ARM720T

(Rev 4)

Technical Reference Manual

ARM

ARM720T
Technical Reference Manual

Copyright © 2001 ARM Limited. All rights reserved.

Release Information

Change history

Date	Issue	Change
22 November 2001	A	ARM720T (Rev 4) first release

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Figure 9-8 on page 9-27 reprinted with permission IEEE Std. 1149.1-1990. IEEE Standard Test Access Port and Boundary Scan Architecture Copyright 2001, by IEEE. The IEEE disclaims any responsibility or liability resulting from the placement and use in the described manner.

Confidentiality Status

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

Product Status

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

Web Address

http://www.arm.com

Contents
ARM720T Technical Reference Manual

Appendix A   **Signal Descriptions**

Glossary

List of Tables
ARM720T Technical Reference Manual

ARM DDI 0229A

List of Figures
ARM720T Technical Reference Manual

Preface

This preface introduces the ARM720T (Rev 4) processor and its reference documentation. It contains the following sections:

- *About this document* on page xiv
- *Feedback* on page xviii.

About this document

This document is a technical reference manual for the ARM720T (Rev 4) processor.

Intended audience

This document has been written for experienced hardware and software engineers who might or might not have experience of the architecture, configuration, integration, and instruction sets with reference to the ARM product range.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM720T processor.

Chapter 2 *Programmer's Model*

Read this chapter for a description of the 32-bit ARM and 16-bit Thumb instruction sets.

Chapter 3 *Configuration*

Read this chapter for a description of how to use the system control coprocessor, CP15, to configure devices that are peripheral to the core.

Chapter 4 *Instruction and Data Cache*

Read this chapter for an overview of the mixed instruction and data cache.

Chapter 5 *Write Buffer*

Read this chapter for a description of how you can enhance the system performance of the ARM720T processor by using the write buffer.

Chapter 6 *The Bus Interface*

Read this chapter for a description of the ARM720T processor bus interface.

Chapter 7 *Memory Management Unit*

Read this chapter for a description of the functions and use of the memory management unit.

Chapter 8 *Coprocessor Interface*

Read this chapter for information about connecting and using external coprocessors.

Chapter 9 *Debugging Your System*

Read this chapter for a description of the hardware extensions and integrated on-chip debug support for the ARM720T processor.

Chapter 10 *ETM Interface*

Read this chapter for a description of the Embedded Trace Macrocell support for the ARM720T processor.

Chapter 11 *Test Support*

Read this chapter for information about performing device-specific test operations.

Appendix A *Signal Descriptions*

Read this appendix for a list of all ARM720T processor interface signals.

Typographical conventions

The following typographical conventions are used in this document:

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

italic
Highlights special terminology, cross-references, and citations.

monospace
Denotes text that can be entered at the keyboard, such as commands, file names and program names, and source code.

<u>mono</u>space
Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic
Denotes arguments to commands or functions where the argument is to be replaced by a specific value.

monospace bold
Denotes language keywords when used outside example code.

Timing diagram conventions

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

Clock	
HIGH to LOW	
Transient	
HIGH/LOW to HIGH	
Bus stable	
Bus to high impedance	
Bus change	
High impedance to stable bus	
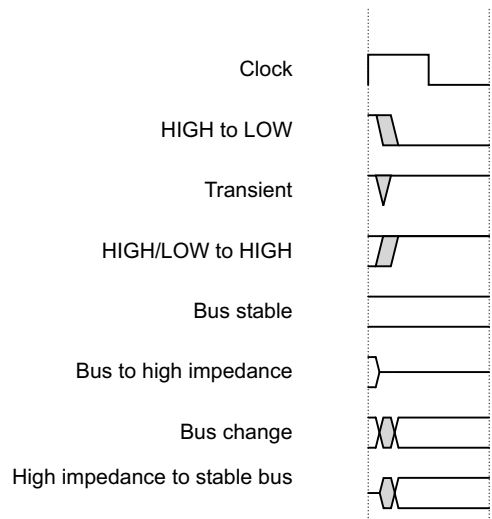
Key to timing diagram conventions

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

Further reading

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

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

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

ARM publications

This document contains information that is specific to the ARM720T processor. Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *AMBA Specification (Rev 2.0)* (ARM IHI 0011)
- *ETM7 (Rev 1) Technical Reference Manual* (ARM DDI 0158)
- *ARM7TDMI-S (Rev 4) Technical Reference Manual* (ARM DDI 0234).

 ARM DDI 0229A

Other publications

This section lists relevant documents published by third parties.

* *Standard Test Access Port and Boundary Scan Architecture* (IEEE Std. 1149.1.1990).

Feedback

ARM Limited welcomes feedback both on the ARM720T processor, and on the documentation.

Feedback on the ARM720T processor

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

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

Feedback on the ARM720T documentation

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

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

General suggestions for additions and improvements are also welcome.

         ARM DDI 0229A

Chapter 1
Introduction

This chapter provides an introduction to the ARM720T processor. It contains the following sections:

- *About the ARM720T (Rev 4) macrocell* on page 1-2
- *Coprocessors* on page 1-7
- *About the instruction set* on page 1-8.

1.1 About the ARM720T (Rev 4) macrocell

The ARM720T (Rev 4) macrocell is a general-purpose 32-bit microprocessor with 8KB cache, enlarged write buffer, and *Memory Management Unit* (MMU) combined in a single chip. The CPU within the ARM720T macrocell is the ARM7TDMI-S processor. The ARM720T (Rev 4) macrocell is software-compatible with the ARM processor family.

The on-chip mixed data and instruction cache, together with the write buffer, substantially raise the average execution speed and reduce the average amount of memory bandwidth required by the processor. This enables the external memory to support additional processors or *Direct Memory Access* (DMA) channels with minimal performance loss.

The MMU supports a conventional two-level page-table structure and several extensions that make it ideal for embedded control, UNIX, and object-oriented systems.

The allocation of virtual addresses with different task IDs improves performance in task switching operations with the cache enabled. These relocated virtual addresses are monitored by the EmbeddedICE-RT block.

The memory interface enables the performance potential to be realized without incurring high costs in the memory system. Speed-critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic. These control signals permit the exploitation of paged mode access offered by industry-standard DRAMs.

The ARM720T processor is provided with an *Embedded Trace Macrocell* (ETM) interface that brings out the required signals from the ARM core to the periphery of the ARM720T macrocell. This enables you to connect a standard ETM7 macrocell.

The ARM720T macrocell is a fully static part and has been designed to minimize power requirements. This makes it ideal for portable applications where low power consumption is essential.

The ARM720T processor architecture is based on *Reduced Instruction Set Computer* (RISC) principles. The instruction set and related decode mechanism are greatly simplified compared with microprogrammed *Complex Instruction Set Computers* (CISCs).

A block diagram of the ARM720T macrocell is shown in Figure 1-1 on page 1-3.
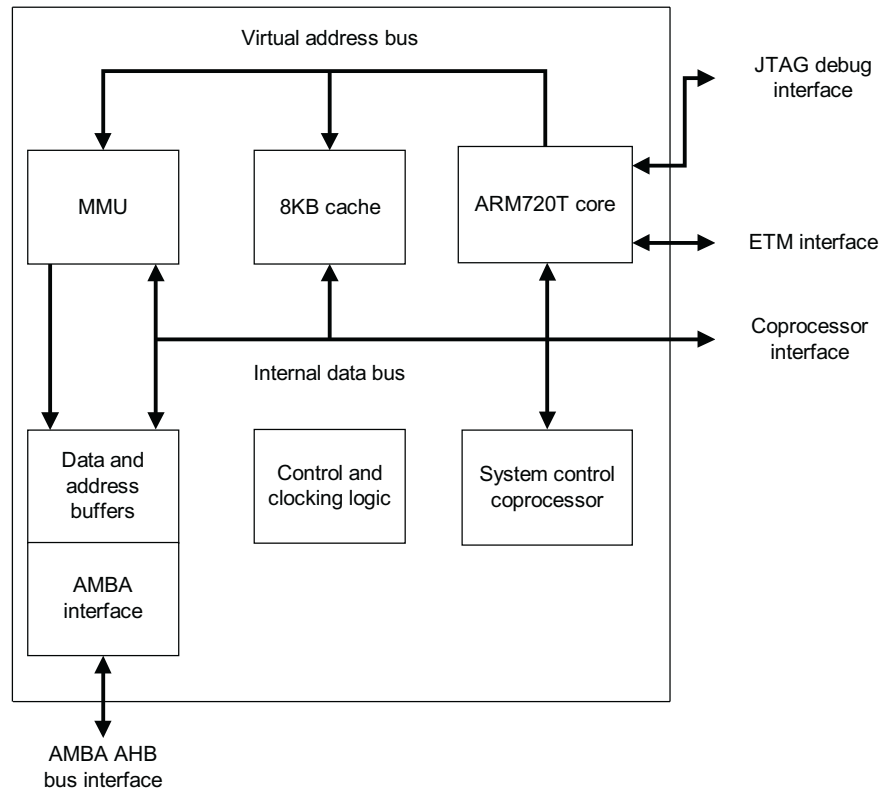
Figure 1-1 Block diagram

The functional signals on the ARM720T (Rev 4) processor are shown in Figure 1-2 on page 1-4.
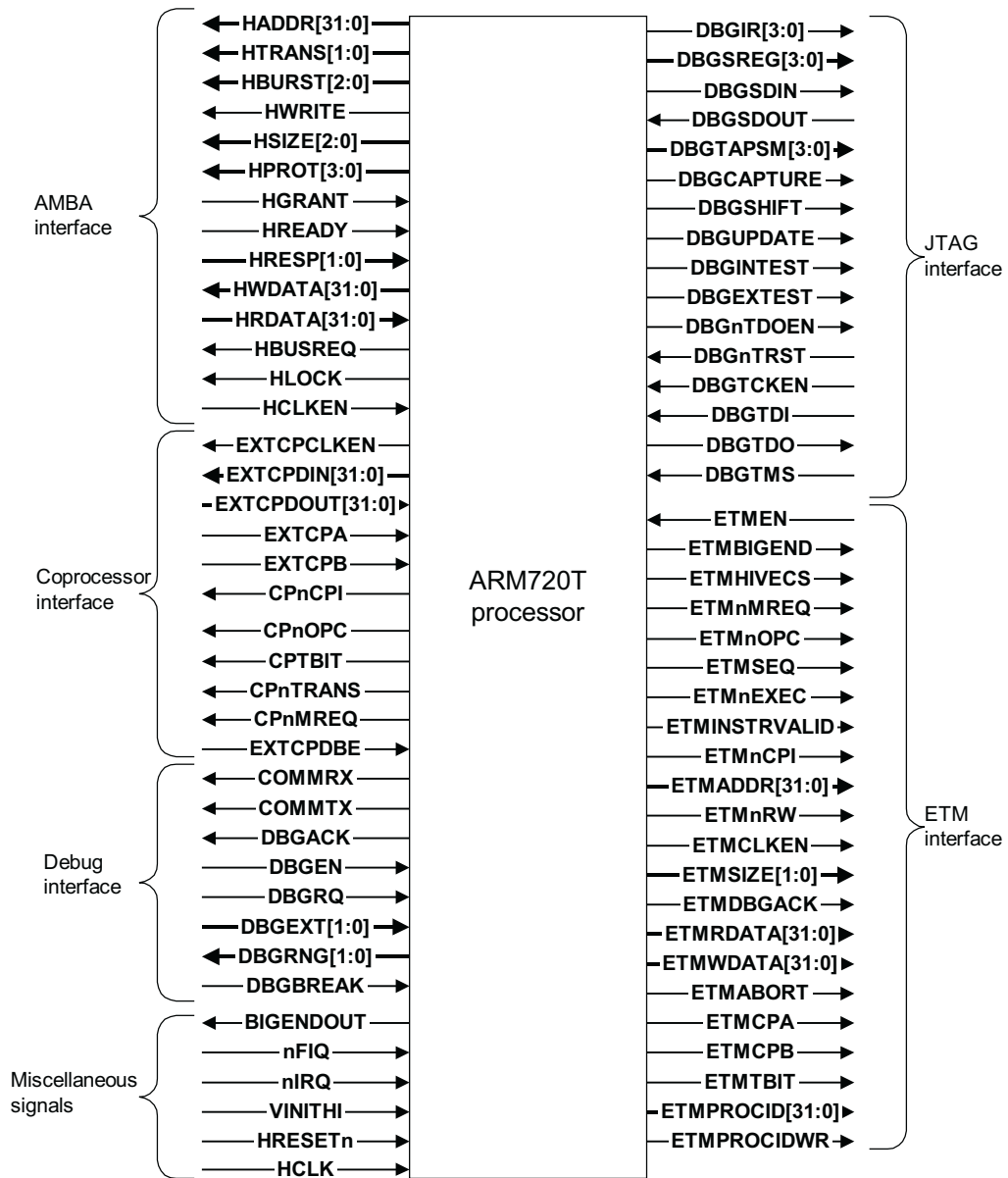
Figure 1-2 ARM720T (Rev 4) processor functional signals

1.1.1 EmbeddedICE-RT logic

The EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM720T core. It enables you to program the conditions under which a breakpoint or watchpoint can occur.

The EmbeddedICE-RT logic is an enhanced implementation of EmbeddedICE, and enables you to perform debugging in *monitor mode*. In monitor mode, the core takes an exception on a breakpoint or watchpoint, rather than entering debug state as it does in *halt mode*.

If the core does not enter debug state when it encounters a watchpoint or breakpoint, it can continue to service hardware interrupt requests as normal. Debugging in monitor mode is extremely useful if the core forms part of the feedback loop of a mechanical system, where stopping the core can potentially lead to system failure.

The EmbeddedICE-RT logic contains a *Debug Communications Channel* (DCC). The DCC is used to pass information between the target and the host debugger. The EmbeddedICE-RT logic is controlled through the *Joint Test Action Group* (JTAG) test access port.

Changes to the programmer's model

To provide support for the EmbeddedICE-RT macrocell, the following changes have been made to the programmer's model for the ARM720T (Rev 4) processor:

Debug control register

There are two new bits in the debug control register:

Bit 4   Monitor mode enable. Use this to control how the device reacts on a breakpoint or watchpoint:
- When set, the core takes the instruction or data abort exception.
- When clear, the core enters debug state.

Bit 5   EmbeddedICE-RT disable. Use this when changing watchpoints and breakpoints:
- When set, this bit disables breakpoints and watchpoints, enabling the breakpoint or watchpoint registers to be programmed with new values.
- When clear, the new breakpoint or watchpoint values become operational.

For more information, see *Debug control register* on page 9-62.

Coprocessor register map

A new register (R2) in the coprocessor register map indicates whether the processor entered the Prefetch or Data Abort exception because of a real abort, or because of a breakpoint or watchpoint. For more details, see *Abort status register* on page 9-61.

For more details, see Chapter 9 *Debugging Your System*.

 ARM DDI 0229A

1.2   Coprocessors

The ARM720T macrocell has an internal coprocessor designated CP15 for internal control of the device (see Chapter 3 *Configuration*).

The ARM720T macrocell also includes a port for the connection of on-chip external coprocessors. This enables extension of the ARM720T functionality in an architecturally-consistent manner.

1.3 About the instruction set

The instruction set comprises ten basic instruction types:

- Two types use the on-chip arithmetic logic unit, barrel shifter, and multiplier to perform high-speed operations on the data in a bank of 31 registers, each 32 bits wide.

- Three types of instruction control the data transfer between memory and the registers:
    — one optimized for flexibility of addressing
    — one for rapid context switching
    — one for swapping data.

- Two instructions control the flow and privilege level of execution.

- Three types are dedicated to the control of external coprocessors. These enable you to extend the functionality of the instruction set off-chip in an open and uniform way.

The ARM instruction set is a good target for compilers of many different high-level languages. Where required for critical code segments, assembly code programming is also straightforward, unlike some RISC processors that depend on sophisticated compiler technology to manage complicated instruction interdependencies.

1.3.1 Format summary

This section provides a summary of the ARM and Thumb instruction sets:

- *ARM instruction set* on page 1-9
- *Thumb instruction set* on page 1-18.

A key to the instruction set tables is shown in Table 1-1.

The ARM7TDMI-S core on the ARM720T macrocell is an implementation of the ARM architecture v4T. For a complete description of both instruction sets, see the *ARM Architecture Reference Manual*.

Table 1-1 Key to tables

Entry	Description
{cond}	Refer to Table 1-11 on page 1-18.
<Oprnd2>	Refer to Table 1-9 on page 1-17.
{field}	Refer to Table 1-10 on page 1-17.

Table 1-1 Key to tables (continued)

Entry	Description
S	Sets condition codes (optional).
B	Byte operation (optional).
H	Halfword operation (optional).
T	Forces address translation. Cannot be used with pre-indexed addresses.
<a_mode2>	Refer to Table 1-3 on page 1-13.
<a_mode2P>	Refer to Table 1-4 on page 1-14.
<a_mode3>	Refer to Table 1-5 on page 1-15.
<a_mode4L>	Refer to Table 1-6 on page 1-16.
<a_mode4S>	Refer to Table 1-7 on page 1-16.
<a_mode5>	Refer to Table 1-8 on page 1-16.
#32bit_Imm	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits.
<reglist>	A comma-separated list of registers, enclosed in braces ({ and }).

1.3.2 ARM instruction set

This section gives an overview of the ARM instructions available. For full details of these instructions, refer to the *ARM Architecture Reference Manual*.

The ARM instruction set formats are shown in Figure 1-3 on page 1-10.

