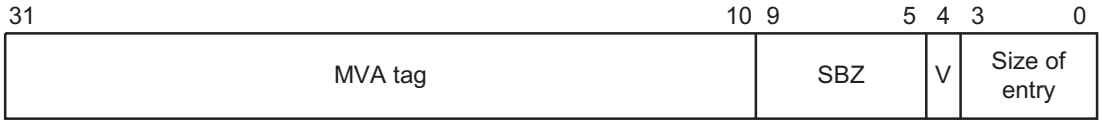


Figure B-3 Rd format for accessing MVA tag of main or lockdown TLB entry

Table B-5 describes the MVA tag access bit fields in the Rd register.

Table B-5 Encoding of the TLB MVA tag bit fields

Bit	Name	Definition
[31:10]	MVA tag	Modified virtual address.
[9:5]	-	Should Be Zero.
[4]	V	Valid bit.
[3:0]	Size of entry	Size of entry: b1011 = 1MB section b0111 = 64KB page b0101 = 16KB subpage of 64KB page b0011 = 4KB page b0001 = 1KB page or 1KB subpage of 4KB page.

3. Use the following MMU Test Register instructions to access the PA and access permission data:

```
MRC p15, 4/5, <Rd>, c15, c4, 0 ; read PA and access permission data
```

```
MCR p15, 4/5, <Rd>, c15, c5, 0 ; write PA and access permission data
```

The Rd register contains the read or write data as shown in Figure B-4.

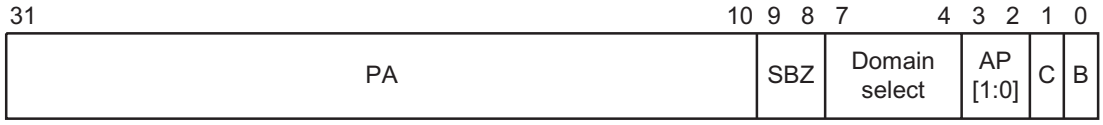


Figure B-4 Rd format for accessing PA and AP data of main or lockdown TLB entry

Table B-6 describes the PA and access permission bit fields in the Rd register.

Table B-6 Encoding of the TLB entry PA and AP bit fields

Bit	Name	Definition
[31:10]	PA	Physical address.
[9:8]	-	Should Be Zero.
[7:4]	Domain select	Domain select: b0000 = D0 b0001 = D1 . . . b1110 = D14 b1111 = D15.
[3:2]	AP	Access permission: b00 = No access. b01 = Privileged, read/write. User, no access. b10 = Privileged, read/write. User read-only. b11 = Privileged, read/write. User, read/write.
[1]	C	Cachable bit.
[0]	B	Bufferable bit.

4. Use the following instruction to complete a write to an entry:

   ```
   MCR p15, 4/5, Rd, c15, c7, 0 ; transfer main storage into RAM
   ```

To write an entry into the 2-way main TLB, the full sequence is therefore:

```
MCR p15, 4/5, <Rd>, c15, c3, 0 ; write tag main TLB storage reg
MCR p15, 4/5, <Rd>, c15, c5, 0 ; write PA/PROT main TLB storage reg
MCR p15, 4/5, <Rd>, c15, c7, 0 ; transfer main storage into RAM
```

To read an entry from the 2-way main TLB, the entry must first be written using the above instructions. The entry can then be read using the following instructions:

```
MRC p15, 4/5, <Rd>, c15, c2, 0 ; read tag main TLB
MRC p15, 4/5, <Rd>, c15, c4, 0 ; read PA/PROT main TLB
```

The data RAM attached to the main MMU is 112 bits wide. The mapping into the data RAM for main TLB writes for the TAG is shown below and would appear on **MMUxWD[111:0]** as shown in Table B-7.

Table B-7 Main TLB mapping to MMUxWD

Way	MMUxWD bits	Description
1	[111:90]	TAG[31:10]
	[89:86]	Size of entry
	[85:64]	PA[31:10]
	[63:60]	Domain select [3:0]
	[59:58]	AP[1:0]
	[57]	Cachable bit
	[56]	Bufferable bit
0	[55:34]	TAG[31:10]
	[33:30]	Size of entry
	[29:8]	PA[31:10]
	[7:4]	Domain select [3:0]
	[3:2]	AP[1:0]
	[1]	Cachable bit
	[0]	Bufferable bit

During writes, the data is replicated so that each way receives the same copy of the data. The exact way that is written and the exact index of the way is specified in the Test and Debug Address Register.

Figure B-5 on page B-10 shows what happens during a write to the data RAM attached to the main MMU.

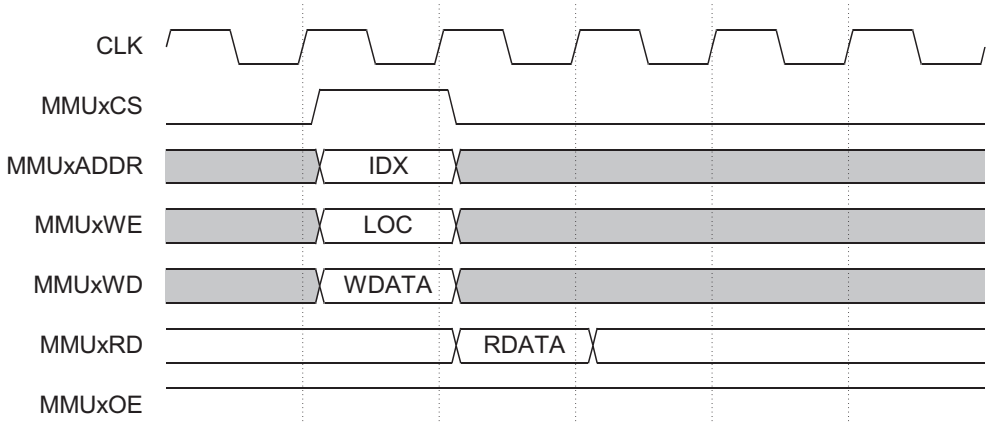


Figure B-5 Write to the data RAM

Note

On the rising clock edge when **MMUxCS**=1, the data on **MMUxWD** is written into the data RAM. The exact index is on **MMUxADDR** (as specified in the Test and Debug Address Register). The lanes written are controlled by the **MMUxWE[3:0]** pins. The mapping is as follows:

MMUxWE[0]: 0= read, 1= write **MMUxWD[29: 0]** into RAM
MMUxWE[1]: 0= read, 1= write **MMUxWD[55:30]** into RAM
MMUxWE[2]: 0= read, 1= write **MMUxWD[85:57]** into RAM
MMUxWE[3]: 0= read, 1= write **MMUxWD[111:86]** into RAM

In the case of the main MMU, the output enable **MMUxOE** is driven at all times. The **MMUxRD** data bus must be strongly driven at all times. The controller samples the data from the **MMUxRD** data bus when a read is being performed.

Inserting or reading entries in the lockdown TLB

Use this procedure to access entries in the lockdown TLB:

1. Use the following Debug and Test Address Register instruction to access a lockdown TLB entry:

   ```
   MCR p15, 0, <Rd>, c15, c1, 0
   ```

   The Rd register selects the lockdown TLB entry as shown in Figure B-6 on page B-11.

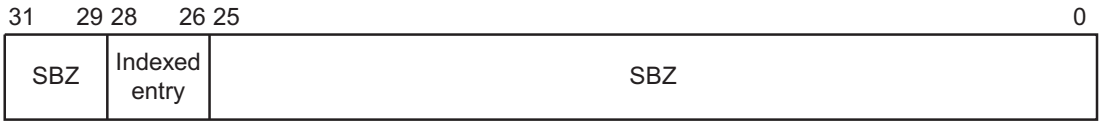


Figure B-6 Rd format for selecting lockdown TLB entry

Table B-8 describes the entry-select bit fields in the Rd register.

Table B-8 Encoding of the lockdown TLB entry-select bit fields

Bit	Name	Definition
[31:29]	-	Should Be Zero
[28:26]	Indexed entry	Indexed entry in lockdown TLB
[25:0]	-	Should Be Zero

2. Use the following MMU Test Register instructions to access the MVA tag:

   ```
   MRC p15, 4, <Rd>, c15, c2, 1 ; read lockdown TLB
   MCR p15, 4, <Rd>, c15, c3, 1 ; write lockdown TLB
   ```

   See Figure B-3 on page B-7 for read or write data in the Rd register.

3. Use the following MMU Test Register instructions to read or write the PA and access permission data:

   ```
   MRC p15, 4, <Rd>, c15, c4, 1 ; read PA and access permission data
   MCR p15, 4, <Rd>, c15, c5, 1 ; write PA and access permission data
   ```

   See Figure B-4 on page B-8 for the read or write data in the Rd register.

4. Use the following instruction to complete a write to an entry:

   ```
   MCR p15, 4, <Rd>, c15, c7, 1 ; transfer lockdown storage into RAM
   ```

To write an entry into the lockdown TLB, the full sequence is therefore:

```
MCR p15, 4/5, <Rd>, c15, c3, 1 ; write tag lockdown TLB storage reg
MCR p15, 4/5, <Rd>, c15, c5, 1 ; write PA/PROT lockdown TLB storage reg
MCR p15, 4/5, <Rd>, c15, c7, 1 ; transfer lockdown storage into RAM
```

To read an entry from the lockdown TLB, the entry must first be written using the above instructions. The entry can then be read using the following instructions:

```
MRC p15, 4/5, <Rd>, c15, c2, 1 ; read tag lockdown TLB
MRC p15, 4/5, <Rd>, c15, c4, 1 ; read PA/PROT lockdown TLB
```

The data to be written or read is placed in ARM register Rd with the format shown in Figure B-4 on page B-8.

B.1.5 Cache Debug Control Register

The Cache Debug Control Register is used to force specific cache behavior required for debug.

The following instructions can be used to access the Cache Debug Control Register:

```
MRC{cond} p15,7,<Rd>,c15,c0,0 ; read cache debug control register
MCR{cond} p15,7,<Rd>,c15,c0,0 ; write cache debug control register
```

The Cache Debug Control Register format is shown in Figure B-7.

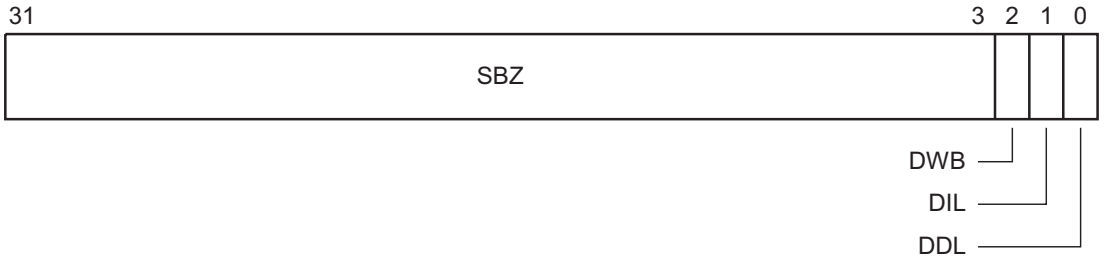


Figure B-7 Cache Debug Control Register format

The Cache Debug Control Register bit assignments are listed in Table B-9. The reset value of the Cache Debug Control Register is `0x0`.