	31 30 29 28	27	26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 09 08	07 06 05	04	03 02 01 00
Data processing immediate	cond	0	0	1	op	S	Rn	Rd	rotate	immediate		
Data processing immediate shift	cond	0	0	0	opcode	S	Rn	Rd	shift immediate	shift	0	Rm
Data processing register shift	cond	0	0	0	opcode	S	Rn	Rd	Rs	0	shift	1	Rm
Multiply	cond	0	0	0	0 0 0 A	S	Rd	Rn	Rs	1 0 0	1	Rm
Multiply long	cond	0	0	0	0 1 U A	S	RdHi	RdLo	Rn	1 0 0	1	Rm
Move from status register	cond	0	0	0	1 0 R 0	0	SBO	Rd	SBZ			
Move immediate to status register	cond	0	0	1	1 0 R 1	0	Mask	SBO	rotate	immediate		
Move register to status register	cond	0	0	0	1 0 R 1	0	Mask	SBO	SBZ		0	Rm
Branch/exchange instruction set	cond	0	0	0	1 0 0 1	0	SBO	SBO	SBO	0 0 0	1	Rm
Load/store immediate offset	cond	0	1	0	P U B W	L	Rn	Rd	immediate			
Load/store register offset	cond	0	1	1	P U B W	L	Rn	Rd	shift immediate	shift	0	Rm
Load/store halfword/ signed byte	cond	0	0	0	P U 1 W	L	Rn	Rd	High offset	1 S H	1	Low offset
Load/store halfword/ signed byte	cond	0	0	0	P U 0 W	L	Rn	Rd	SBZ	1 S H	1	Rm
Swap/swap byte	cond	0	0	0	1 0 B 0	0	Rn	Rd	SBZ	1 0 0	1	Rm
Load/store multiple	cond	1	0	0	P U S W	L	Rn	Register list				
Coprocessor data processing	cond	1	1	1	0 op1		CRn	CRd	cp_num	op2	0	CRm
Coprocessor register transfers	cond	1	1	1	0 op1	L	CRn	Rd	cp_num	op2	1	CRm
Coprocessor load and store	cond	1	1	0	P U N W	L	Rn	CRd	cp_num	8_bit_offset		
Branch and branch with link	cond	1	0	1	L	24_bit_offset						
Software interrupt	cond	1	1	1	1	swi_number						
Undefined	cond	0	1	1	x x x x	x	x x x x	x x x x	x x x x	x x x	1	x x x x

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Figure 1-3 ARM instruction set formats

——— **Note** ———

Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for example, a multiply instruction with bit 6 set. You must not use these instructions, as their action might change in future ARM implementations.

The ARM instruction set summary is shown in Table 1-2.

Table 1-2 ARM instruction summary

Operation		Assembler
Move	Move	`MOV{cond}{S} Rd, <Oprnd2>`
	Move NOT	`MVN{cond}{S} Rd, <Oprnd2>`
	Move SPSR to register	`MRS{cond} Rd, SPSR`
	Move CPSR to register	`MRS{cond} Rd, CPSR`
	Move register to SPSR	`MSR{cond} SPSR{field}, Rm`
	Move register to CPSR	`MSR{cond} CPSR{field}, Rm`
	Move immediate to SPSR flags	`MSR{cond} SPSR_f, #32bit_Imm`
	Move immediate to CPSR flags	`MSR{cond} CPSR_f, #32bit_Imm`
Arithmetic	Add	`ADD{cond}{S} Rd, Rn, <Oprnd2>`
	Add with carry	`ADC{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract	`SUB{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract with carry	`SBC{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract reverse subtract	`RSB{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract reverse subtract with carry	`RSC{cond}{S} Rd, Rn, <Oprnd2>`
	Multiply	`MUL{cond}{S} Rd, Rm, Rs`
	Multiply accumulate	`MLA{cond}{S} Rd, Rm, Rs, Rn`
	Multiply unsigned long	`UMULL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply unsigned accumulate long	`UMLAL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply signed long	`SMULL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply signed accumulate long	`SMLAL{cond}{S} RdLo, RdHi, Rm, Rs`
	Compare	`CMP{cond} Rd, <Oprnd2>`
	Compare negative	`CMN{cond} Rd, <Oprnd2>`
Logical	Test	`TST{cond} Rn, <Oprnd2>`
	Test equivalence	`TEQ{cond} Rn, <Oprnd2>`

Table 1-2 ARM instruction summary (continued)

Operation		Assembler
	AND	`AND{cond}{S} Rd, Rn, <Oprnd2>`
	EOR	`EOR{cond}{S} Rd, Rn, <Oprnd2>`
	ORR	`ORR{cond}{S} Rd, Rn, <Oprnd2>`
	Bit clear	`BIC{cond}{S} Rd, Rn, <Oprnd2>`
Branch	Branch	`B{cond} label`
	Branch with link	`BL{cond} label`
	Branch, and exchange instruction set	`BX{cond} Rn`
Load	Word	`LDR{cond} Rd, <a_mode2>`
	Word with User Mode privilege	`LDR{cond}T Rd, <a_mode2P>`
	Byte	`LDR{cond}B Rd, <a_mode2>`
	Byte with User Mode privilege	`LDR{cond}BT Rd, <a_mode2P>`
	Byte signed	`LDR{cond}SB Rd, <a_mode3>`
	Halfword	`LDR{cond}H Rd, <a_mode3>`
	Halfword signed	`LDR{cond}SH Rd, <a_mode3>`
Multiple block data operations	Increment before	`LDM{cond}IB Rd{!}, <reglist>{^}`
	Increment after	`LDM{cond}IA Rd{!}, <reglist>{^}`
	Decrement before	`LDM{cond}DB Rd{!}, <reglist>{^}`
	Decrement after	`LDM{cond}DA Rd{!}, <reglist>{^}`
	Stack operations	`LDM{cond}<a_mode4L> Rd{!}, <reglist>`
	Stack operations, and restore CPSR	`LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^`
	User registers	`LDM{cond}<a_mode4L> Rd{!}, <reglist>^`
Store	Word	`STR{cond} Rd, <a_mode2>`
	Word with User Mode privilege	`STR{cond}T Rd, <a_mode2P>`
	Byte	`STR{cond}B Rd, <a_mode2>`

Table 1-2 ARM instruction summary (continued)

Operation		Assembler
	Byte with User Mode privilege	`STR{cond}BT Rd, <a_mode2P>`
	Halfword	`STR{cond}H Rd, <a_mode3>`
Multiple block data operations	Increment before	`STM{cond}IB Rd{!}, <reglist>{^}`
	Increment after	`STM{cond}IA Rd{!}, <reglist>{^}`
	Decrement before	`STM{cond}DB Rd{!}, <reglist>{^}`
	Decrement after	`STM{cond}DA Rd{!}, <reglist>{^}`
	Stack operations	`STM{cond}<a_mode4S> Rd{!}, <reglist>`
	User registers	`STM{cond}<a_mode4S> Rd{!}, <reglist>^`
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`
	Byte	`SWP{cond}B Rd, Rm, [Rn]`
Coprocessors	Data operations	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`
	Move to ARM reg from coproc	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Move to coproc from ARM reg	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode5>`
	Store	`STC{cond} p<cpnum>, CRd, <a_mode5>`
Software Interrupt		`SWI 24bit_Imm`

Addressing mode 2, <a_mode2>, is shown in Table 1-3.

Table 1-3 Addressing mode 2

Operation	Assembler
Immediate offset	`[Rn, #+/-12bit_Offset]`
Register offset	`[Rn, +/-Rm]`
Scaled register offset	`[Rn, +/-Rm, LSL #5bit_shift_imm]`
	`[Rn, +/-Rm, LSR #5bit_shift_imm]`

Table 1-3 Addressing mode 2 (continued)

Operation	Assembler
	`[Rn, +/-Rm, ASR #5bit_shift_imm]`
	`[Rn, +/-Rm, ROR #5bit_shift_imm]`
	`[Rn, +/-Rm, RRX]`
Pre-indexed immediate offset	`[Rn, #+/-12bit_Offset]!`
Pre-indexed register offset	`[Rn, +/-Rm]!`
Pre-indexed scaled register offset	`[Rn, +/-Rm, LSL #5bit_shift_imm]!`
	`[Rn, +/-Rm, LSR #5bit_shift_imm]!`
	`[Rn, +/-Rm, ASR #5bit_shift_imm]!`
	`[Rn, +/-Rm, ROR #5bit_shift_imm]!`
	`[Rn, +/-Rm, RRX]!`
Post-indexed immediate offset	`[Rn], #+/-12bit_Offset`
Post-indexed register offset	`[Rn], +/-Rm`
Post-indexed scaled register offset	`[Rn], +/-Rm, LSL #5bit_shift_imm`
	`[Rn], +/-Rm, LSR #5bit_shift_imm`
	`[Rn], +/-Rm, ASR #5bit_shift_imm`
	`[Rn], +/-Rm, ROR #5bit_shift_imm`
	`[Rn, +/-Rm, RRX]`

Addressing mode 2 (privileged), `<a_mode2P>`, is shown in Table 1-4.

Table 1-4 Addressing mode 2 (privileged)

Operation	Assembler
Immediate offset	`[Rn, #+/-12bit_Offset]`
Register offset	`[Rn, +/-Rm]`
Scaled register offset	`[Rn, +/-Rm, LSL #5bit_shift_imm]`
	`[Rn, +/-Rm, LSR #5bit_shift_imm]`

Table 1-4 Addressing mode 2 (privileged) (continued)

Operation	Assembler
	`[Rn, +/-Rm, ASR #5bit_shift_imm]`
	`[Rn, +/-Rm, ROR #5bit_shift_imm]`
	`[Rn, +/-Rm, RRX]`
Post-indexed immediate offset	`[Rn], #+/-12bit_Offset`
Post-indexed register offset	`[Rn], +/-Rm`
Post-indexed scaled register offset	`[Rn], +/-Rm, LSL #5bit_shift_imm`
	`[Rn], +/-Rm, LSR #5bit_shift_imm`
	`[Rn], +/-Rm, ASR #5bit_shift_imm`
	`[Rn], +/-Rm, ROR #5bit_shift_imm`
	`[Rn, +/-Rm, RRX]`

Addressing mode 3 (signed byte, and halfword data transfer), `<a_mode3>`, is shown in Table 1-5.

Table 1-5 Addressing mode 3

Operation	Assembler
Immediate offset	`[Rn, #+/-8bit_Offset]`
Pre-indexed	`[Rn, #+/-8bit_Offset]!`
Post-indexed	`[Rn], #+/-8bit_Offset`
Register	`[Rn, +/-Rm]`
Pre-indexed	`[Rn, +/-Rm]!`
Post-indexed	`[Rn], +/-Rm`

Addressing mode 4 (load), <a_mode4L>, is shown in Table 1-6.

Table 1-6 Addressing mode 4 (load)

Addressing mode		Stack type	
IA	Increment after	FD	Full descending
IB	Increment before	ED	Empty descending
DA	Decrement after	FA	Full ascending
DB	Decrement before	EA	Empty ascending

Addressing mode 4 (store), <a_mode4S>, is shown in Table 1-7.

Table 1-7 Addressing mode 4 (store)

Addressing mode		Stack type	
IA	Increment after	EA	Empty ascending
IB	Increment before	FA	Full ascending
DA	Decrement after	ED	Empty descending
DB	Decrement before	FD	Full descending

Addressing mode 5 (coprocessor data transfer), <a_mode5>, is shown in Table 1-8.

Table 1-8 Addressing mode 5

Operation	Assembler
Immediate offset	[Rn, #+/-(8bit_Offset*4)]
Pre-indexed	[Rn, #+/-(8bit_Offset*4)]!
Post-indexed	[Rn], #+/-(8bit_Offset*4)

Operand 2, <0prnd2>, is shown in Table 1-9.

Table 1-9 Operand 2

Operation	Assembler
Immediate value	`#32bit_Imm`
Logical shift left	`Rm LSL #5bit_Imm`
Logical shift right	`Rm LSR #5bit_Imm`
Arithmetic shift right	`Rm ASR #5bit_Imm`
Rotate right	`Rm ROR #5bit_Imm`
Register	`Rm`
Logical shift left	`Rm LSL Rs`
Logical shift right	`Rm LSR Rs`
Arithmetic shift right	`Rm ASR Rs`
Rotate right	`Rm ROR Rs`
Rotate right extended	`Rm RRX`

Fields, {field}, are shown in Table 1-10.

Table 1-10 Fields

Suffix	Sets
_c	Control field mask bit (bit 3)
_f	Flags field mask bit (bit 0)
_s	Status field mask bit (bit 1)
_x	Extension field mask bit (bit 2)

Condition fields, {cond}, are shown in Table 1-11.

Table 1-11 Condition fields

Suffix	Description	Condition(s)
EQ	Equal	Z set
NE	Not equal	Z clear
CS	Unsigned higher, or same	C set
CC	Unsigned lower	C clear
MI	Negative	N set
PL	Positive, or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set, Z clear
LS	Unsigned lower, or same	C clear, Z set
GE	Greater, or equal	N=V (N and V set or N and V clear)
LT	Less than	N<>V (N set and V clear) or (N clear and V set)
GT	Greater than	Z clear, N=V (N and V set or N and V clear)
LE	Less than, or equal	Z set or N<>V (N set and V clear) or (N clear and V set)
AL	Always	Always

1.3.3 Thumb instruction set

This section gives an overview of the Thumb instructions available. For full details of these instructions, see the *ARM Architecture Reference Manual*.

The Thumb instruction set formats are shown in Figure 1-4 on page 1-19.

		15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
Move shifted register	01	0	0	0	Op		Offset5					Rs			Rd		
Add and subtract	02	0	0	0	1	1	1	Op	Rn/ offset3			Rs			Rd		
Move, compare, add, and subtract immediate	03	0	0	1	Op		Rd			Offset8							
ALU operation	04	0	1	0	0	0	0	Op				Rs			Rd		
High register operations and branch exchange	05	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			RdHd		
PC-relative load	06	0	1	0	0	1	Rd			Word8							
Load and store with relative offset	07	0	1	0	1	L	B	0	Ro			Rb			Rd		
Load and store sign-extended byte and halfword	08	0	1	0	1	H	S	1	Ro			Rb			Rd		
Load and store with immediate offset	09	0	1	1	B	L	Offset5					Rb			Rd		
Load and store halfword	10	1	0	0	0	L	Offset5					Rb			Rd		
SP-relative load and store	11	1	0	0	1	L	Rd			Word8							
Load address	12	1	0	1	0	SP	Rd			Word8							
Add offset to stack pointer	13	1	0	1	1	0	0	0	0	S	SWord7						
Push and pop registers	14	1	0	1	1	L	1	0	R	Rlist							
Multiple load and store	15	1	1	0	0	L	Rb			Rlist							
Conditional branch	16	1	1	0	1	Cond				Softset8							
Software interrupt	17	1	1	0	1	1	1	1	1	Value8							
Unconditional branch	18	1	1	1	0	0	Offset11										
Long branch with link	19	1	1	1	1	H	Offset										

15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Figure 1-4 Thumb instruction set formats

The Thumb instruction set summary is shown in Table 1-12.

Table 1-12 Thumb instruction summary

Operation		Assembler
Move	Immediate	`MOV Rd, #8bit_Imm`
	High to Low	`MOV Rd, Hs`
	Low to High	`MOV Hd, Rs`
	High to High	`MOV Hd, Hs`
Arithmetic	Add	`ADD Rd, Rs, #3bit_Imm`
	Add Low, and Low	`ADD Rd, Rs, Rn`
	Add High to Low	`ADD Rd, Hs`
	Add Low to High	`ADD Hd, Rs`
	Add High to High	`ADD Hd, Hs`
	Add Immediate	`ADD Rd, #8bit_Imm`
	Add Value to SP	`ADD SP, #7bit_Imm ADD SP, #-7bit_Imm`
	Add with carry	`ADC Rd, Rs`
	Subtract	`SUB Rd, Rs, Rn SUB Rd, Rs, #3bit_Imm`
	Subtract Immediate	`SUB Rd, #8bit_Imm`
	Subtract with carry	`SBC Rd, Rs`
	Negate	`NEG Rd, Rs`
	Multiply	`MUL Rd, Rs`
	Compare Low, and Low	`CMP Rd, Rs`
	Compare Low, and High	`CMP Rd, Hs`
	Compare High, and Low	`CMP Hd, Rs`
	Compare High, and High	`CMP Hd, Hs`
	Compare Negative	`CMN Rd, Rs`
	Compare Immediate	`CMP Rd, #8bit_Imm`
Logical	AND	`AND Rd, Rs`

Table 1-12 Thumb instruction summary (continued)

Operation			Assembler
		EOR	`EOR Rd, Rs`
		OR	`ORR Rd, Rs`
		Bit clear	`BIC Rd, Rs`
		Move NOT	`MVN Rd, Rs`
		Test bits	`TST Rd, Rs`
Shift/Rotate		Logical shift left	`LSL Rd, Rs, #5bit_shift_imm LSL Rd, Rs`
		Logical shift right	`LSR Rd, Rs, #5bit_shift_imm LSR Rd, Rs`
		Arithmetic shift right	`ASR Rd, Rs, #5bit_shift_imm ASR Rd, Rs`
		Rotate right	`ROR Rd, Rs`
Branch	Conditional		
		if Z set	`BEQ label`
		if Z clear	`BNE label`
		if C set	`BCS label`
		if C clear	`BCC label`
		if N set	`BMI label`
		if N clear	`BPL label`
		if V set	`BVS label`
		if V clear	`BVC label`
		if C set, and Z clear	`BHI label`
		if C clear, and Z set	`BLS label`
		if N set, and V set, or if N clear, and V clear	`BGE label`
		if N set, and V clear, or if N clear, and V set	`BLT label`
		if Z clear, and N, or V set, or if Z clear, and N, or V clear	`BGT label`

Table 1-12 Thumb instruction summary (continued)

Operation			Assembler
		if Z set, or N set, and V clear, or N clear, and V set	`BLE label`
	Unconditional		`B label`
		Long branch with link	`BL label`
		Optional state change	
		to address held in Lo reg	`BX Rs`
		to address held in Hi reg	`BX Hs`
Load	With immediate offset		
		word	`LDR Rd, [Rb, #7bit_offset]`
		halfword	`LDRH Rd, [Rb, #6bit_offset]`
		byte	`LDRB Rd, [Rb, #5bit_offset]`
	With register offset		
		word	`LDR Rd, [Rb, Ro]`
		halfword	`LDRH Rd, [Rb, Ro]`
		signed halfword	`LDRSH Rd, [Rb, Ro]`
		byte	`LDRB Rd, [Rb, Ro]`
		signed byte	`LDRSB Rd, [Rb, Ro]`
		PC-relative	`LDR Rd, [PC, #10bit_Offset]`
		SP-relative	`LDR Rd, [SP, #10bit_Offset]`
	Address		
		using PC	`ADD Rd, PC, #10bit_Offset`
		using SP	`ADD Rd, SP, #10bit_Offset`
		Multiple	`LDMIA Rb!, <reglist>`
Store	With immediate offset		
		word	`STR Rd, [Rb, #7bit_offset]`

Table 1-12 Thumb instruction summary (continued)

Operation			Assembler
		halfword	STRH Rd, [Rb, #6bit_offset]
		byte	STRB Rd, [Rb, #5bit_offset]
	With register offset		
		word	STR Rd, [Rb, Ro]
		halfword	STRH Rd, [Rb, Ro]
		byte	STRB Rd, [Rb, Ro]
	SP-relative		STR Rd, [SP, #10bit_offset]
	Multiple		STMIA Rb!, <reglist>
Push/Pop		Push registers onto stack	PUSH <reglist>
		Push LR, and registers onto stack	PUSH <reglist, LR>
		Pop registers from stack	POP <reglist>
		Pop registers, and PC from stack	POP <reglist, PC>
Software Interrupt			SWI 8bit_Imm

Chapter 2
Programmer's Model

This chapter describes the programmer's model for the ARM720T processor . It contains the following sections:

- *Processor operating states* on page 2-2
- *Memory formats* on page 2-3
- *Instruction length* on page 2-5
- *Data types* on page 2-6
- *Operating modes* on page 2-7
- *Registers* on page 2-8
- *Program status registers* on page 2-13
- *Exceptions* on page 2-16
- *Relocation of low virtual addresses by the FCSE PID* on page 2-23
- *Reset* on page 2-24
- *Implementation-defined behavior of instructions* on page 2-25.

Copyright © 2001 ARM Limited. All rights reserved.

2.1   Processor operating states

From the point of view of the programmer, the ARM720T processor can be in one of two states:

ARM state   This executes 32-bit, word-aligned ARM instructions.

Thumb state   This operates with 16-bit, halfword-aligned Thumb instructions. In this state, the PC uses bit 1 to select between alternate halfwords.

2.1.1   Switching between processor states

Transition between processor states does not affect the processor mode or the contents of the registers.

Entering Thumb state

Entry into Thumb state can be achieved by executing a BX instruction with the state bit (bit 0) set in the operand register.

Transition to Thumb state also occurs automatically on return from an exception, for example, *Interrupt ReQuest* (IRQ), *Fast Interrupt reQuest* (FIQ), UNDEF, ABORT, and *SoftWare Interrupt* (SWI) if the exception was entered with the processor in Thumb state.

Entering ARM state

Entry into ARM state happens:

*   On execution of the BX instruction with the state bit clear in the operand register.

*   On the processor taking an exception, for example, IRQ, FIQ, RESET, UNDEF, ABORT, and SWI. In this case, the PC is placed in the link register of the exception mode, and execution starts at the vector address of the exception.

2.2 Memory formats

The ARM720T processor views memory as a linear collection of bytes numbered upwards from zero, as follows:

Bytes 0 to 3   Hold the first stored word.

Bytes 4 to 7   Hold the second stored word.

Bytes 8 to 11   Hold the third stored word.

Words are stored in memory as big or little-endian, as described in the following sections:
* *Big-endian format*
* *Little-endian format* on page 2-4.

The endianness used depends on the status of the big-end bit in the control register of the system control coprocessor. See *Register 1, control register* on page 3-5 for more information.

2.2.1 Big-endian format

In big-endian format, the most significant byte of a word is stored at the lowest numbered byte and the least significant byte at the highest numbered byte. Byte 0 of the memory system is therefore connected to data lines 31 to 24.

Big-endian format is shown in Figure 2-1.

31 24	23 16	15 8	7 0	Word address
8	9	10	11	8
4	5	6	7	4
0	1	2	3	0

Higher address ↑ Lower address
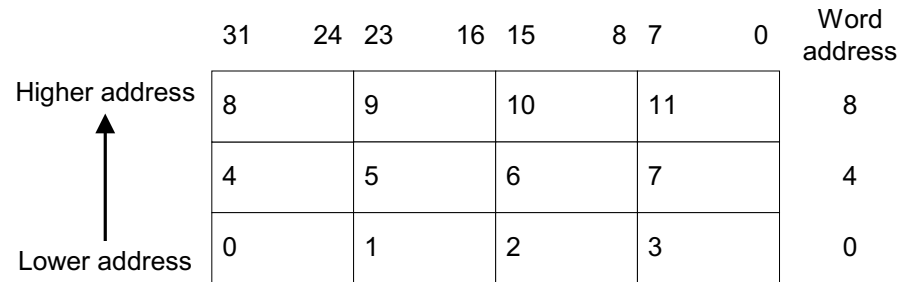
Most significant byte is at lowest address

Word is addressed by byte address of most significant byte

Figure 2-1 Big-endian addresses of bytes with words

2.2.2   Little-endian format

In little-endian format, the lowest numbered byte in a word is considered the least significant byte of the word, and the highest numbered byte the most significant. Byte 0 of the memory system is therefore connected to data lines 7 to 0.

Little-endian format is shown in Figure 2-2.

	31 24	23 16	15 8	7 0	Word address
Higher address	11	10	9	8	8
	7	6	5	4	4
Lower address	3	2	1	0	0
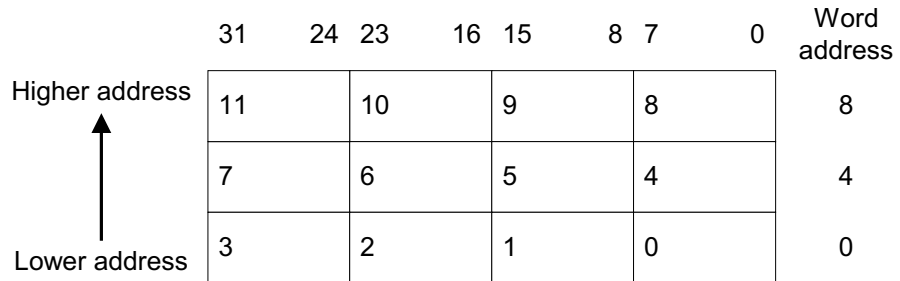
Least significant byte is at lowest address

Word is addressed by byte address of least significant byte

Figure 2-2 Little-endian addresses of bytes with words

2.3   Instruction length

Instructions are:

- 32 bits long in ARM state
- 16 bits long in Thumb state.

2.4   Data types

The ARM720T processor supports the following data types:

- byte (8-bit)
- halfword (16-bit)
- word (32-bit).

You must align these as follows:

- word quantities to 4-byte boundaries
- halfwords quantities to 2-byte boundaries
- byte quantities can be placed on any byte boundary.

       ARM DDI 0229A

2.5 Operating modes

The ARM720T processor supports seven modes of operation, as shown in Table 2-1.

Table 2-1 ARM720T modes of operation

Mode	Type	Description
User	usr	The normal ARM program execution mode
FIQ	fiq	Used for most performance-critical interrupts in a system
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	Protected mode for the operating system
Abort mode	abt	Entered after a Data Abort or instruction Prefetch Abort
System	sys	A privileged User mode for the operating system
Undefined	und	Entered when an Undefined Instruction is executed

2.5.1 Changing operating modes

Mode changes can be made under software control, by external interrupts or during exception processing. Most application programs execute in User mode. The non-User modes, known as privileged modes, are entered in order to service interrupts or exceptions, or to access protected resources.

2.6   Registers

The ARM720T processor has a total of 37 registers:

- 31 general-purpose 32-bit registers
- six program status registers.

These registers cannot all be seen at once. The processor state and operating mode dictate which registers are available to the programmer.

2.6.1   The ARM state register set

In ARM state, 16 general registers and one or two status registers are visible at any one time. In privileged (non-User) modes, mode-specific banked registers are switched in. Figure 2-3 on page 2-9 shows which registers are available in each mode. The banked registers are marked with a shaded triangle.

The ARM state register set contains 16 directly accessible registers, R0 to R15. All of these, except R15, are general-purpose, and can be used to hold either data or address values. Registers R14 and R15 also have special roles, as follows:

Register 14   R14 is used as the subroutine link register. This receives a copy of R15 when a *Branch and Link* (BL) code instruction is executed. At all other times it can be treated as a general-purpose register. The corresponding banked registers R14_svc, R14_irq, R14_fiq, R14_abt, and R14_und are similarly used to hold the return values of R15 when interrupts and exceptions arise, or when BL instructions are executed within interrupt or exception routines.

Register 15   R15 holds the *Program Counter* (PC). In ARM state, bits [1:0] of R15 are zero and bits [31:2] contain the PC. In Thumb state, bit 0 is zero and bits [31:1] contain the PC.

In addition to these, the *Current Program Status Register* (CPSR) is used to store status information. It contains condition code flags and the current mode bits.

Interrupt modes

FIQ mode has seven banked registers mapped to R8-14 (R8_fiq-R14_fiq). In ARM state, many FIQ handlers can use these banked registers, to avoid having to save any registers onto a stack. User, IRQ, Supervisor, Abort, and Undefined modes each have two banked registers mapped to R13 and R14, allowing each of these modes to have a private stack pointer and link registers.

ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13	r13_fiq	r13_svc	r13_abt	r13_irq	r13_und
r14	r14_fiq	r14_svc	r14_abt	r14_irq	r14_und
r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

◺ *= banked register*

Figure 2-3 Register organization in ARM state

2.6.2 The Thumb state register set

The Thumb state register set is a subset of the ARM state set. You have direct access to:

- eight general registers, (R0–R7)
- the PC
- a *Stack Pointer* (SP) *register*
- a *Link Register* (LR)
- the CPSR.

There are banked SPs, LRs, and *Saved Process Status Registers* (SPSRs) for each privileged mode. This is shown in Figure 2-4.

Thumb state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
SP	SP_fiq	SP_svc	SP_abt	SP_irq	SP_und
LR	LR_fiq	LR_svc	LR_abt	LR_irq	LR_und
PC	PC	PC	PC	PC	PC

Thumb state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

= banked register

Figure 2-4 Register organization in Thumb state

2.6.3 The relationship between ARM and Thumb state registers

The Thumb state registers relate to the ARM state registers in the following ways:

- Thumb state R0–R7, and ARM state R0–R7 are identical
- Thumb state CPSR and SPSRs, and ARM state CPSR and SPSRs are identical
- Thumb state SP maps onto ARM state R13
- Thumb state LR maps onto ARM state R14
- Thumb state PC maps onto ARM state PC (R15).

This relationship is shown in Figure 2-5.
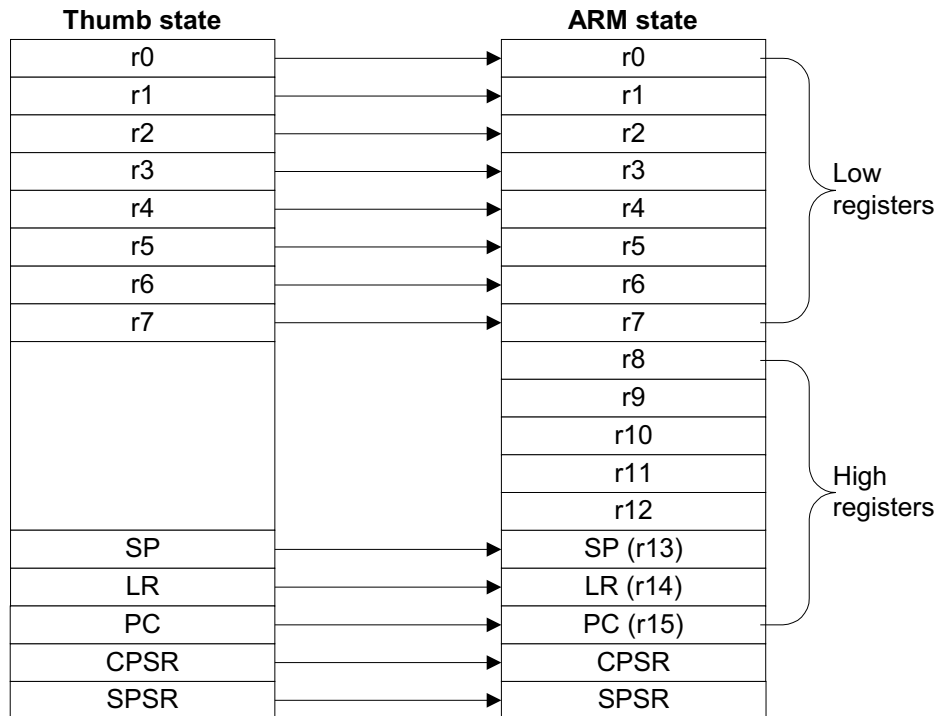
Figure 2-5 Mapping of Thumb state registers onto ARM state registers

2.6.4 Accessing high registers in Thumb state

In Thumb state, ARM registers R8–R15 (the high registers) are not part of the standard register set. However, the assembly language programmer has limited access to them, and can use them for fast temporary storage.

A value can be transferred from a register in the range R0 – R7 (a low register) to a high register, and from a high register to a low register, using special variants of the MOV instruction. High register values can also be compared against or added to low register values with the CMP and ADD instructions. See the *ARM Architecture Reference Manual* for details on high register operations.

2.7   Program status registers

The ARM720T processor contains a CPSR, and five SPSRs for use by exception handlers. These registers:

- hold information about the most recently performed ALU operation
- control the enabling and disabling of interrupts
- set the processor operating mode.

The arrangement of bits is shown in Figure 2-6.
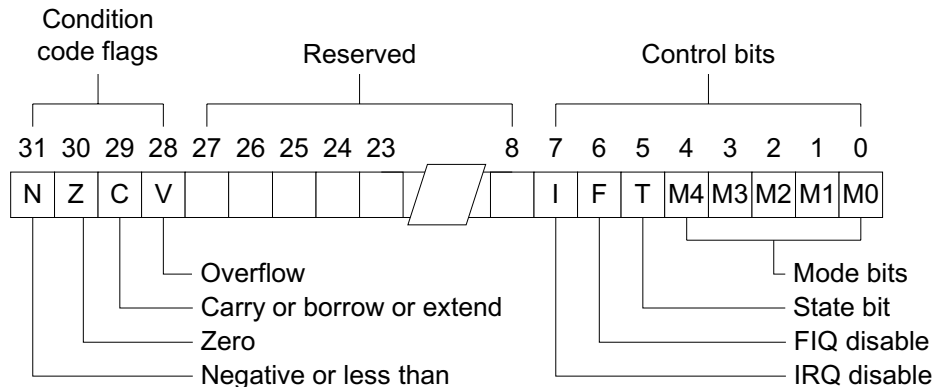
Figure 2-6 Program status register format

2.7.1   The condition code flags

The N, Z, C, and V bits are the condition code flags. These can be changed as a result of arithmetic and logical operations, and tested to determine whether an instruction executes.

In ARM state, all instructions can be executed conditionally. In Thumb state, only the Branch instruction is capable of conditional execution. See the *ARM Architecture Reference Manual* for details.

2.7.2   The control bits

The bottom eight bits of a PSR (incorporating I, F, T, and M[4:0]) are known collectively as the control bits. These change when an exception arises. If the processor is operating in a privileged mode, they can also be manipulated by software:

I and F bits   These are the interrupt disable bits. When set, these disable the IRQ and FIQ interrupts respectively.

The T bit   This reflects the operating state. When this bit is set, the processor is executing in Thumb state, otherwise it is executing in ARM state. This is reflected on the **CPTBIT** external signal. Software must never change the state of the **CPTBIT** in the CPSR. If this happens, the processor then enters an unpredictable state.

M[4:0] bits   These are the mode bits. These determine the processor operating mode, as shown in Table 2-2 on page 2-15. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described can be used.

——— **Note** ———

If you program any illegal value into the mode bits, M[4:0], then the processor enters an unrecoverable state. If this occurs, apply reset.

2.7.3 Reserved bits

The remaining bits in the PSRs are reserved. When changing flag or control bits of a PSR, you must ensure that these unused bits are not altered. Also, your program must not rely on them containing specific values, because in future processors they might read as one or zero.