Table B-9 Cache Debug Control Register bit assignments

Bit	Name	Function	Description
[31:3]	-	Reserved	Read = Unpredictable Write = Should Be Zero
[2]	DWB	Disable write-back (force WT)	0 = Enable write-back behavior 1 = Force write-through behavior
[1]	DIL	Disable ICache linefill	0 = Enable ICache linefills 1 = Disable ICache linefills
[0]	DDL	Disable DCache linefill	0 = Enable DCache linefills 1 = Disable DCache linefills

Forcing write-through behavior

Setting the DWB bit to 1 forces the DCache to treat all cachable accesses as though they were in a write-through region of memory. The setting of the DWB bit overrides any setting specified in either the MMU page tables or in the Memory Region Remap Register.

If the cache contains dirty cache lines, these remain dirty while the DWB bit is set, unless they are written back because of a write-back eviction after a linefill, or because of an explicit clean operation.

Lines that are clean are not marked as dirty if they are updated while the DWB bit is set. This functionality allows a debugger to download code or data to external memory, without the requirement to clean part or all of the DCache to ensure that the code or data being downloaded has been written to external memory.

Note

If the DWB bit is set, and a write is made to a cache line that is dirty, then both the cache line and external memory are updated with the write data. Other entries in the cache line still have to be written back to main memory to achieve coherency.

Disabling cache linefills

Setting the DDL and DIL bits prevents the relevant cache from updating when performing a linefill on a miss. When set, a linefill is performed on a cache miss, reading eight words from external memory, but the cache is not updated with the linefill data. The memory region mapping is unchanged. This mode of operation is required for debug so that the memory image, as seen by the ARM9EJ-S core, can be examined in a non-invasive manner. Cache hits from a cachable region read data words from the cache, and cache misses from a cachable region read words directly from memory.

B.1.6 MMU Debug Control Register

You can use the MMU Debug Control Register to enable TLB and micro TLB entries to be preserved during debug. For debug to be non-invasive, bits [5:0] must be set to b111111 prior to changing any other CP15 registers, or issuing any system speed load or store. If main TLB loading is disabled, page table walks still take place, but the resultant data is forwarded around the TLB.

It might be necessary to temporarily change the contents of a page table entry to facilitate debug operations. Disabling main TLB matches using bit 6 or 7 enables the modified contents of the page table to be used for an access without having to invalidate any entries in the main TLB.

You can access the MMU Debug Control Register using the following instructions:

```
MRC{cond} p15,7,<Rd>,c15,c1,0 ; read MMU debug control register
MCR{cond} p15,7,<Rd>,c15,c1,0 ; write MMU debug control register
```

The MMU Debug Control Register format is shown in Figure B-8.

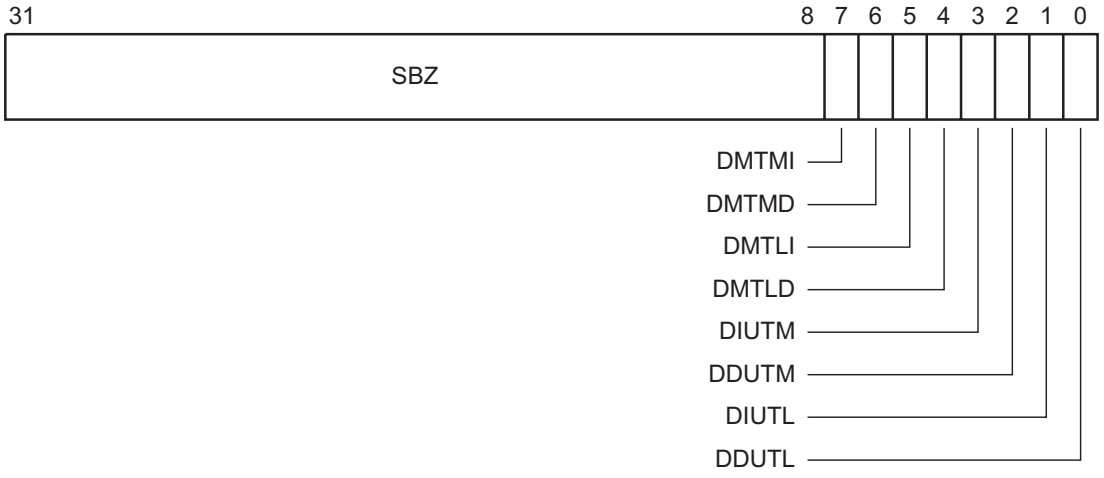


Figure B-8 MMU Debug Control Register format

The MMU Debug Control Register bit assignments are given in Table B-10. The reset value of the MMU Debug Control Register is `0x0`.

Table B-10 MMU Debug Control Register bit assignments

Bit	Name	Function	Description
[31:8]	-	Reserved	Read = Unpredictable Write = Should Be Zero
[7]	DMTMI	Disable main TLB matching for instruction fetches	0 = Enable matching 1 = Disable matching
[6]	DMTMD	Disable main TLB matching for data accesses	0 = Enable matching 1 = Disable matching
[5]	DMTLI	Disable main TLB load because of instruction fetch miss	0 = Enable TLB load 1 = Disable TLB load
[4]	DMTLD	Disable main TLB load because of data access miss	0 = Enable TLB load 1 = Disable TLB load

Table B-10 MMU Debug Control Register bit assignments (continued)

Bit	Name	Function	Description
[3]	DIUTM	Disable instruction micro TLB match	0 = Enable I-micro TLB load 1 = Disable I-micro TLB load
[2]	DDUTM	Disable data micro TLB match	0 = Enable D-micro TLB match 1 = Disable D-micro TLB match
[1]	DIUTL	Disable instruction micro TLB load	0 = Enable D-micro TLB load 1 = Disable D-micro TLB load
[0]	DDUTL	Disable data micro TLB load	0 = Enable I-micro TLB load 1 = Disable I-micro TLB load

B.1.7 Memory Region Remap Register

The read/write Memory Region Remap Register overrides the setting specified in the MMU page tables, and the default behavior if the MMU is disabled.

The Memory Region Register has four fields for remapping instruction-side memory regions and four fields for remapping data-side memory regions.

You can access the Memory Region Remap Register with the instructions in Table B-11.

Table B-11 Memory Region Remap Register instructions

Instruction	Operation
`MRC p15, 0, Rd, c15, c2, 0`	Read Memory Region Remap Register
`MCR p15, 0, Rd, c15, c2, 0`	Write Memory Region Remap Register

Figure B-9 shows the bit fields of the Memory Region Remap Register.

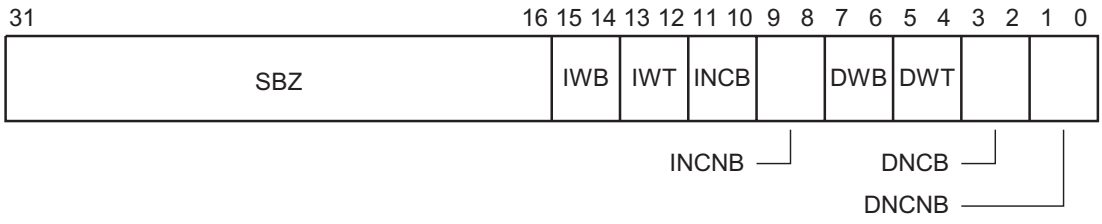


Figure B-9 Memory Region Remap Register format

Table B-12 describes the bit fields of the Memory Region Remap Register.

Table B-12 Encoding of the Memory Region Remap Register

Bit	Name	Definition	Reset state
[31:16]	-	Should Be Zero	0x0000
[15:14]	IWB	Remap select bits for instruction-side write-back region	b11
[13:12]	IWT	Remap select bits for instruction-side write-through region	b10
[11:10]	INCB	Remap select bits for instruction-side noncachable bufferable region	b01
[9:8]	INCNB	Remap select bits for instruction-side noncachable nonbufferable region	b00
[7:6]	DWB	Remap select bits for data-side write-back region	b11
[5:4]	DWT	Remap select bits for data-side write-through region	b10
[3:2]	DNCB	Remap select bits for data-side noncachable bufferable region	b01
[1:0]	DNCNB	Remap select bits for data-side noncachable nonbufferable region	b00

Table B-13 shows the encoding of each of the remap fields.

Table B-13 Encoding of the remap fields

Remap field
b00 = noncachable nonbufferable
b01 = noncachable bufferable
b10 = write-through
b11 = write-back

Figure B-10 shows the flow and precedence of CP15 c15 control bits in resolving the cachable and bufferable attributes of a memory reference.

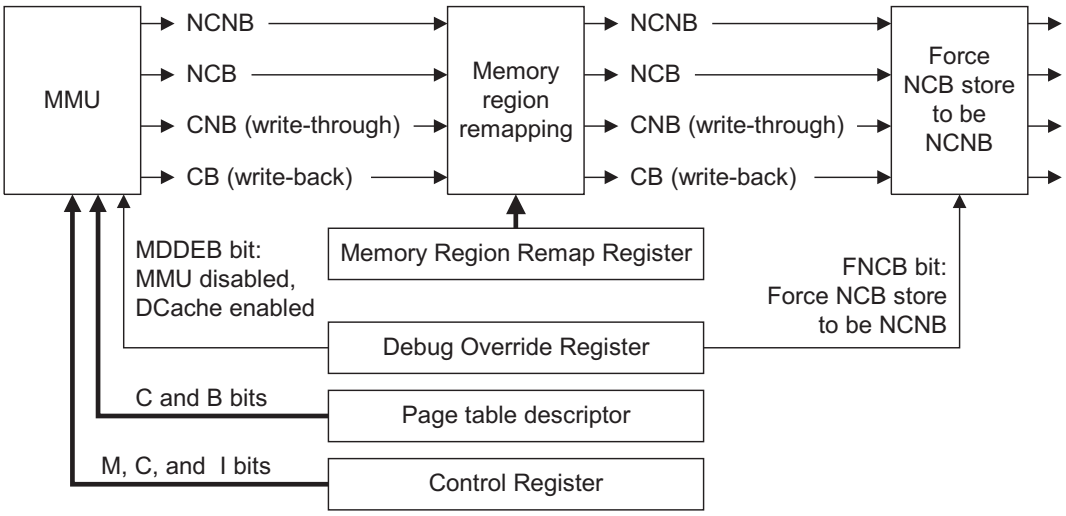


Figure B-10 Memory region attribute resolution

Glossary

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

Abort A mechanism that indicates to a core that it must halt execution of an attempted illegal memory access. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a Prefetch or Data Abort, and an internal or External Abort.

See also Data Abort, External Abort and Prefetch Abort.

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

Access permission The mechanism that controls if a task or process is allowed to access sections or pages of memory. If an access is attempted to an area of memory without the required permissions, a permission fault is raised.

Addressing modes A mechanism, shared by many different instructions, for generating values used by the instructions. For four of the ARM addressing modes, the values generated are memory addresses (which is the traditional role of an addressing mode). A fifth addressing mode generates values to be used as operands by data-processing instructions.

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

See also Advanced Microcontroller Bus Architecture and AHB-Lite.

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

Advanced Peripheral Bus (APB)
The AMBA Advanced Peripheral Bus is a simpler bus protocol than AHB. It is designed for use with ancillary or general-purpose peripherals such as timers, interrupt controllers, UARTs, and I/O ports. Connection to the main system bus is through a system-to-peripheral bus bridge that helps to reduce system power consumption.

See also Advanced High-performance Bus.

AHB *See* Advanced High-performance Bus.

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

AMBA *See* Advanced Microcontroller Bus Architecture.

AP See *Access permission*.

APB *See* Advanced Peripheral Bus.

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

Application Specific Standard Part/Product (ASSP)
An integrated circuit that has been designed to perform a specific application function. Usually consists of two or more separate circuit functions combined as a building block suitable for use in a range of products for one or more specific application markets.