Table 2-2 PSR mode bit values

M[4:0]	Mode	Visible Thumb state registers	Visible ARM state registers
10000	User	R7 to R0, LR, SP PC, CPSR	R14 to R0, PC, CPSR
10001	FIQ	R7 to R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7 to R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7 to R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12 to R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7 to R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12 to R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7 to R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12 to R0, R14_abt..R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7 to R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12 to R0, R14_und, R13_und, PC, CPSR, SPSR_und
11111	System	R7 to R0, LR, SP PC, CPSR	R14 to R0, PC, CPSR

2.8 Exceptions

Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example to service an interrupt from a peripheral. Before an exception can be handled, the current processor state must be preserved so that the original program can resume when the handler routine has finished.

Several exceptions can arise at the same time. If this happens, they are dealt with in a fixed order. See *Exception priorities* on page 2-21.

Exception behavior is described in the following sections:
* *Action on entering an exception*
* *Action on leaving an exception* on page 2-17
* *Exception entry and exit summary* on page 2-17
* *Fast interrupt request* on page 2-18
* *Interrupt request* on page 2-19
* *Abort* on page 2-19
* *Software interrupt* on page 2-20
* *Undefined instruction* on page 2-20
* *Exception vectors* on page 2-21
* *Exception priorities* on page 2-21
* *Exception restrictions* on page 2-22.

2.8.1 Action on entering an exception

When handling an exception, the ARM720T processor behaves as follows:

1. It preserves the address of the next instruction in the appropriate LR.

   a. If the exception has been entered from ARM state, the address of the next instruction is copied into the LR (that is, current PC+4 or PC+8 depending on the exception). See Table 2-3 on page 2-17 for details).

   b. If the exception has been entered from Thumb state, the value written into the LR is the current PC, offset by a value so that the program resumes from the correct place on return from the exception. This means that the exception handler does not have to determine which state the exception was entered from.

   For example, in the case of SWI:

   ```
   MOVS PC, R14_svc
   ```

   always returns to the next instruction regardless of whether the SWI was executed in ARM or Thumb state.

2. It copies the CPSR into the appropriate SPSR.

3.   It forces the CPSR mode bits to a value that depends on the exception.

4.   It forces the PC to fetch the next instruction from the relevant exception vector.

It can also set the interrupt disable flags to prevent otherwise unmanageable nestings of exceptions.

If the processor is in Thumb state when an exception occurs, it automatically switches into ARM state when the PC is loaded with the exception vector address.

2.8.2   Action on leaving an exception

On completion, the exception handler:

1.   Moves the LR, minus an offset where appropriate, to the PC. The offset varies depending on the type of exception.

2.   Copies the SPSR back to the CPSR.

3.   Clears the interrupt disable flags, if they were set on entry.

——— **Note** ———

An explicit switch back to Thumb state is never necessary, because restoring the CPSR from the SPSR automatically sets the T bit to the value it held immediately prior to the exception.

2.8.3   Exception entry and exit summary

Table 2-3 summarizes the PC value preserved in the relevant R14 on exception entry, and the recommended instruction for exiting the exception handler.

Table 2-3 Exception entry and exit

Exception	Return instruction	Previous state	
		ARM R14_x	**Thumb R14_x**
BL[a]	`MOV PC, R14`	PC + 4	PC + 2
SWI[a]	`MOVS PC, R14_svc`	PC + 4	PC + 2
UDEF[a]	`MOVS PC, R14_und`	PC + 4	PC + 2
FIQ[b]	`SUBS PC, R14_fiq, #4`	PC + 4	PC + 4
IRQ[b]	`SUBS PC, R14_irq, #4`	PC + 4	PC + 4

Exception	Return instruction	Previous state	
		ARM R14_x	**Thumb R14_x**
PABT[a]	SUBS PC, R14_abt, #4	PC + 4	PC + 4
DABT[c]	SUBS PC, R14_abt, #8	PC + 8	PC + 8
RESET[d]	NA	-	-

a. PC is the address of the BL, SWI, Undefined Instruction, or Fetch, that had the Prefetch Abort.

b. PC is the address of the instruction that was not executed because the FIQ or IRQ took priority.

c. PC is the address of the Load or Store instruction that generated the Data Abort.

d. The value saved in R14_svc upon reset is unpredictable.

2.8.4   Fast interrupt request

The FIQ exception is used for most performance-critical interrupts in a system. In ARM state it has sufficient private registers to remove the necessity for register saving, minimizing the overhead of context switching.

FIQ is externally generated by taking the **nFIQ** input LOW. **nFIQ** and **nIRQ** are considered asynchronous, and a cycle delay for synchronization is incurred before the interrupt can affect the processor flow.

Irrespective of whether the exception was entered from ARM or Thumb state, a FIQ handler must leave the interrupt by executing:

SUBS PC, R14_fiq, #4

FIQ can be disabled by setting the F flag in the CPSR.

——— **Note** ———

This is not possible from User mode.

If the F flag is clear, the ARM720T processor checks for a LOW level on the output of the FIQ synchronizer at the end of each instruction.

2.8.5   Interrupt request

The IRQ exception is a normal interrupt caused by a LOW level on the **nIRQ** input. IRQ has a lower priority than FIQ and is masked out when a FIQ sequence is entered. It can be disabled at any time by setting the I bit in the CPSR, though this can only be done from a privileged (non-User) mode.

Irrespective of whether the exception was entered from ARM or Thumb state, an IRQ handler must return from the interrupt by executing:

```
SUBS PC, R14_irq, #4
```

2.8.6   Abort

An abort indicates that the current memory access cannot be completed. It can be signaled either by the protection unit, or by the **HRESP** bus. The ARM720T core checks for the abort exception during memory access cycles.

There are two types of abort, as follows:

Prefetch Abort   This occurs during an instruction prefetch. The prefetched instruction is marked as invalid, but the exception is not taken until the instruction reaches the head of the pipeline. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, the abort does not take place.

Data Abort   This occurs during a data access. The action taken depends on the instruction type:

- Single data transfer instructions (LDR, STR) write-back modified base registers. The Abort handler must be aware of this.

- The swap instruction (SWP) is aborted as though it had not been executed.

- Block data transfer instructions (LDM, STM) complete. If write-back is set, the base is updated. If the instruction attempts to overwrite the base with data (that is, it has the base in the transfer list), the overwriting is prevented. All register overwriting is prevented after an abort is indicated. This means, in particular, that R15 (always the last register to be transferred) is preserved in an aborted LDM instruction.

After fixing the reason for the abort, the handler must execute the following irrespective of the processor state (ARM or Thumb):

```
SUBS PC, R14_abt, #4   for a Prefetch Abort
```

```
SUBS PC, R14_abt, #8   for a Data Abort
```

This restores both the PC and the CPSR, and retries the aborted instruction.

———— **Note** ————

There are restrictions on the use of the external abort signal. See *External aborts* on page 7-27.

2.8.7   Software interrupt

The SWI instruction is used for entering Supervisor mode, usually to request a particular supervisor function. A SWI handler must return by executing the following irrespective of the state (ARM or Thumb):

```
MOV PC, R14_svc
```

This restores the PC and CPSR, and returns to the instruction following the SWI.

2.8.8   Undefined instruction

When the ARM720T processor encounters an instruction that it cannot handle, it takes the undefined instruction trap. This mechanism can be used to extend either the Thumb or ARM instruction set by software emulation.

After emulating the failed instruction, the trap handler must execute the following irrespective of the state (ARM or Thumb):

```
MOVS PC, R14_und
```

This restores the CPSR and returns to the instruction following the Undefined Instruction.

2.8.9 Exception vectors

The ARM720T processor can have exception vectors mapped to either low or high addresses, controlled by the V bit in the control register of the system control coprocessor (See *Register 1, control register* on page 3-5). Table 2-4 shows the exception vector addresses.

Table 2-4 Exception vector addresses

High address	Low address	Exception	Mode on entry
0xFFFF0000	0x00000000	Reset	Supervisor
0xFFFF0004	0x00000004	Undefined instruction	Undefined
0xFFFF0008	0x00000008	Software interrupt	Supervisor
0xFFFF000C	0x0000000C	Abort (prefetch)	Abort
0xFFFF0010	0x00000010	Abort (data)	Abort
0xFFFF0014	0x00000014	Reserved	Reserved
0xFFFF0018	0x00000018	IRQ	IRQ
0xFFFF001C	0x0000001C	FIQ	FIQ

——— **Note** ———

The low addresses are the defaults.

2.8.10 Exception priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they are handled:

1. Reset (highest priority).

2. Data Abort.

3. FIQ.

4. IRQ.

5. Prefetch Abort.

6. Undefined Instruction, SWI (lowest priority).

2.8.11   Exception restrictions

Undefined Instruction and SWI are mutually exclusive, because they each correspond to particular (non-overlapping) decodings of the current instruction.

If a Data Abort occurs at the same time as an FIQ, and FIQs are enabled, the CPSR F flag is clear, the ARM720T processor enters the Data Abort handler and then immediately proceeds to the FIQ vector. A normal return from FIQ causes the Data Abort handler to resume execution. Placing Data Abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection. The time for this exception entry must be added to worst-case FIQ latency calculations.

2.9 Relocation of low virtual addresses by the FCSE PID

The ARM720T processor provides a mechanism, *Fast Context Switch Extension* (FCSE), to translate virtual addresses to physical addresses based on the current value of the FCSE *Process IDentifier* (PID).

The virtual address produced by the processor core going to the IDC and MMU can be relocated if it lies in the bottom 32MB of the virtual address. That is, virtual address bits [31:25] = b0000000 by the substitution of the seven bits [31:25] of the FCSE PID register in the CP15 coprocessor.

A change to the FCSE PID exhibits similar behavior to a delayed branch if:

*   the two instructions fetched immediately following an instruction to change the FCSE PID are fetched with a relocation to the previous FCSE PID

*   the addresses of the instructions being fetched lie within the range of addresses to be relocated.

On reset, the FCSE PID register bits [31:25] are set to b0000000, disabling all relocation. For this reason, the low address reset exception vector is effectively never relocated by this mechanism.

———— **Note** ————

All addresses produced by the processor core undergo this translation if they lie in the appropriate address range. This includes the exception vectors if they are configured to lie in the bottom of the virtual memory map. This configuration is determined by the V bit in the CP15 control register (CP15 register 1).

2.10   Reset

When the **HRESETn** signal goes LOW, the ARM720T processor:

1.   Abandons the executing instruction.

2.   Flushes the cache and *Translation Lookaside Buffer* (TLB).

3.   Disables the *Write Buffer* (WB), cache, and MMU.

4.   Resets the FCSE PID.

5.   Continues to fetch instructions from incrementing word addresses.

When **HRESETn** is LOW, the processor samples the **VINITHI** external input and stores the result in the V bit in CP15 register 1.

When **HRESETn** goes HIGH again, the ARM720T processor:

1.   Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and SPSR is not defined.

2.   Forces M[4:0] to b10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR T bit.

3.   Forces the PC to fetch the next instruction from the reset exception vector. Exception vectors are located at either high or low addresses depending on the state of the V bit in CP15 register 1 (LOW = low addresses, HIGH = high addresses).

4.   Resumes execution in ARM state.

2.11 Implementation-defined behavior of instructions

The *ARM Architectural Reference Manual* defines the instruction set of the ARM720T processor:

- See *Indexed Addressing on a Data Abort* for the behavior of instructions that are identified as implementation-defined in the *ARM Architectural Reference Manual*.
- See *Early termination* for those features that define signed and unsigned early termination on the ARM720T processor.

2.11.1 Indexed Addressing on a Data Abort

In the event of a Data Abort with pre-indexed or post-indexed addressing, the value left in Rn is defined to be the updated base register value for the following instructions:

- LDC
- LDM
- LDR
- LDRB
- LDRBT
- LDRH
- LDRSB
- LDRSH
- LDRT
- STC
- STM
- STR
- STRB
- STRBT
- STRH
- STRT.

2.11.2 Early termination

On the ARM720T, early termination is defined as:

MLA, MUL Signed early termination.

SMULL, SMLAL Signed early termination.

UMULL, UMLAL Unsigned early termination.

ARM DDI 0229A

Chapter 3
Configuration

This chapter describes the configuration of the ARM720T processor. It contains the following sections.

- *About configuration* on page 3-2
- *Internal coprocessor instructions* on page 3-3
- *Registers* on page 3-4.

3.1 About configuration

The operation and configuration of ARM720T is controlled:

* directly using coprocessor instructions to CP15, the system control coprocessor
* indirectly using the MMU page tables.

The coprocessor instructions manipulate a number of on-chip registers that control the configuration of the following:

* cache
* write buffer
* MMU
* other configuration options.

3.1.1 Compatibility

To ensure backwards compatibility of future CPUs:

* all reserved or unused bits in registers and coprocessor instructions must be programmed to 0

* invalid registers must not be read or written

* the following bits must be programmed to 0:
  — Register 1, bits[31:14] and bits [12:10]
  — Register 2, bits[13:0]
  — Register 5, bits[31:9]
  — Register 7, bits[31:0]
  — Register 13 FCSE PID, bits[24:0].

3.1.2 Notation

Throughout this section, the following terms and abbreviations are used:

Unpredictable (UNP)

> If specified for reads, the data returned when reading from this location is unpredictable. It can have any value. If specified for writes, writing to this location causes unpredictable behavior or change in device configuration.

Should Be Zero (SBZ)

> When writing to this location, all bits of this field should be zero.

3.2   Internal coprocessor instructions

The instruction set for the ARM720T macrocell enables you to implement specialized additional instructions using coprocessors. These are separate processing units that are coupled to the ARM720T processor, although CP15 is built into the ARM720T (Rev 4) macrocell.

——— **Note** ———

The CP15 register map might change in future ARM processors. You are strongly recommended to structure software so that any code accessing CP15 is contained in a single module. It can then be updated easily.

CP15 registers can only be accessed with `MRC` and `MCR` instructions in a privileged mode. The instruction bit pattern of the `MRC` and `MCR` instructions is shown in Figure 3-1.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

| Cond | 1 1 1 0 | opcode_1 | L | CRn | Rd | 1 1 1 1 | opcode_2 | 1 | CRm |

Figure 3-1 MRC and MCR bit pattern

`CDP`, `LDC`, and `STC` instructions, as well as unprivileged `MRC` and `MCR` instructions to CP15 cause the Undefined Instruction trap to be taken.

The CRn field of `MRC` and `MCR` instructions specifies the coprocessor register to access. The CRm field and opcode_2 fields specify a particular action when addressing some registers.

In all instructions accessing CP15:

*   the opcode_1 field should be zero (SBZ)

*   the opcode_2 and CRm fields should be zero except when accessing registers 7, 8, and 13 when the specified values must be used to select the desired cache, TLB, or process identifier operations.

3.3 Registers

The ARM720T processor contains registers that control the cache and MMU operation. These registers are accessed using MCR and MRC instructions to CP15 with the processor in a privileged mode.

Only some of registers R0 to R15 are valid. An access to an invalid register causes neither the access nor an undefined instruction trap, and therefore must never be carried out.

Table 3-1 Cache and MMU control register

Register	Register reads	Register writes
0	ID register	Reserved
1	Control	Control
2	Translation table base	Translation table base
3	Domain access control	Domain access control
4	Reserved	Reserved
5	Fault status	Fault status
6	Fault address	Fault address
7	Reserved	Cache operations
8	Reserved	TLB operations
9 – 12	Reserved	Reserved
13	Process identifier	Process identifier
14 – 15	Reserved	Reserved

3.3.1 Register 0, ID register

Reading from CP15 register 0 returns the value:

`0x41807204`

───── **Note** ─────

The final nibble represents the core revision.

─────────────

The CRm and opcode_2 fields should be zero when reading CP15 register 0. This is shown in Figure 3-2.

31 30 29 28	27 26 25 24	23 22 21 20	19 18 17 16	15 14 13 12	11 10 09 08	07 06 05 04	03 02 01 00
0 1 0 0	0 0 0 1	1 0 0 0	0 0 0 0	0 1 1 1	0 0 1 0	0 0 0 0	0 1 0 0

Figure 3-2 ID register read

Writing to CP15 register 0 is unpredictable. ID register write is shown in Figure 3-3.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
UNP

Figure 3-3 ID register write

3.3.2 Register 1, control register

Reading from CP15 register 1 reads the control bits. The CRm and opcode_2 fields should be zero when reading CP15 register 1. Register 1 read is shown in Figure 3-4.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14	13	12 11 10 09 08	07	06	05	04	03	02	01	00
UNP	V	UNP	R	S	B	L	D	P	W	C	A	M

Figure 3-4 Register 1 read

Writing to CP15 register 1 sets the control bits. The CRm and opcode_2 fields should be zero when writing to CP15 register 1. Register 1 write is shown in Figure 3-5.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14	13	12 11 10	09 08	07	06	05	04	03	02	01	00
UNP/SBZ	V	UNP/ SBZ	R	S	B	L	D	P	W	C	A	M

Figure 3-5 Register 1 write

With the exception of the V bit, all defined control bits are set to zero on reset. The control bits have the following functions:

M Bit 0   MMU enable/disable:

                    0 = MMU disabled

                    1 = MMU enabled.

A Bit 1   Alignment fault enable/disable:

                    0 = Address Alignment Fault Checking disabled

                    1 = Address Alignment Fault Checking enabled.

C Bit 2   Cache enable/disable:

                    0 = Instruction and/or Data Cache (IDC) disabled

                    1 = Instruction and/or Data Cache (IDC) enabled.