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

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

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

ASIC *See* Application Specific Integrated Circuit.

ASSP *See* Application Specific Standard Part/Product.

ATPG *See* Automatic Test Pattern Generation.

Automatic Test Pattern Generation (ATPG)
The process of automatically generating manufacturing test vectors for an ASIC design, using a specialized software tool.

Back-annotation The process of applying timing characteristics from the implementation process onto a model.

Banked registers Those physical registers whose use is defined by the current processor mode. The banked registers are r8 to r14.

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

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

Beat Alternative word for an individual transfer within a burst. For example, an INCR4 burst comprises four beats.

See also Burst.

Big-endian Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.

See also Little-endian and Endianness.

Big-endian memory Memory in which:

- a byte or halfword at a word-aligned address is the most significant byte or halfword within the word at that address
- a byte at a halfword-aligned address is the most significant byte within the halfword at that address.

See also Little-endian memory.

Block address An address that comprises a tag, an index, and a word field. The tag bits identify the way that contains the matching cache entry for a cache hit. The index bits identify the set being addressed. The word field contains the word address that can be used to identify specific words, halfwords, or bytes within the cache entry.

See also Cache terminology diagram on the last page of this glossary.

Boundary scan chain

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

Breakpoint A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested.

See also Watchpoint.

Burst A group of transfers to consecutive addresses. Because the addresses are consecutive, there is no requirement to supply an address for any of the transfers after the first one. This increases the speed at which the group of transfers can occur. Bursts over AHB buses are controlled using the **HBURST** signals to specify if transfers are single, four-beat, eight-beat, or 16-beat bursts, and to specify how the addresses are incremented.

See also Beat.

Bus Interface Unit The *Bus Interface Unit* (BIU) controls all data accesses across the AHB. It arbitrates and schedules AHB requests.

Byte An 8-bit data item.

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

See also Cache terminology diagram on the last page of this glossary.

Cache contention
When the number of frequently-used memory cache lines that use a particular cache set exceeds the set-associativity of the cache. In this case, main memory activity increases and performance decreases.

Cache hit
A memory access that can be processed at high speed because the instruction or data that it addresses is already held in the cache.

Cache line
The basic unit of storage in a cache. It is always a power of two words in size (usually four or 8 words), and is required to be aligned to a suitable memory boundary.

See also Cache terminology diagram on the last page of this glossary.

Cache line index
The number associated with each cache line in a cache way. Within each cache way, the cache lines are numbered from 0 to (set associativity) -1.

See also Cache terminology diagram on the last page of this glossary.

Cache lockdown
To fix a line in cache memory so that it cannot be overwritten. Cache lockdown enables critical instructions and/or data to be loaded into the cache so that the cache lines containing them are not subsequently reallocated. This ensures that all subsequent accesses to the instructions/data concerned are cache hits, and therefore complete as quickly as possible.

Cache miss
A memory access that cannot be processed at high speed because the instruction/data it addresses is not in the cache and a main memory access is required.

Cache set
A cache set is a group of cache lines (or blocks). A set contains all the ways that can be addressed with the same index. The number of cache sets is always a power of two.

See also Cache terminology diagram on the last page of this glossary.

Cache way
A group of cache lines (or blocks). It is 2 to the power of the number of index bits in size.

See also Cache terminology diagram on the last page of this glossary.

CAM
See Content Addressable Memory.

Cast out
See Victim.

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

See also Dirty.

Clock gating Gating a clock signal for a macrocell with a control signal and using the modified clock that results to control the operating state of the macrocell.

Clocks Per Instruction *See* Cycles Per Instuction.

Coherency *See* Memory coherency.

Cold reset Also known as power-on reset. Starting the processor by turning power on. Turning power off and then back on again clears main memory and many internal settings. Some program failures can lock up the processor and require a cold reset to enable the system to be used again. In other cases, only a warm reset is required.

See also Warm reset.

Communications channel
The hardware used for communicating between the software running on the processor, and an external host, using the debug interface. When this communication is for debug purposes, it is called the Debug Comms Channel. In an ARMv6 compliant core, the communications channel includes the Data Transfer Register, some bits of the Data Status and Control Register, and the external debug interface controller, such as the DBGTAP controller in the case of the JTAG interface.

Condensed Reference Format (CRF)
An ARM proprietary file format for specifying test vectors.

Condition field A 4-bit field in an instruction that is used to specify a condition under which the instruction can execute.

Conditional execution
If the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

Content Addressable Memory (CAM)
Memory that is identified by its contents. Content Addressable Memory is used in CAM-RAM architecture caches to store the tags for cache entries. addressable memory.

CAM includes comparison logic with each bit of storage. A data value is broadcast to all words of storage and compared with the values there. Words that match are flagged in some way. Subsequent operations can then work on flagged words. It is possible to read the flagged words out one at a time or write to certain bit positions in all of them.

Context The environment that each process operates in for a multitasking operating system. In ARM processors, this is limited to mean the Physical Address range that it can access in memory and the associated memory access permissions.

See also Fast context switch.

Control bits The bottom eight bits of a Program Status Register (PSR). The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

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

Copy back *See* Write-back.

Core A core is that part of a processor that contains the ALU, the datapath, the general-purpose registers, the Program Counter, and the instruction decode and control circuitry.

Core module In the context of an ARM Integrator, a core module is an add-on development board that contains an ARM processor and local memory. Core modules can run standalone, or can be stacked onto Integrator motherboards.

Core reset *See* Warm reset.

CPI *See* Cycles per instruction.

CPSR *See* Current Program Status Register

CRF *See* Condensed Reference Format.

Current Program Status Register (CPSR)
The register that holds the current operating processor status.

Cycles Per instruction (CPI)
Cycles per instruction (or clocks per instruction) is a measure of the number of computer instructions that can be performed in one clock cycle. This figure of merit can be used to compare the performance of different CPUs against each other. The lower the value, the better the performance.