W Bit 3   Write buffer enable/disable:

                    0 = Write Buffer disabled

                    1 = Write Buffer enabled.

P Bit 4   When read, returns 1. When written, is ignored.

D Bit 5   When read, returns 1. When written, is ignored.

L Bit 6   When read, returns 1. When written, is ignored.

B Bit 7   Big-endian/little-endian:

                    0 = Little-endian operation

                    1 = Big-endian operation.

S Bit 8   System protection: Modifies the MMU protection system.

R Bit 9   ROM protection: Modifies the MMU protection system.

Bits 12:10   When read, this returns an unpredictable value. When written, it
                    should be zero, or a value read from these bits on the same
                    processor.

——— **Note** ———

Using a read-write-modify sequence when modifying this register provides the greatest
future compatibility.

V Bit 13   Location of exception vectors:

                    0 = low addresses

                    1 = high addresses.

Copyright © 2001 ARM Limited. All rights reserved.   ARM DDI 0229A

The value of the V bit reflects the state of the **VINITHI** external input, sampled while **HRESETn** is LOW.

Bits 31:14   When read, this returns an unpredictable value. When written, it should be zero, or a value read from these bits on the same processor.

Enabling the MMU

You must take care if the translated address differs from the untranslated address, because the instructions following the enabling of the MMU are fetched using no address translation. Enabling the MMU can be considered as a branch with delayed execution.

A similar situation occurs when the MMU is disabled. The correct code sequence for enabling and disabling the MMU is given *Interaction of the MMU and cache* on page 7-28.

———— **Note** ————
If the cache and write buffer are enabled when the MMU is not enabled, the results are unpredictable.

3.3.3   Register 2, translation table base register

Reading from CP15 register 2 returns the pointer to the currently active first-level translation table in bits [31:14] and an unpredictable value in bits [13:0]. The CRm and opcode_2 fields should be zero when reading CP15 register 2.

Writing to CP15 register 2 updates the pointer to the currently active first-level translation table from the value in bits [31:14] of the written value. Bits [13:0] should be zero. The CRm and opcode_2 fields should be zero when writing CP15 register 2. Register 2 is shown in Figure 3-6.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Translation base table	UNP/SBZ

Figure 3-6 Register 2

3.3.4   Register 3, domain access control register

Reading from CP15 register 3 returns the value of the domain access control register.

Writing to CP15 register 3 writes the value of the domain access control register.

The domain access control register consists of 16 2-bit fields, each of which defines the access permissions for one of the 16 domains (D15-D0).

The CRm and opcode_2 fields should be zero when reading or writing to CP15 register 3. This is shown in Figure 3-7.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
```

| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
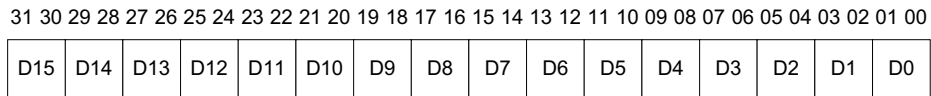
Figure 3-7 Register 3

3.3.5   Register 4, reserved

Register 4 is reserved. Reading CP15 register 4 is unpredictable. Writing to CP15 register 4 is unpredictable. This is shown in Figure 3-8.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
```

| UNP |

Figure 3-8 Register 4

3.3.6   Register 5, fault status register

Reading CP15 register 5 returns the value of the *Fault Status Register* (FSR). The FSR contains the source of the last fault.

——— **Note** ———

Only the bottom 9 bits are returned. The upper 23 bits are unpredictable.

                                   ARM DDI 0229A

The FSR indicates the domain and type of access being attempted when an abort occurred:

Bit 8   This is always read as zero. Bit 8 is ignored on writes.

Bits [7:4]   These specify which of the 16 domains (D15-D0) was being accessed when a fault occurred.

Bits [3:1]   These indicate the type of access being attempted.

The encoding of these bits is shown in *Fault address and fault status registers* on page 7-21. The FSR is always updated when a fault occurs.

Writing to CP15 register 5 sets the FSR to the value of the data written. This is useful when a debugger has to restore the value of the FSR. The upper 24 bits written should be zero.

The CRm and opcode_2 fields should be zero when reading or writing CP15 register 5. Register 5 is shown in Figure 3-9.

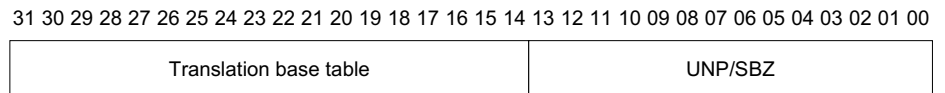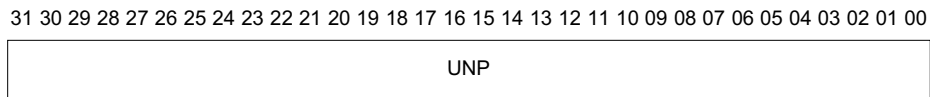31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

UNP/SBZ	0	Domain	Status

Figure 3-9 Register 5

3.3.7   Register 6, Fault Address Register

Reading CP15 register 6 returns the value of the *Fault Address Register* (FAR). The FAR holds the virtual address of the access that was attempted when a fault occurred. The FAR is updated for data faults and prefetch faults.

Writing to CP15 register 6 sets the FAR to the value of the data written. This is useful when a debugger has to restore the value of the FAR.

The CRm and opcode_2 fields should be zero when reading or writing CP15 register 6. Register 6 is shown in Figure 3-10.

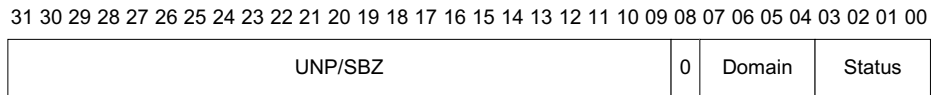31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Fault address
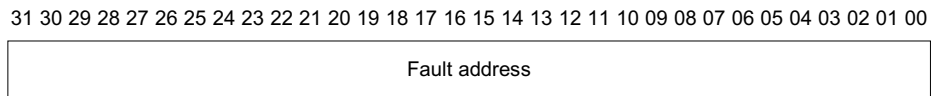
Figure 3-10 Register 6

—— **Note** ——

Register 6 contains a modified virtual address if the FCSE PID register is nonzero.

3.3.8   Register 7, cache operations

Writing to CP15 register 7 manages the unified instruction and data cache of the ARM720T. Only one cache operation is defined using the following opcode_2 and CRm fields in the MCR instruction that writes the CP15 register 7.

—— **Caution** ——

The Invalidate ID cache function invalidates all cache data. Use this with caution.

Register 7 is shown in Table 3-2.

Table 3-2 Cache operation

Function	opcode_2 value	CRm value	Data	Instruction
Invalidate ID cache	0b000	0b0111	SBZ	MCR p15, 0, Rd, c7, c7, 0

Reading from CP15 register 7 is undefined.

3.3.9   Register 8, TLB operations

Writing to CP15 register 8 controls the *Translation Lookaside Buffer* (TLB). The ARM720T processor implements a unified instruction and data TLB.

Two TLB operations are defined. The function to be performed is selected by the opcode_2 and CRm fields in the MCR instruction used to write CP15 register 8.

The TLB operations and the instructions that you can use are shown in Table 3-3.

Table 3-3 TLB operations

Function	opcode_2 value	CRm value	Data	Instruction
Invalidate TLB	0b000	0b0111	SBZ	`MCR p15, 0, Rd, c8, c5, 0`
				`MCR p15, 0, Rd, c8, c6, 0`
				`MCR p15, 0, Rd, c8, c7, 0`
Invalidate TLB single entry	0b001	0b0111	Modified Virtual Address	`MCR p15, 0, Rd, c8, c5, 1`
				`MCR p15, 0, Rd, c8, c6, 1`
				`MCR p15, 0, Rd, c8, c7, 1`

In the instructions shown in Table 3-3, `c7` is the preferred value for the CRn field, because it indicates a unified MMU.

Reading from CP15 register 8 is undefined.

The Invalidate TLB single entry function invalidates any TLB entry corresponding to the *Modified Virtual Address* (MVA) given in `Rd`.

3.3.10 Registers 9 to 12, reserved

Accessing any of these registers is undefined. Writing to any of these registers is undefined.

3.3.11 Register 13, process identifier

Two independent process identifier registers can be accessed using register 13:
- *Fast context switch extension process identifier*
- *Trace process identifier* on page 3-12.

Fast context switch extension process identifier

Reading from CP15 register 13 with opcode_2 = 0 returns the value of the FCSE PID. This is shown in Figure 3-11 on page 3-12.
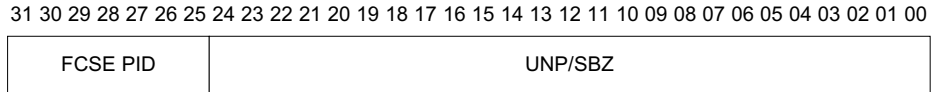
```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
```

FCSE PID	UNP/SBZ

Figure 3-11 Register 13 with opcode_2=0

──── **Note** ────

Only bits [31:25] are returned. The remaining 25 bits are unpredictable.

Writing to CP15 register 13 with opcode_2 = 0 updates the FCSE PID from the value in bits [31:25]. Bits [24:0] should be zero. The FCSE PID is set to b0000000 on Reset.

The CRm and opcode_2 should be zero when reading or writing the FCSE PID.

Changing FCSE PID

You must take care when changing the FCSE PID because the following instructions have been fetched with the previous FCSE PID. In this way, changing the FCSE PID has similarities with a branch with delayed execution. See *Relocation of low virtual addresses by the FCSE PID* on page 2-23.

Trace process identifier

A 32-bit read/write register is provided to hold a Trace *PROCess IDentifier* (PROCID) up to 32-bits in length visible to the ETM7. This is achieved by reading from or writing to the CP15 register 13 with opcode_2 = 1 as shown in Figure 3-12.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
```
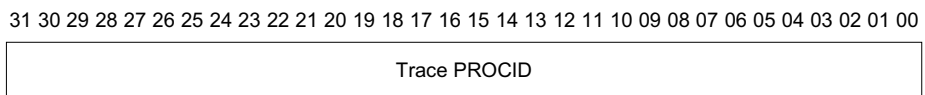
Trace PROCID

Figure 3-12 Register 13 with opcode_2=1

The **PROCIDWR** signal is exported to notify the ETM7 that the Trace PROCID has been written.

3.3.12 Register 14, reserved

Accessing this register is undefined. Writing to this register is undefined.

 ARM DDI 0229A

3.3.13 Register 15, test register

The CP15 register 15 is used for device-specific test operations. For more information, see Chapter 11 *Test Support*.

Chapter 4
Instruction and Data Cache

This chapter describes the instruction and data cache. It contains the following sections:

- *About the instruction and data cache* on page 4-2
- *IDC validity* on page 4-4
- *IDC enable, disable, and reset* on page 4-5.

4.1 About the instruction and data cache

The cache only operates on a write-through basis with a read-miss allocation policy and a random replacement algorithm.

4.1.1 IDC operation

The ARM720T contains an 8KB mixed *Instruction and Data Cache* (IDC).

The IDC has 512 lines of 32 bytes (eight words), arranged as a 4-way set-associative cache, and uses the virtual addresses generated by the processor core after relocation by the FCSE PID as appropriate. The IDC is always reloaded a line at a time (eight words). It can be enabled or disabled using the ARM720T control register and is disabled on **HRESETn**.

—— **Note** ——

The MMU must never be disabled when the cache is on. However, you can enable the two devices simultaneously with a single write to the control register (see *Register 1, control register* on page 3-5).

4.1.2 Cachable bit

The C bit determines whether data being read can be placed in the IDC and used for subsequent read operations. Typically, main memory is marked as cachable to improve system performance, and I/O space is marked as noncachable to stop the data being stored in the ARM720T cache.

For example, if the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of the initial data held in the cache. The cachable bit can be configured for both pages and sections.

Cachable reads (C=1)

A line fetch of eight words is performed when a cache miss occurs in a cachable area of memory, and it is randomly placed in a cache bank.

—— **Note** ——

Memory aborts are not supported on cache line fetches.

Uncachable reads (C=0)

An external memory access is performed and the cache is not written.

4.1.3 Read-lock-write

The IDC treats the read-lock-write instruction as a special case:

Read phase   Always forces a read of external memory, regardless of whether the data is contained in the cache.

Write phase   Is treated as a normal write operation. If the data is already in the cache, the cache is updated.

Externally, the two phases are flagged as indivisible by asserting the **HLOCK** signal.

4.2 IDC validity

The IDC operates with virtual addresses, so you must ensure that its contents remain consistent with the virtual to physical mappings performed by the MMU. If the memory mappings are changed, the IDC validity must be ensured.

4.2.1 Software IDC flush

The entire IDC can be marked as invalid by writing to the cache operations register R7. The cache is flushed immediately the register is written, but the following two instruction fetches can come from the cache before the register is written.

4.2.2 Doubly-mapped space

Because the cache works with virtual addresses, it is assumed that every virtual address maps to a different physical address. If the same physical location is accessed by more than one virtual address, the cache cannot maintain consistency. Each virtual address has a separate entry in the cache, and only one entry can be updated on a processor write operation.

To avoid any cache inconsistencies, both doubly-mapped virtual addresses must be marked as uncachable.

4.3   IDC enable, disable, and reset

The IDC is automatically disabled and flushed on **HRESETn**. When enabled, cachable read accesses cause lines to be placed in the cache.

To enable the IDC:

1.   Make sure that the MMU is enabled first by setting bit 0 in the control register.

2.   Enable the IDC by setting bit 2 in the control register. The MMU and IDC can be enabled simultaneously with a single write to the control register.

To disable the IDC:

1.   Clear bit 2 in the control register.

2.   Perform a flush by writing to the cache operations register.

Chapter 5
Write Buffer

This chapter describes the write buffer. It contains the following sections:
- *About the write buffer* on page 5-2
- *Write buffer operation* on page 5-3.

5.1   About the write buffer

The write buffer of the ARM720T processor is provided to improve system performance. It can buffer up to:

•   eight words of data

•   eight independent addresses.

You can enable and disable the write buffer using the W bit, bit 3, in the control register. The buffer is disabled and flushed on reset.

The operation of the write buffer is further controlled by the *Bufferable* (B) bit, which is stored in the MMU page tables. For this reason, the MMU must be enabled before using the write buffer. The two functions can, however, be enabled simultaneously, with a single write to the control register.

For a write to use the write buffer, both the W bit in the control register and the B bit in the corresponding page table must be set.

——— **Note** ———

It is not possible to abort buffered writes externally. The error response on **HRESP[1:0]** is ignored. Areas of memory that can generate aborts must be marked as unbufferable in the MMU page tables.

5.1.1   Bufferable bit

This bit controls whether a write operation uses or does not use the write buffer. Typically, main memory is bufferable and I/O space unbufferable. The B bit can be configured for both pages and sections.

5.2 Write buffer operation

You control the operation of the write buffer with CP15 register 1, the control register (see *Register 1, control register* on page 3-5).

When the CPU performs a write operation, the translation entry for that address is inspected and the state of the B bit determines the subsequent action. If the write buffer is disabled using the control register, buffered writes are treated in the same way as unbuffered writes.

To enable the write buffer:

1.   Ensure that the MMU is enabled by setting bit 0 in the control register.

2.   Enable the write buffer by setting bit 3 in the control register.

      You can enable the MMU and write buffer simultaneously with a single write to the control register.

To disable the write buffer, clear bit 3 in the control register. Any writes already in the write buffer complete normally. The write buffer attempts a write operation as long as there is data present.

5.2.1 Bufferable write

If the write buffer is enabled and the processor performs a write to a bufferable area, the data is placed in the write buffer at the speed of **HCLK**, and the CPU continues execution. The write buffer then performs the external write in parallel.

If the write buffer is full, the processor is stalled until there is an empty line in the buffer.

5.2.2 Unbufferable write

If the write buffer is disabled or the CPU performs a write to an unbufferable area, the processor is stalled until the write buffer empties and the write completes externally. This might require synchronization and several external clock cycles.

5.2.3 Read-lock-write

The write phase of a read-lock-write sequence (SWP instruction) is treated as an unbuffered write, even if it is marked as buffered.

5.2.4 Reading from a noncachable area

If the CPU performs a read from a noncachable area, the write buffer is drained and the processor is stalled.

5.2.5   Draining the write buffer

You can force a drain of the write buffer by performing a read from a noncachable location.

5.2.6   Multi-word writes

All accesses are treated as non-sequential, which means that writes require an address slot and a data slot for each word. For this reason, buffered STM accesses are less efficient than unbuffered STM accesses. You are advised to disable the write buffer (by clearing bit 3 in CP15 register 1) before moving large blocks of data.

                   ARM DDI 0229A

Chapter 6
The Bus Interface

This chapter describes the signals on the bus interface of the ARM720T (Rev 4) processor. It contains the following sections:

6.1 About the bus interface

The ARM720T (Rev 4) processor is an *Advanced High-performance Bus* (AHB) bus master. It is described fully in the *AMBA Specification (Rev 2.0)*.

6.1.1 Summary of the AHB transfer mechanism

An AHB transfer comprises the following:

Address phase   This lasts only a single cycle. The address cannot be extended, so all slaves must sample the address during the address phase.

Data phase   This phase can be extended using the **HREADY** signal. When LOW, **HREADY** causes wait states to be inserted into the transfer and allows extra time for a slave to provide or sample data.

A write data bus is used to move data from the master to a slave.

A read data bus is used to move data from a slave to the master.

Figure 6-1 shows a transfer with no wait states (this is the simplest type of transfer).
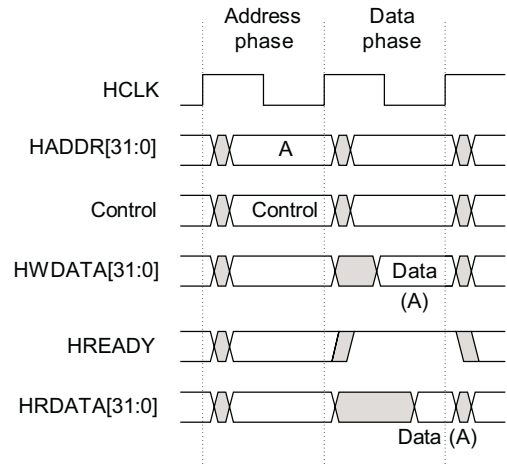
Figure 6-1 Simple AHB transfer

A granted bus master starts an AHB transfer by driving the address and control signals. These signals provide the following information about the transfer:

- address
- direction
- width of the transfer
- whether the transfer forms part of a burst
- the type of burst.

A burst is a series of transfers. The ARM720T (Rev 4) processor performs the following types of burst:

- incrementing burst of unspecified length
- 8-beat incrementing burst.

  Incrementing bursts do not wrap at address boundaries. The address of each transfer in the burst is an increment of the address of the previous transfer in the burst.

For more information, see *Address and control signals* on page 6-9.

For a complete description of the AHB transfer mechanism, see the *AMBA Specification (Rev 2.0)*.

6.2   Bus interface signals

The signals in the ARM720T (Rev 4) processor bus interface can be grouped into the following categories:

Transfer type   **HTRANS[1:0]**

See *Transfer types* on page 6-6.

Address and control

HADDR[31:0]

HWRITE

HSIZE[2:0]

HBURST[2:0]

HPROT[3:0]

See *Address and control signals* on page 6-9.

Slave transfer response

HREADY

HRESP[1:0]

See *Slave transfer response signals* on page 6-12.

Data   **HRDATA[31:0]**

HWDATA[31:0]

See *Data buses* on page 6-14.

Arbitration   **HBUSREQ**

HGRANT

HLOCK

See *Arbitration* on page 6-17.

Clock   **HCLK**

HCLKEN

See *Bus clocking* on page 6-18.

Reset   **HRESETn**

See *Reset* on page 6-19.

Each of these signal groups shares a common timing relationship to the bus interface cycle. All signals in the ARM720T (Rev 4) processor bus interface are generated from or sampled by the rising edge of **HCLK**.

The AHB bus master interface signals are shown in Figure 6-2.

Figure 6-2 AHB bus master interface

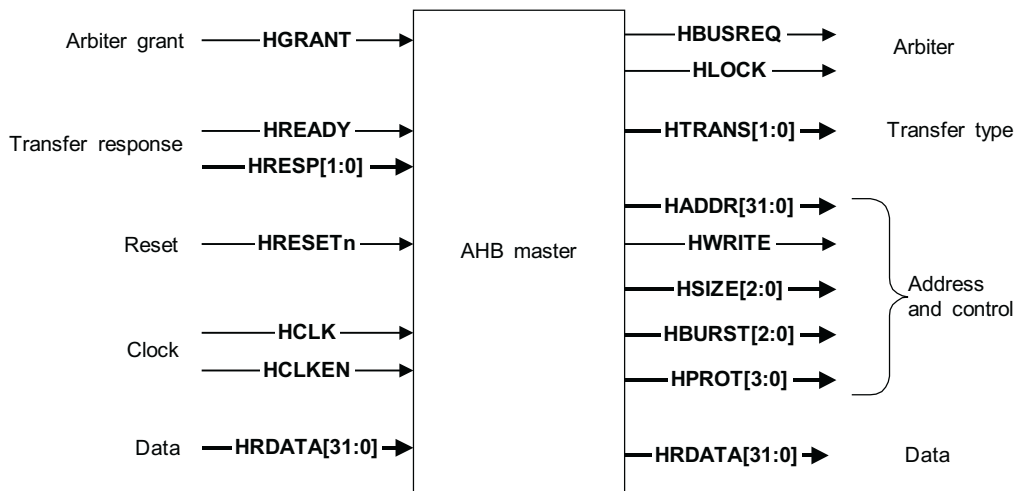## 6.3 Transfer types

The ARM720T processor bus interface is pipelined, so the address-class signals and the memory request signals are broadcast in the bus cycle ahead of the bus cycle to which they refer. This gives the maximum time for a memory cycle to decode the address and respond to the access request.
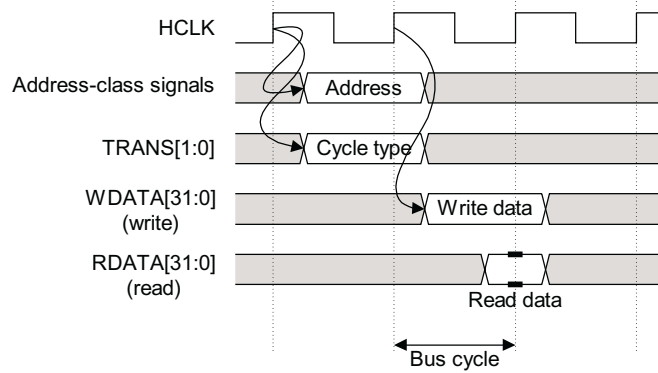
A single memory cycle is shown in Figure 3-1.

Figure 6-3 Simple memory cycle

There are three types of transfer. The transfer type is indicated by the **HTRANS[1:0]** signal as shown in Table 6-1.

Table 6-1 Transfer type encoding

HTRANS[1:0]	Transfer type	Description
00	IDLE	Indicates that no data transfer is required. The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer.
		Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer must be ignored by the slave.
10	NONSEQ	Indicates the first transfer of a burst or a single transfer. The address and control signals are unrelated to the previous transfer.
		Single transfers on the bus are treated as bursts that comprise one transfer.
11	SEQ	In a burst, all transfers apart from the first are SEQUENTIAL.
		The address is related to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). In the case of a wrapping burst, the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8, or 16).
		The control information is identical to the previous transfer.

——— **Note** ———

In the *AMBA Specification (Rev 2.0)*, **HTRANS[1:0]** = 01 indicates a BUSY cycle, but these are never inserted by the ARM720T (Rev 4) processor.

Figure 6-4 on page 6-8 shows some examples of different transfer types.
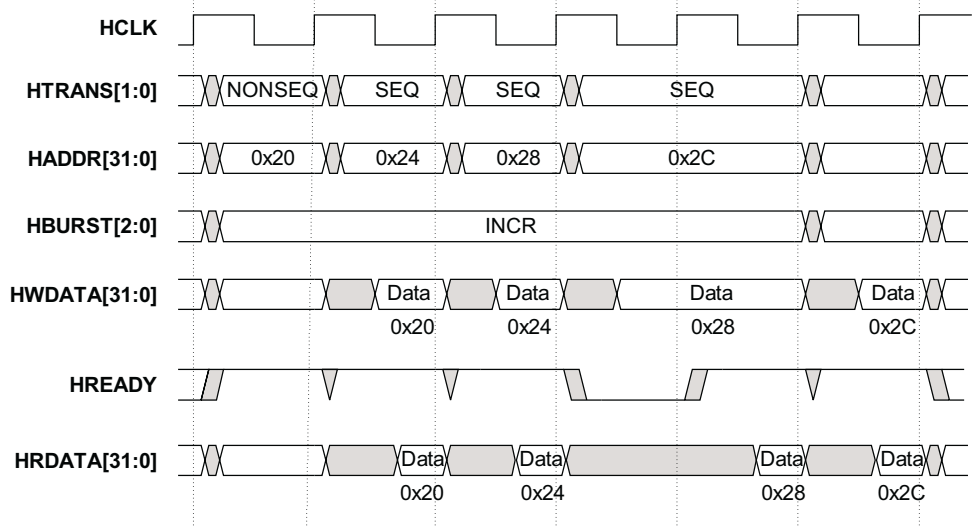
Figure 6-4 Transfer type examples

In Figure 6-4:

- The first transfer is the start of a burst and is therefore NONSEQUENTIAL.

- The master performs the second transfer of the burst immediately.

- The master performs the third transfer of the burst immediately, but this time the slave is unable to complete and uses **HREADY** to insert a single wait state.

- The final transfer of the burst completes with zero wait states.

 ARM DDI 0229A

6.4 Address and control signals

The address and control signals are described in the following sections:

• *HADDR[31:0]*

• *HWRITE*

• *HSIZE[2:0]*

• *HBURST[2:0]* on page 6-10