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

See also Abort, External Abort, and Prefetch Abort.

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

DBGTAP *See* Debug Test Access Port.

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

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

Debug Test Access Port (DBGTAP)
The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **DBGTDI**, **DBGTDO**, **DBGTMS**, and **TCK**. The optional terminal is **TRST** (**DBGnTRST**). This signal is mandatory in ARM cores because it is used to reset the debug logic.

Direct-mapped cache
A one-way set-associative cache. Each cache set consists of a single cache line, so cache look-up selects and checks a single cache line.

Direct Memory Access (DMA)
An operation that accesses main memory directly, without the processor performing any accesses to the data concerned.

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

See also Clean.

DMA *See* Direct Memory Access.

DNM *See* Do Not Modify.

Domain A collection of sections, large pages and small pages of memory, which can have their access permissions switched rapidly by writing to the Domain Access Control Register (CP15 register c3).

Do Not Modify (DNM)
In Do Not Modify fields, the value must not be altered by software. DNM fields read as Unpredictable values, and must only be written with the same value read from the same field on the same processor.
Throughout this manual, DNM fields are sometimes followed by RAZ or RAO in parentheses to show which way the bits should read for future compatibility, but programmers must not rely on this behavior.

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

Doubleword-aligned
A data item having a memory address that is divisible by 8.

EmbeddedICE logic An on-chip logic block that provides TAP-based debug support for ARM processor cores. It is accessed through the TAP controller on the ARM core using the JTAG interface.

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

Embedded Trace Buffer
The ETB provides on-chip storage of trace data using a configurable sized RAM.

Embedded Trace Macrocell (ETM)
A hardware macrocell which, when connected to a processor core, outputs instruction and data trace information on a trace port. The ETM provides processor driven trace through a trace port compliant to the ATB protocol.

Endianness Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory. An aspect of the system's memory mapping.

See also Little-endian and Big-endian

ETM See *Embedded Trace Macrocell*.

Event 1 (Simple) An observable condition that can be used by an ETM to control aspects of a trace.

2 (Complex) A boolean combination of simple events that is used by an ETM to control aspects of a trace.

Exception A fault or error event that is considered serious enough to require that program execution is interrupted. Examples include attempting to perform an invalid memory access, external interrupts, and undefined instructions. When an exception occurs, normal program flow is interrupted and execution is resumed at the corresponding exception vector. This contains the first instruction of the interrupt handler to deal with the exception.

Exception service routine
See Interrupt handler.

Exception vector *See* Interrupt vector.

External Abort An indication from an external memory system to a core that it must halt execution of an attempted illegal memory access. An External Abort is caused by the external memory system as a result of attempting to access invalid memory.

See also Abort, Data Abort and Prefetch Abort.

Fast context switch
In a multitasking system, the point at which the time-slice allocated to one process stops and the one for the next process starts. If processes are switched often enough, they can appear to a user to be running in parallel, as well as being able to respond quicker to external events that might affect them.

In ARM processors, a fast context switch is caused by the selection of a non-zero PID value to switch the context to that of the next process. A fast context switch causes each Virtual Address for a memory access, generated by the ARM processor, to produce a Modified Virtual Address which is sent to the rest of the memory system to be used in place of a normal Virtual Address. For some cache control operations Virtual Addresses are passed to the memory system as data. In these cases no address modification takes place.

See also Fast Context Switch Extension.

Fast Context Switch Extension (FCSE)
An extension to the ARM architecture that enables cached processors with an MMU to present different addresses to the rest of the memory system for different software processes, even when those processes are using identical addresses.

See also Fast context switch.

FCSE *See* Fast Context Switch Extension.

Flat address mapping
A system of organizing memory in which each Physical Address contained within the memory space is the same as its corresponding Virtual Address.

Fully-associative cache
A cache that has just one cache set that consists of the entire cache. The number of cache entries is the same as the number of cache ways.

See also Direct-mapped cache.

Half-rate clocking (ETM)
Dividing the trace clock by two so that the TPA can sample trace data signals on both the rising and falling edges of the trace clock. The primary purpose of half-rate clocking is to reduce the signal transition rate on the trace clock of an ASIC for very high-speed systems.

Halfword A 16-bit data item.

Halt mode One of two mutually exclusive debug modes. In halt mode all processor execution halts when a breakpoint or watchpoint is encountered. All processor state, coprocessor state, memory and input/output locations can be examined and altered by the JTAG interface.

See also Monitor debug-mode.

High vectors Alternative locations for exception vectors. The high vector address range is near the top of the address space, rather than at the bottom.

Host A computer that provides data and other services to another computer. Especially, a computer providing debugging services to a target being debugged.

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

IGN *See* Ignore.

Ignore (IGN) Must ignore memory writes.

Illegal instruction An instruction that is architecturally Undefined.

IMB *See* Instruction Memory Barrier.

Implementation-defined
Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

Implementation-specific
Means that the behavior is not architecturally defined, and does not have to be documented by individual implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

Index *See* Cache index.

Index register A register specified in some load or store instructions. The value of this register is used as an offset to be added to or subtracted from the base register value to form the virtual address, which is sent to memory. Some addressing modes optionally enable the index register value to be shifted prior to the addition or subtraction.

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

Instruction cycle count
The number of cycles for which an instruction occupies the Execute stage of the pipeline.

Instruction Memory Barrier (IMB)
An operation to ensure that the prefetch buffer is flushed of all out-of-date instructions.

Internal scan chain A series of registers connected together to form a path through a device, used during production testing to import test patterns into internal nodes of the device and export the resulting values.

Interrupt handler A program that control of the processor is passed to when an interrupt occurs.

Interrupt vector One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt handler.

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

Joint Test Action Group (JTAG)
The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices. It is commonly known by the initials JTAG.