• *HPROT[3:0]* on page 6-10.

6.4.1 HADDR[31:0]

HADDR[31:0] is the 32-bit address bus that specifies the address for the transfer. All addresses are byte addresses, so a burst of word accesses results in the address bus incrementing by four for each cycle.

The address bus provides 4GB of linear addressing space. This means that:

• when a word access is signalled, the memory system must ignore the bottom two bits, **HADDR[1:0]**

• when a halfword access is signalled the memory system must ignore the bottom bit, **HADDR[0]**.

6.4.2 HWRITE

HWRITE specifies the direction of the transfer as follows:

HWRITE HIGH   Indicates an ARM720T processor write cycle.

HWRITE LOW   Indicates an ARM720T processor read cycle.

A burst of S cycles is always either a read burst or a write burst. The direction cannot be changed in the middle of a burst.

6.4.3 HSIZE[2:0]

The **SIZE[2:0]** bus encodes the size of the transfer. The ARM720T processor can transfer word, halfword, and byte quantities. This is encoded on **SIZE[2:0]** as shown in Table 6-2 on page 6-10.

——— **Note** ———

To use the C compiler and the ARM debug tool chain, your system must support the writing of arbitrary bytes and halfwords. You must provide write enables down to the level of every individual byte to ensure support for all possible transfer sizes, up to the bus width.

Table 6-2 Transfer size encodings

HSIZE[2:0]	Size	Transfer width
000	8 bits	Byte
001	16 bits	Halfword
010	32 bits	Word

6.4.4 HBURST[2:0]

HBURST[2:0] indicates the type of burst, as shown in Table 6-3.

Table 6-3 Burst type encodings

HBURST[2:0]	Type	Description
000	SINGLE	Single transfer
001	INCR	Incrementing burst of unspecified length
101	INCR8	8-beat incrementing burst

For more details of burst operation, see the *AMBA Specification (Rev 2.0)*.

6.4.5 HPROT[3:0]

HPROT[3:0] is the protection control bus. These signals provide additional information about a bus access and are primarily intended to enable a module to implement an access permission scheme.

These signals indicate whether the transfer is:
- an opcode fetch or data access
- a privileged-mode access or User-mode access.

 ARM DDI 0229A

For bus masters with a memory management unit, these signals also indicate whether the current access is cachable or bufferable.

Table 6-4 shows the protection control encodings.

Table 6-4 Protection control encodings

HPROT[3] cachable	HPROT[2] bufferable	HPROT[1] privileged	HPROT[0] data/opcode	Description
-	-	-	0	Opcode fetch
-	-	-	1	Data access
-	-	0	-	User access
-	-	1	-	Privileged access
-	0	-	-	Not bufferable
-	1	-	-	Bufferable
0	-	-	-	Not cachable
1	-	-	-	Cachable

Some bus masters are not capable of generating accurate protection information, so it is recommended that slaves do not use the **HPROT[3:0]** signals unless strictly necessary.

6.5 Slave transfer response signals

After a master has started a transfer, the slave determines how the transfer progresses. No provision is made in the AHB specification for a bus master to cancel a transfer after it has begun.

Whenever a slave is accessed it must provide a response using the following signals:

HRESP[1:0]   Indicates the status of the transfer.

HREADY   Used to extend the transfer. This signal works in combination with **HRESP[1:0]**.

The slave can complete the transfer in a number of ways. It can:

* complete the transfer immediately

* insert one or more wait states to allow time to complete the transfer

* signal an error to indicate that the transfer has failed

* delay the completion of the transfer, but allow the master and slave to back off the bus, leaving it available for other transfers.

6.5.1 HREADY

The **HREADY** signal is used to extend the data portion of an AHB transfer, as follows:

HREADY LOW   Indicates that the transfer data is to be extended. It causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data.

HREADY HIGH   Indicates that the transfer can complete.

Every slave must have a predetermined maximum number of wait states that it inserts before it backs off the bus, in order to allow the calculation of the latency of accessing the bus. To prevent any single access locking the bus for a large number of clock cycles, it is recommended that slaves do not insert more than 16 wait states.

6.5.2 HRESP[1:0]

HRESP[1:0] is used by the slave to show the status of a transfer. The **HRESP[1:0]** encodings are shown in Table 6-5 on page 6-13.

Table 6-5 Response encodings

HRESP[1:0]	Response	Description
00	OKAY	When **HREADY** is HIGH, this response indicates that the transfer has completed successfully. The OKAY response is also used for any additional cycles that are inserted, with **HREADY** LOW, prior to giving one of the three other responses.
01	ERROR	This response indicates that a transfer error has occurred and the transfer has been unsuccessful. Typically this is used for a protection error, such as an attempt to write to a read-only memory location. The error condition must be signalled to the bus master so that it is aware the transfer has been unsuccessful. A two-cycle response is required for an error condition.
10	RETRY	The RETRY response shows the transfer has not yet completed, so the bus master should retry the transfer. The master should continue to retry the transfer until it completes. A two-cycle RETRY response is required.
11	SPLIT	The transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete. A two-cycle SPLIT response is required.

For a full description of the slave transfer responses, see the *AMBA Specification (Rev 2.0)*.

6.6 Data buses

To enable you to implement an AHB system without the use of tristate drivers, separate 32-bit read and write data buses are required.

6.6.1 HWDATA[31:0]

The write data bus is driven by the bus master during write transfers. If the transfer is extended, the bus master must hold the data valid until the transfer completes, as indicated by **HREADY** HIGH.

All transfers must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (that is A[1:0] = 00), and halfword transfers must be aligned to halfword address boundaries (that is A[0] = 0).

The bus master drives all byte lanes regardless of the size of the transfer:

- For halfword transfers, for example 0x1234, **HWDATA[31:0]** is driven with the value 0x12341234, regardless of endianness.

- For byte transfers, for example 0x12, **HWDATA[31:0]** is driven with the value 0x12121212, regardless of endianness.

6.6.2 HRDATA[31:0]

The read data bus is driven by the appropriate slave during read transfers. If the slave extends the read transfer by holding **HREADY** LOW, the slave has to provide valid data only at the end of the final cycle of the transfer, as indicated by **HREADY** HIGH.

For transfers that are narrower than the width of the bus, the slave only has to provide valid data on the active byte lanes. The bus master is responsible for selecting the data from the correct byte lanes. The following tables identify active byte lanes:
- Table 6-6 on page 6-15 shows active byte lanes for little-endian systems
- Table 6-7 on page 6-16 shows active byte lanes for big-endian systems.

A slave has to provide valid data only when a transfer completes with an OKAY response on **HRESP[1:0]**. SPLIT, RETRY, and ERROR responses do not require valid read data.

6.6.3 Endianness

It is essential that all modules are of the same endianness and also that any data routing or bridges are of the same endianness.

 ARM DDI 0229A

Dynamic endianness is not supported, because in most embedded systems, this leads to a significant silicon overhead that is redundant.

It is recommended that only modules that will be used in a wide variety of applications are made bi-endian, with either a configuration pin or internal control bit to select the endianness. For more application-specific blocks, fixing the endianness to either little-endian or big-endian results in a smaller, lower power, higher performance interface.

Table 6-6 shows active byte lanes for little-endian systems.

Table 6-6 Active byte lanes for a 32-bit little-endian data bus

Transfer size	Address offset	DATA[31:24]	DATA[23:16]	DATA[15:8]	DATA[7:0]
Word	0	✔	✔	✔	✔
Halfword	0	-	-	✔	✔
Halfword	2	✔	✔	-	-
Byte	0	-	-	-	✔
Byte	1	-	-	✔	-
Byte	2	-	✔	-	-
Byte	3	✔	-	-	-

Table 6-7 on page 6-16 shows active byte lanes for big-endian systems.

Table 6-7 Active byte lanes for a 32-bit big-endian data bus

Transfer size	Address offset	DATA[31:24]	DATA[23:16]	DATA[15:8]	DATA[7:0]
Word	0	✔	✔	✔	✔
Halfword	0	✔	✔	-	-
Halfword	2	-	-	✔	✔
Byte	0	✔	-	-	-
Byte	1	-	✔	-	-
Byte	2	-	-	✔	-
Byte	3	-	-	-	✔

6.7 Arbitration

The arbitration mechanism is described fully in the *AMBA Specification (Rev 2.0)*. This mechanism is used to ensure that only one master has access to the bus at any one time. The arbiter performs this function by observing a number of different requests to use the bus and deciding which is currently the highest priority master requesting the bus. The arbiter also receives requests from slaves that want to complete SPLIT transfers.

Any slaves that are not capable of performing SPLIT transfers do not have to be aware of the arbitration process, except that they need to observe the fact that a burst of transfers might not complete if the ownership of the bus is changed.

6.7.1 HBUSREQ

The bus request signal is used by a bus master to request access to the bus. Each bus master has its own **HBUSREQ** signal to the arbiter and there can be up to 16 separate bus masters in any system.

6.7.2 HLOCK

The lock signal is asserted by a master at the same time as the bus request signal. This indicates to the arbiter that the master is performing a number of indivisible transfers and the arbiter must not grant any other bus master access to the bus once the first transfer of the locked transfers has commenced. **HLOCK** must be asserted at least a cycle before the address to which it refers, to prevent the arbiter from changing the grant signals.

6.7.3 HGRANT

The grant signal is generated by the arbiter and indicates that the appropriate master is currently the highest priority master requesting the bus, taking into account locked transfers and SPLIT transfers.

A master gains ownership of the address bus when **HGRANT** is HIGH and **HREADY** is HIGH at the rising edge of **HCLK**.

6.8 Bus clocking

There are two clock inputs on the ARM720T (Rev 4) processor bus interface.

6.8.1 HCLK

The bus is clocked by the system clock, **HCLK**. This clock times all bus transfers. All signal timings are related to the rising edge of **HCLK**.

6.8.2 HCLKEN

HCLK is enabled by the **HCLKEN** signal. You can use **HCLKEN** to slow the bus transfer rate by dividing **HCLK** for the bus interface.

——— **Note** ———

HCLKEN is not a clock enable for the CPU itself, but only for the bus. Use **HREADY** to insert wait states on the bus.

6.9 Reset

The bus reset signal is **HRESETn**. This signal is the global reset, used to reset the system and the bus. It can be asserted asynchronously, but is deasserted synchronously after the rising edge of **HCLK**.

During reset, all masters must ensure the following:

- the address and control signals are at valid levels
- **HTRANS[1:0]** indicates IDLE.

HRESETn is the only active LOW signal in the AMBA AHB specification.

ARM DDI 0229A

Chapter 7
Memory Management Unit

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

- *About the MMU* on page 7-2
- *MMU program-accessible registers* on page 7-4
- *Address translation* on page 7-5
- *MMU faults and CPU aborts* on page 7-20
- *Fault address and fault status registers* on page 7-21
- *Domain access control* on page 7-22
- *Fault checking sequence* on page 7-24
- *External aborts* on page 7-27
- *Interaction of the MMU and cache* on page 7-28.

7.1 About the MMU

The ARM720T processor implements an enhanced ARM architecture v4 MMU to provide translation and access permission checks for the instruction and data address ports of the core. The MMU is controlled from a single set of two-level page tables stored in main memory, that are enabled by the M bit in CP15 register 1, providing a single address translation and protection scheme.

The MMU features are:

* standard ARMv4 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 sections

* access permissions for large pages and small pages can be specified separately for each quarter of the page (these quarters are called subpages)

* 16 domains implemented in hardware

* 64-entry TLB

* hardware page table walks

* round-robin replacement algorithm (also called cyclic)

* invalidate whole TLB, using CP15 register 8

* invalidate TLB entry, selected by *Modified Virtual Address* (MVA), using CP15 register 8.

7.1.1 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:
* the 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.

7.1.2   Translated entries

The TLB caches 64 translated entries. During CPU memory accesses, the TLB provides the protection information to the access control logic.

If the TLB contains a translated entry for the MVA, the access control logic determines if 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 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 VA) 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.

The entry to be written is chosen by cycling sequentially through the TLB locations.

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

7.2   MMU program-accessible registers

Table 7-1 lists the CP15 registers that are used in conjunction with page table descriptors stored in memory to determine the operation of the MMU.

Table 7-1 CP15 register functions