JTAG *See* Joint Test Action Group.

Line *See* Cache line.

Little-endian Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

See also Big-endian and Endianness.

Little-endian memory
Memory in which:
- a byte or halfword at a word-aligned address is the least significant byte or halfword within the word at that address
- a byte at a halfword-aligned address is the least significant byte within the halfword at that address.

See also Big-endian memory.

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

Load Store Unit (LSU)
The part of a processor that handles load and store transfers.

LSU *See* Load Store Unit.

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

Memory bank One of two or more parallel divisions of interleaved memory, usually one word wide, that enable reads and writes of multiple words at a time, rather than single words. All memory banks are addressed simultaneously and a bank enable or chip select signal determines which of the banks is accessed for each transfer. Accesses to sequential word addresses cause accesses to sequential banks. This enables the delays associated with accessing a bank to occur during the access to its adjacent bank, speeding up memory transfers.

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

Memory Management Unit (MMU)
Hardware that controls caches and access permissions to blocks of memory, and translates virtual addresses to physical addresses.

Memory Protection Unit (MPU)
Hardware that controls access permissions to blocks of memory. Unlike an MMU, an MPU does not translate virtual addresses to physical addresses.

Microprocessor *See* Processor.

Miss *See* Cache miss.

MMU *See* Memory Management Unit.

Modified Virtual Address (MVA)
A Virtual Address produced by the ARM processor can be changed by the current Process ID to provide a *Modified Virtual Address* (MVA) for the MMUs and caches.

See also Fast Context Switch Extension.

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

See also Halt mode.

MPU *See* Memory Protection Unit.

Multi-ICE A JTAG-based tool for debugging embedded systems.

MVA *See* Modified Virtual Address.

NCB See *Noncachable Buffered*.

NCNB See *Noncachable Nonbufferable*.

Noncachable Buffered Is a memory region where reads are performed from main memory and are not allocated to the cache. Writes are performed to main memory through a write buffer, so processor core execution can continue while the write is completed to main memory.

Noncachable Nonbufferable Is a memory region where reads are performed from main memory and are not allocated to the cache. Writes are performed to main memory without buffering, so processor core execution is halted while the write is completed.

PA *See* Physical Address.

Penalty The number of cycles in which no useful Execute stage pipeline activity can occur because an instruction flow is different from that assumed or predicted.

Power-on reset *See* Cold reset.

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

Prefetch Abort An indication from a memory system to a core that it must halt execution of an attempted illegal memory access. A Prefetch Abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

See also Data Abort, External Abort and Abort.

Processor
A processor is the circuitry in a computer system required to process data using the computer instructions. It is an abbreviation of microprocessor. A clock source, power supplies, and main memory are also required to create a minimum complete working computer system.

Physical Address (PA)
The MMU performs a translation on *Modified Virtual Addresses* (MVA) to produce the *Physical Address* (PA) which is given to AHB to perform an external access. The PA is also stored in the data cache to avoid the necessity for address translation when data is cast out of the cache.

See also Fast Context Switch Extension.

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

RealView ICE
A system for debugging embedded processor cores using a JTAG interface.

Region
A partition of instruction or data memory space.

Remapping
Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM when the initialization has been completed.

Reserved
A field in a control register or instruction format is reserved if the field is to be defined by the implementation, or produces Unpredictable results if the contents of the field are not zero. These fields are reserved for use in future extensions of the architecture or are implementation-specific. All reserved bits not used by the implementation must be written as 0 and read as 0.

Saved Program Status Register (SPSR)
The register that holds the CPSR of the task immediately before the exception occurred that caused the switch to the current mode.

SBO
See Should Be One.

SBZ
See Should Be Zero.

SBZP
See Should Be Zero or Preserved.

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

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

Set *See* Cache set.

Set-associative cache
In a set-associative cache, lines can only be placed in the cache in locations that correspond to the modulo division of the memory address by the number of sets. If there are *n* ways in a cache, the cache is termed *n*-way set-associative. The set-associativity can be any number greater than or equal to 1 and is not restricted to being a power of two.

Short vector operation
An operation involving more than one destination register and perhaps more than one source register in the generation of the result for each destination.

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

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

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

SPICE Simulation Program with Integrated Circuit Emphasis. An accurate transistor-level electronic circuit simulation tool that can be used to predict how an equivalent real circuit will behave for given circuit conditions.

SPSR *See* Saved Program Status Register

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

See also Cache terminology diagram on the last page of this glossary.

TAP *See* Test access port.

TCM *See* Tightly coupled memory.

Test Access Port (TAP)
The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **TRST**. This signal is mandatory in ARM cores because it is used to reset the debug logic.

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

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

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

TLB *See* Translation Look-aside Buffer.

Translation Lookaside Buffer (TLB)
A cache of recently used page table entries that avoid the overhead of page table walking on every memory access. Part of the Memory Management Unit.

Translation table A table, held in memory, that contains data that defines the properties of memory areas of various fixed sizes.

Translation table walk
The process of doing a full translation table lookup. It is performed automatically by hardware.

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

Unpredictable Means that the behavior of the ETM cannot be relied upon. Such conditions have not been validated. When applied to the programming of an event resource, only the output of that event resource is Unpredictable.

Unpredictable behavior can affect the behavior of the entire system, because the ETM is capable of causing the core to enter debug state, and external outputs may be used for other purposes.

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

VA *See* Virtual Address.

Victim A cache line, selected to be discarded to make room for a replacement cache line that is required as a result of a cache miss. The way in which the victim is selected for eviction is processor-specific. A victim is also known as a cast out.

Virtual Address (VA)

The MMU uses its page tables to translate a Virtual Address into a Physical Address. The processor executes code at the Virtual Address, which might be located elsewhere in physical memory.

See also Fast Context Switch Extension, Modified Virtual Address, and Physical Address.

Warm reset Also known as a core reset. Initializes the majority of the processor excluding the debug controller and debug logic. This type of reset is useful if you are using the debugging features of a processor.

Watchpoint A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to allow inspection of register contents, memory locations, and variable values when memory is written to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested. *See also* Breakpoint.

Way *See* Cache way.

WB *See* Write-back.

Word A 32-bit data item.

Write Writes are defined as operations that have the semantics of a store. That is, the ARM instructions SRS, STM, STRD, STC, STRT, STRH, STRB, STRBT, STREX, SWP, and SWPB, and the Thumb instructions STM, STR, STRH, STRB, and PUSH. Java instructions that are accelerated by hardware can cause a number of writes to occur, according to the state of the Java stack and the implementation of the Java hardware acceleration.

Write-back (WB) In a write-back cache, data is only written to main memory when it is forced out of the cache on line replacement following a cache miss. Otherwise, writes by the processor only update the cache. (Also known as copyback).

Write buffer A block of high-speed memory, arranged as a FIFO buffer, between the data cache and main memory, whose purpose is to optimize stores to main memory.

Write completion
The memory system indicates to the processor that a write has been completed at a point in the transaction where the memory system is able to guarantee that the effect of the write is visible to all processors in the system. This is not the case if the write is associated with a memory synchronization primitive, or is to a Device or Strongly Ordered region. In these cases the memory system might only indicate completion of the write when the access has affected the state of the target, unless it is impossible to distinguish between having the effect of the write visible and having the state of target updated.

This stricter requirement for some types of memory ensures that any side-effects of the memory access can be guaranteed by the processor to have taken place. You can use this to prevent the starting of a subsequent operation in the program order until the side-effects are visible.

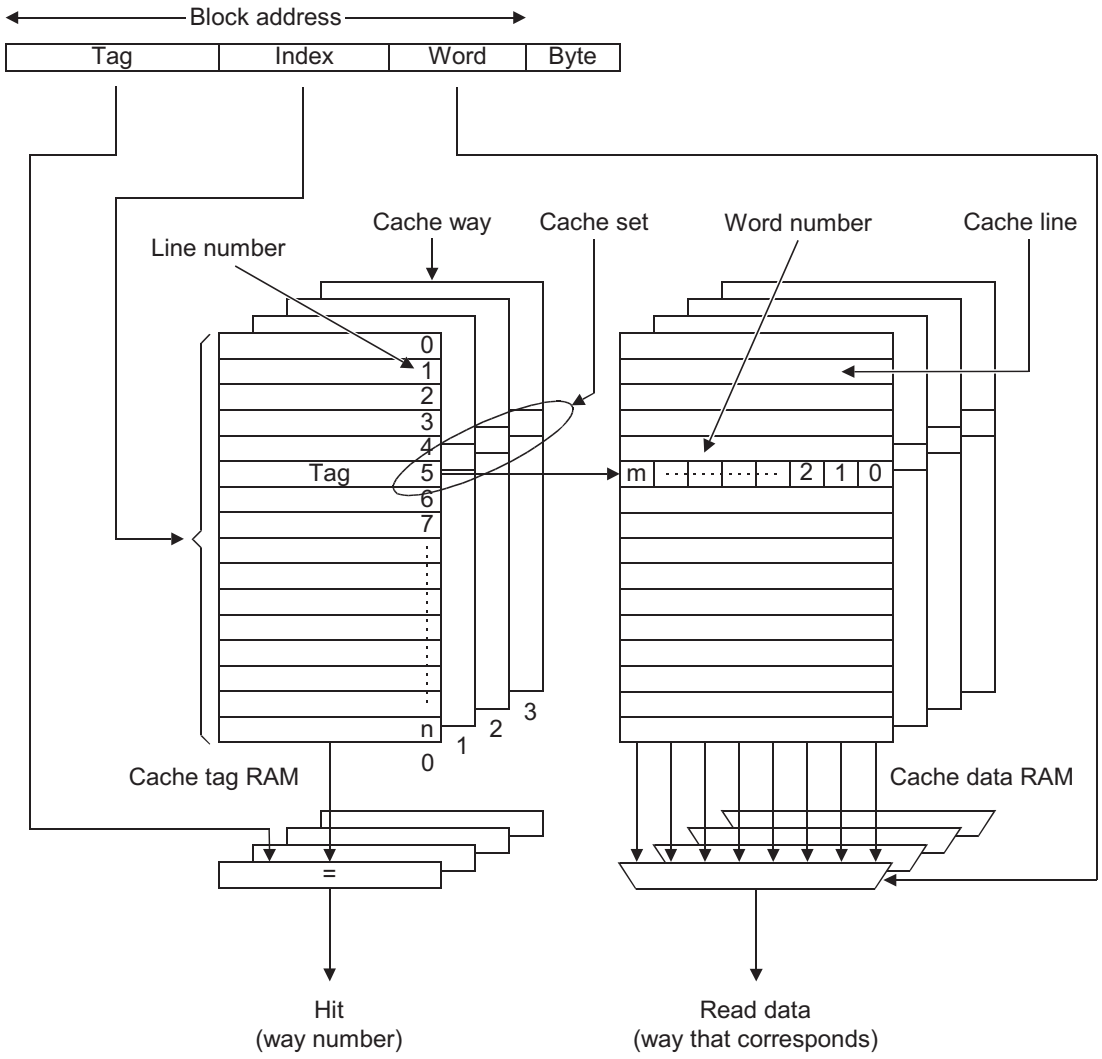
Write-through (WT)
In a write-through cache, data is written to main memory at the same time as the cache is updated.

WT
See Write-through.

Cache terminology diagram

The diagram below illustrates the following cache terminology:

- block address
- cache line
- cache set
- cache way
- index
- tag.



Index

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

D

E

F

H

I

J

L

M

N

O

P

R

S

T

U

V

W

Z