Register	Number	Bits	Register description
Control register	1	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	2	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	3	31:0	Comprises 16 2-bit fields. Each field defines the access control attributes for one of 16 domains (D15–D0).
Fault status register	5	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–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 7-9 on page 7-21.
Fault address register	6	31:0	Holds the MVA associated with the access that caused the abort. See Table 7-9 on page 7-21 for details of the address stored for each type of fault. You can use banked register 14 to determine the VA associated with a Prefetch Abort.
TLB operations register	8	31:0	You can write to this register to make the MMU perform TLB maintenance operations. These are: • invalidating all the entries in the TLB • invalidating a specific entry.

All the CP15 MMU registers, except register 8, contain state. You can read them using `MRC` instructions, and write to them using `MCR` instructions. Registers 5 and 6 are also written by the MMU during all aborts. Writing to register 8 causes the MMU to perform a TLB operation, to manipulate TLB entries. This register cannot be read.

CP15 is described in Chapter 3 *Configuration*, with details of register formats and the coprocessor instructions you can use to access them.

7.3 Address translation

The MMU translates VAs generated by the CPU core, and by CP15 register 13, into physical addresses to access external memory. It also derives and checks the access permission, using the TLB.

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 you to traverse this translation table and load entries into the TLB.

There are one or two stages in the hardware table walking, and permission checking, process. The number of stages depends 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. The page-mapped accesses are for:

* large pages
* small pages
* tiny pages.

The translation process always starts out 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 a subsequent level two fetch.

7.3.1 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* (TTB) register points to the base address of a table in physical memory that contains section or page descriptors, or both. The 14 low-order bits of the TTB register are set to zero on a read, and the table must reside on a 16KB boundary. Figure 7-1 shows the format of the TTB register.
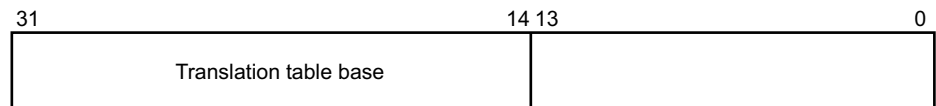
31	14 13	0
Translation table base		

The translation table has up to 4096 x 32-bit entries, each describing 1MB of virtual memory. This allows up to 4GB of virtual memory to be addressed. Figure 7-2 on page 7-6 shows the table walk process.
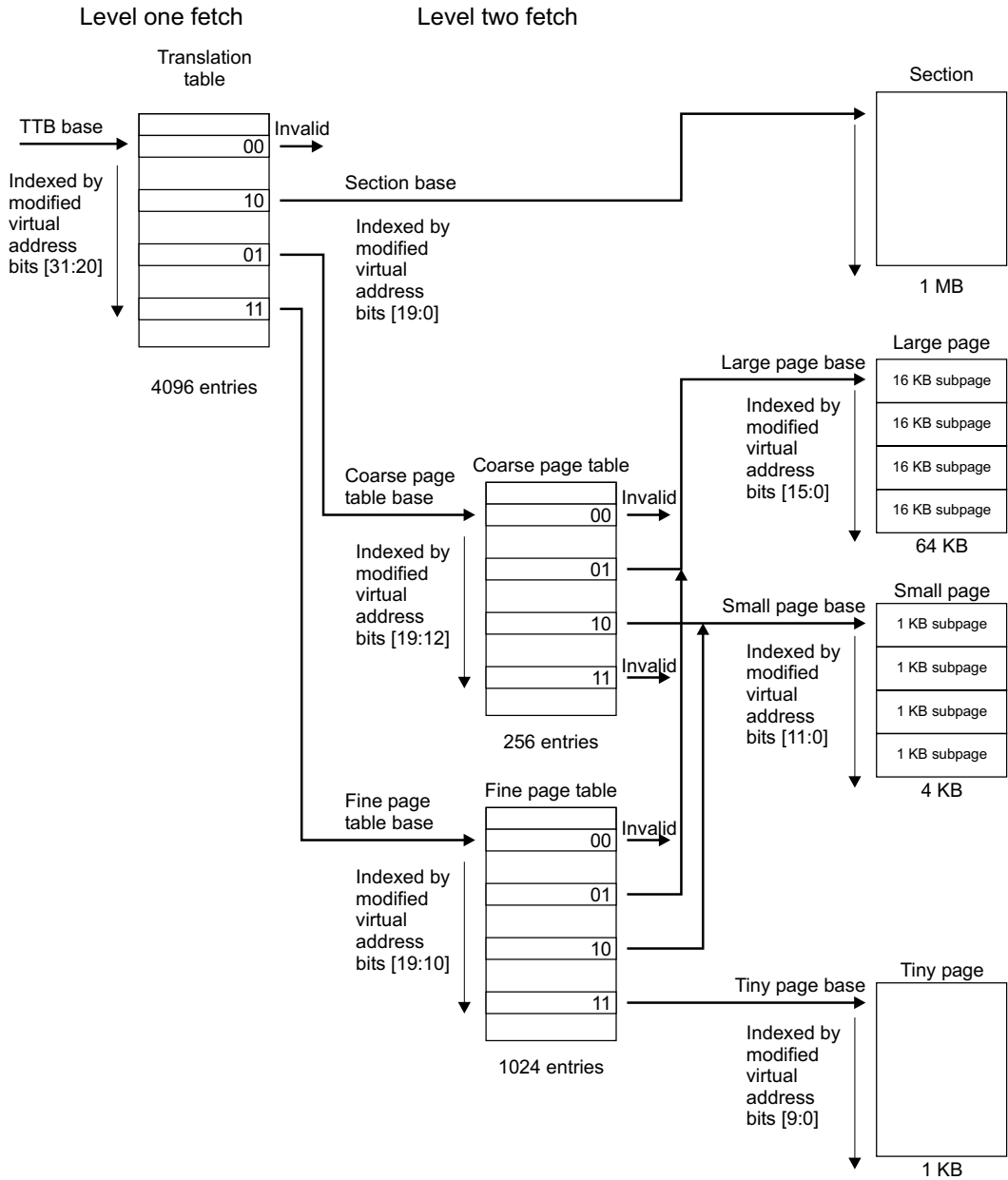
Level one fetch   Level two fetch

Translation table

TTB base   Invalid

00

Indexed by modified virtual address bits [31:20]   Section base

10

01

11

4096 entries

Indexed by modified virtual address bits [19:0]

Section

1 MB

Large page

Large page base   16 KB subpage

Indexed by modified virtual address bits [15:0]   16 KB subpage

16 KB subpage

16 KB subpage

64 KB

Coarse page table base   Coarse page table

Invalid

00

Indexed by modified virtual address bits [19:12]

01

10

11   Invalid

256 entries

Small page base   Small page

1 KB subpage

Indexed by modified virtual address bits [11:0]   1 KB subpage

1 KB subpage

1 KB subpage

4 KB

Fine page table base   Fine page table

Invalid

00

Indexed by modified virtual address bits [19:10]

01

10

11

1024 entries

Tiny page base   Tiny page

Indexed by modified virtual address bits [9:0]

1 KB

Figure 7-2 Translating page tables

7.3.2   Level one fetch

Bits [31:14] of the TTB register are concatenated with bits [31:20] of the MVA to produce a 30-bit address as shown in Figure 7-3.
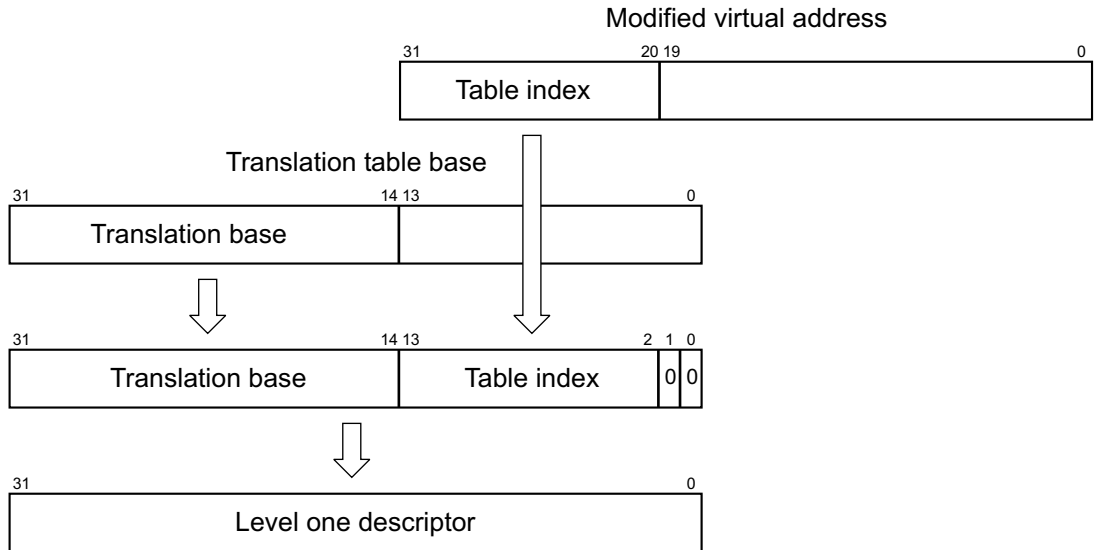

Figure 7-3 Accessing translation table level one descriptors

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

7.3.3   Level one descriptor

The level one descriptor returned is either a section descriptor, a coarse page table descriptor, or a fine page table descriptor, or is invalid. Figure 7-4 on page 7-8 shows the format of a level one descriptor.

Figure 7-4 Level one 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 level two 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.

Level one descriptor bit assignments are shown in Table 7-2.

Table 7-2 Level one 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. *Domain access control* on page 7-22 and *Fault checking sequence* on page 7-24 show how to interpret the access permission bits
9	9	11:9	Should be zero
8:5	8:5	8:5	Domain control bits

Table 7-2 Level one descriptor bits (continued)

Bits			Description
Section	**Coarse**	**Fine**	
4	4	4	Must be 1
3:2	-	-	These bits, C and B, indicate whether the area of memory mapped by this page is treated as cachable or noncachable, and bufferable or nonbufferable. (The system is always write-through.)
-	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 7-3

The two least significant bits of the level one descriptor indicate the descriptor type as shown in Table 7-3.

Table 7-3 Interpreting level one 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

7.3.4 Section descriptor

A section descriptor provides the base address of a 1MB block of memory. Figure 7-5 shows the format of a section descriptor.
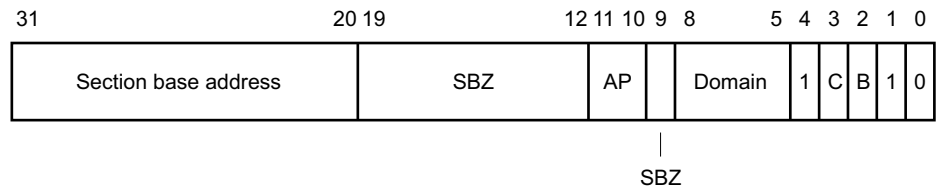
Figure 7-5 Section descriptor

Section descriptor bit assignments are described in Table 7-4.

Table 7-4 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	(AP) 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 backward compatibility
3:2	These bits, C and B, indicate whether the area of memory mapped by this page is treated as cachable or noncachable, and bufferable or nonbufferable. (The system is always write-through.)
1:0	These bits must be 10 to indicate a section descriptor

7.3.5 Coarse page table descriptor

A coarse page table descriptor provides the base address of a page table that contains level two 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 7-6 shows the format of a coarse page table descriptor.

31		10 9 8	5 4 3 2 1 0
Coarse page table base address		Domain	1	SBZ	0	1

SBZ
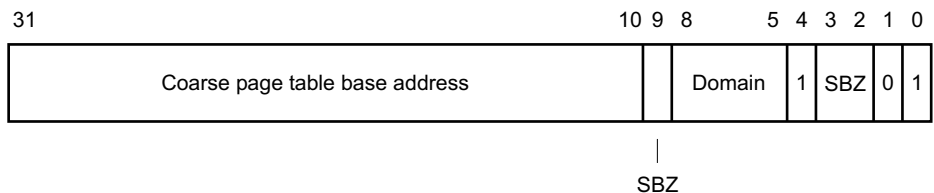
Figure 7-6 Coarse page table descriptor

--- **Note** ---

If a coarse page table descriptor is returned from the level one fetch, a level two fetch is initiated.

Coarse page table descriptor bit assignments are described in Table 7-5.

Table 7-5 Coarse page table descriptor bits

Bits	Description
31:10	These bits form the base for referencing the level two 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

7.3.6 Fine page table descriptor

A fine page table descriptor provides the base address of a page table that contains level two 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 7-7 shows the format of a fine page table descriptor.

31	12 11	9 8	5 4	3 2	1 0
Fine page table base address	SBZ	Domain	1	SBZ	1 1
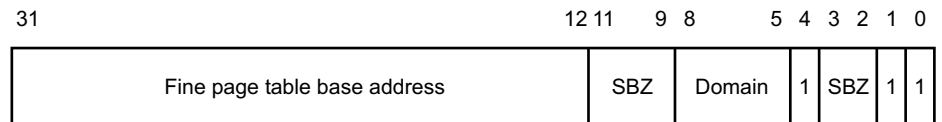
Figure 7-7 Fine page table descriptor

───── **Note** ─────

If a fine page table descriptor is returned from the level one fetch, a level two fetch is initiated.

─────────────

Fine page table descriptor bit assignments are described in Table 7-6.

Table 7-6 Fine page table descriptor bits

Bits	Description
31:12	These bits form the base for referencing the level two 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

7.3.7 Translating section references

Figure 7-8 on page 7-13 shows the complete section translation sequence.

 ARM DDI 0229A

Modified virtual address

31	20 19	0
Table index	Section index	

Translation table base

31	14 13	0
Translation base		

31	14 13	2 1 0
Translation base	Table index	0 0

Section level one descriptor

31	20 19	12 11 10 9 8	5 4 3 2 1 0
Section base address		AP Domain	1 C B 1 0

Physical address

31	20 19	0
Section base address	Section index	
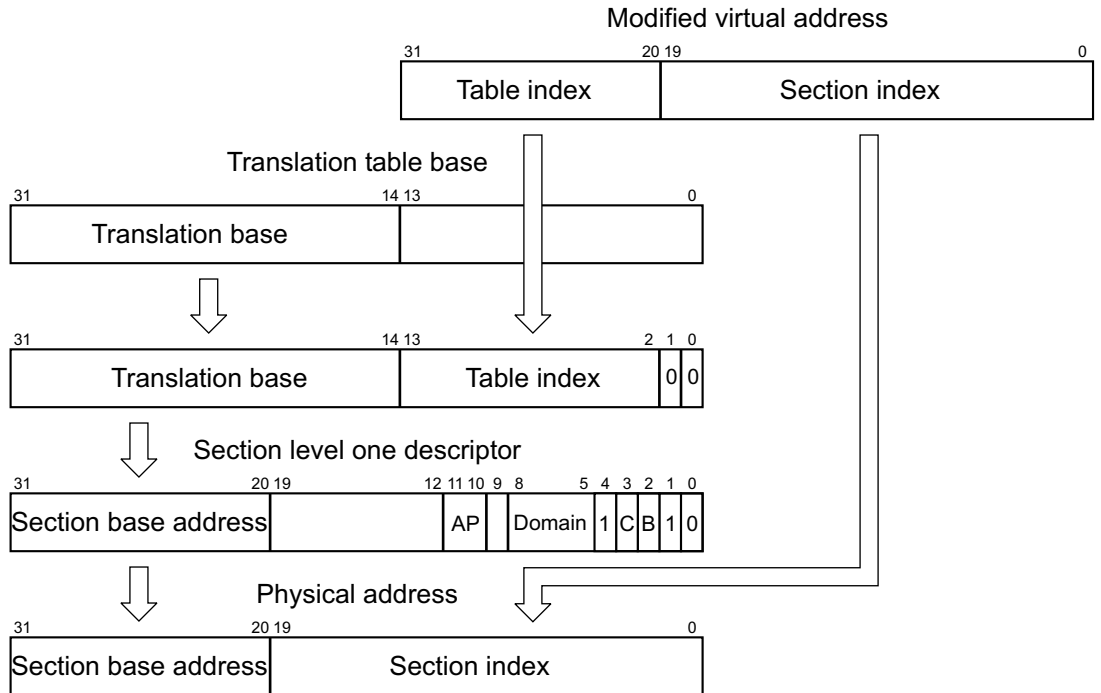
Figure 7-8 Section translation

——— **Note** ———

You must check access permissions contained in the level one descriptor before
generating the physical address.

7.3.8   Level two descriptor

If the level one 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 level two descriptor is returned. Figure 7-9 on page 7-14 shows the
format of level two descriptors.

Figure 7-9 Level two descriptor

A level two 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.

Level two descriptor bit assignments are described in Table 7-7.

Table 7-7 Level two 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 7-7 Level two descriptor bits (continued)

Bits			Description
Large	Small	Tiny	
11:4	11:4	5:4	Access permission bits. *Domain access control* on page 7-22 and *Fault checking sequence* on page 7-24 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 cachable or noncachable, and bufferable or nonbufferable. (The system is always write-through.)
1:0	1:0	1:0	These bits indicate the page size and validity and are interpreted as shown in Table 7-8

The two least significant bits of the level two descriptor indicate the descriptor type as shown in Table 7-8.

Table 7-8 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.

7.3.9 Translating large page references

Figure 7-10 on page 7-16 shows the complete translation sequence for a 64KB large page.
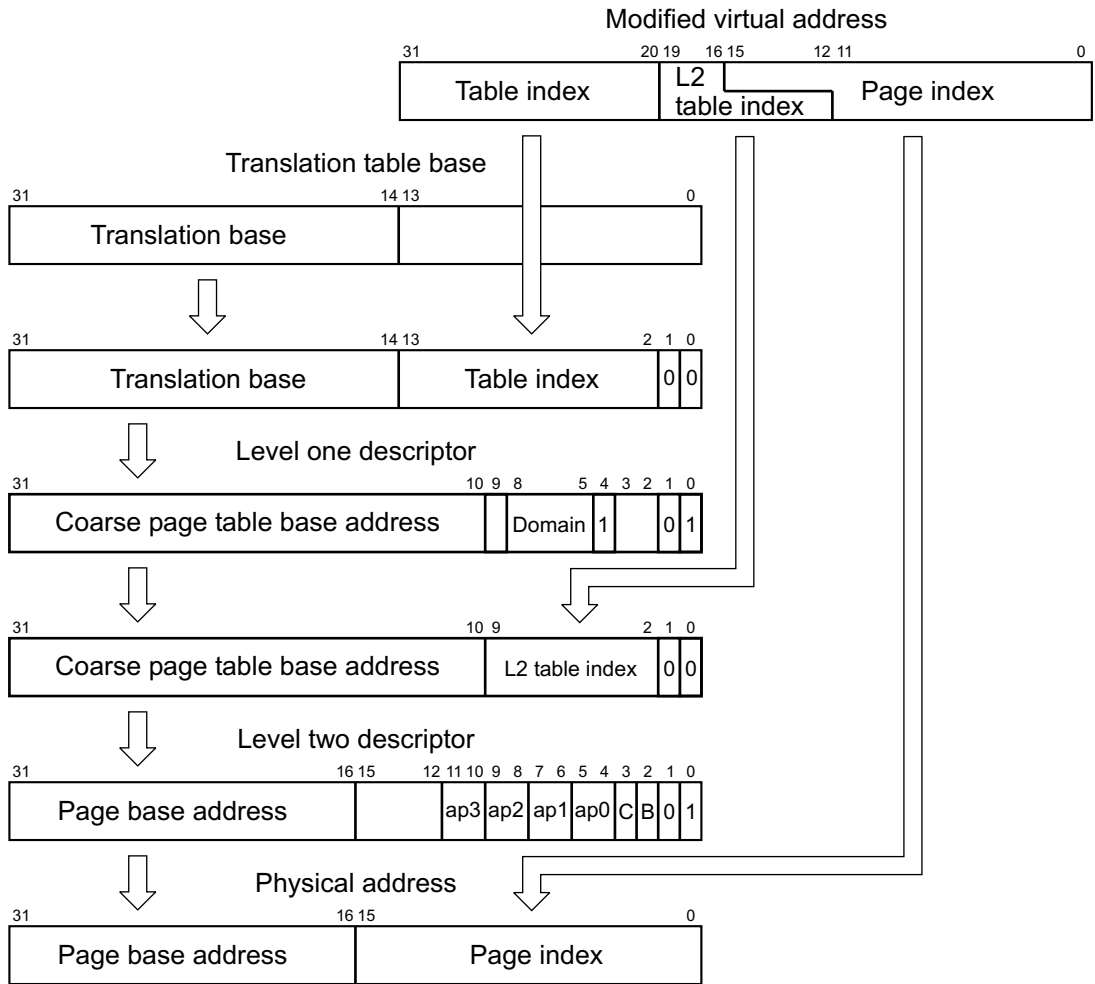
Figure 7-10 Large page translation from a coarse page table

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.

7.3.10   Translating small page references

Figure 7-11 shows the complete translation sequence for a 4KB small page.
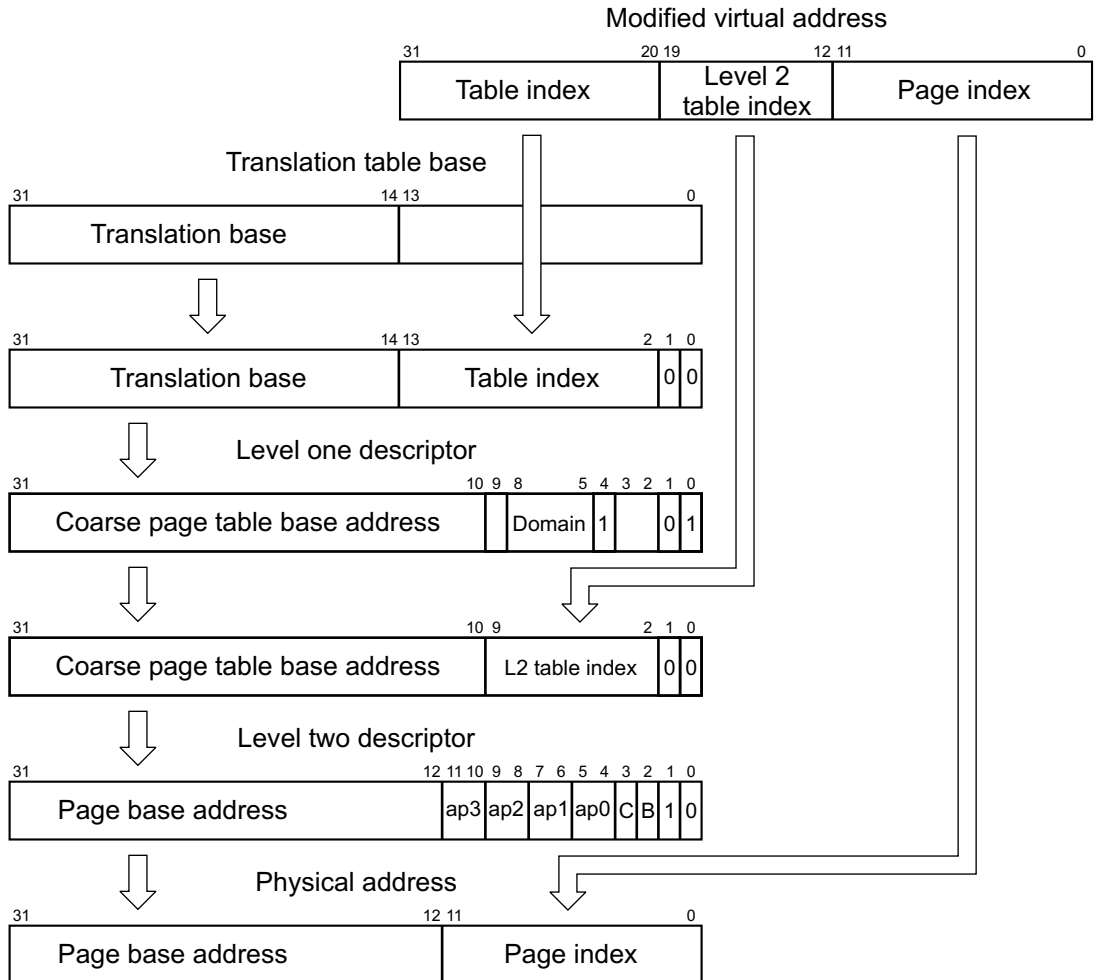
Figure 7-11 Small page translation from a coarse page table

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.

7.3.11   Translating tiny page references

Figure 7-12 shows the complete translation sequence for a 1KB tiny page.
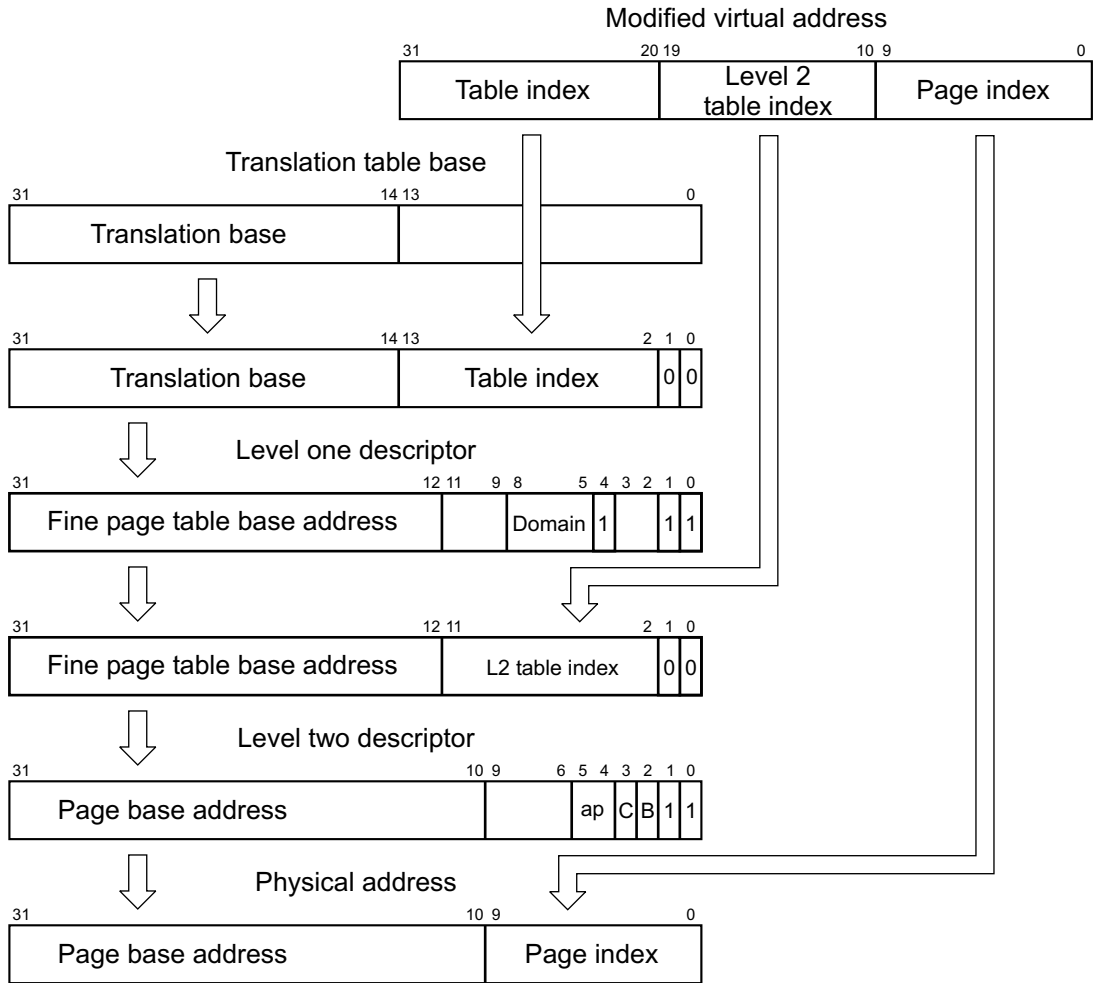
Figure 7-12 Tiny page translation from a fine page table

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

———— **Note** ————

The domain specified in the level one description and access permissions specified in the level one description together determine whether the access has permissions to proceed. See section *Domain access control* on page 7-22 for details.

7.3.12 Subpages

You can define access permissions for subpages of small and large pages. If, during a page walk, a small or large page has a non-identical 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.

7.4   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:

*   noncachable loads
*   nonbufferable writes.

Alignment fault checking is enabled by the A bit in CP15 register 1. 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 fault status register and fault address register (see *Fault address and fault status registers* on page 7-21).

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

7.5 Fault address and fault status registers

On an abort, the MMU places an encoded 4-bit value, FS[3:0], along with the 4-bit encoded domain number, in the data FSR, and the MVA associated with the 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 7-9.

7.5.1 Fault status

Table 7-9 describes the various access permissions and controls supported by the data MMU and details how these are interpreted to generate faults.

Table 7-9 Priority encoding of fault status

Priority	Source	Size	Status	Domain	FAR
Highest	Alignment	-	b00x1	Invalid	MVA of access causing abort
	Translation	Section	b0101	Invalid	MVA of access causing abort
		Page	b0111	Valid	
	Domain	Section	b1001	Valid	MVA of access causing abort
		Page	b1011	Valid	
	Permission	Section	b1101	Valid	MVA of access causing abort
		Page	b1111	Valid	
Lowest	External abort on noncachable nonbufferable access or noncachable bufferable read	Section	b1000	Valid	MVA of access causing abort
		Page	b1010	Valid	

———— **Note** ————

Alignment faults can write either b0001 or b0011 into FS[3:0]. Invalid values in domains [3:0] can occur because the fault is raised before a valid domain field has been read from a page table descriptor. Any abort masked by the priority encoding can be regenerated by fixing the primary abort and restarting the instruction.

7.6   Domain access control

MMU accesses are primarily controlled through the use of domains. There are 16 domains and each has a 2-bit field to define access to it. Two types of user are supported, clients and managers. The domains are defined in the domain access control register. Figure 7-13 shows how the 32 bits of the register are allocated to define the 16 2-bit domains.
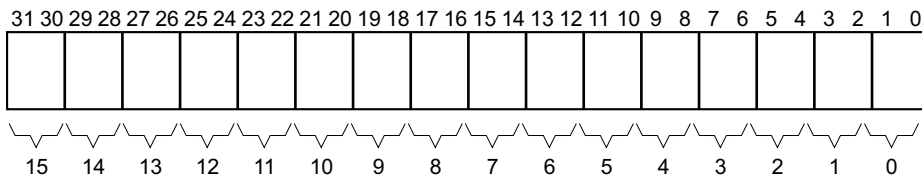
Figure 7-13 Domain access control register format

Table 7-10 defines how the bits within each domain are interpreted to specify the access permissions.

Table 7-10 Interpreting access control bits in domain access control register

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

              ARM DDI 0229A

Table 7-11 shows how to interpret the *Access Permission* (AP) bits and how their interpretation is dependent on the S and R bits (control register bits 8 and 9).

Table 7-11 Interpreting access permission (AP) bits

AP	S	R	Supervisor permissions	User permissions	Description
00	0	0	No access	No access	Any access generates a permission fault
00	1	0	Read-only	No access	Only Supervisor read permitted
00	0	1	Read-only	Read-only	Any write generates a permission fault
00	1	1	Reserved	-	_a
01	x	x	Read/write	No access	Access allowed only in Supervisor mode
10	x	x	Read/write	Read-only	Writes in User mode cause permission fault
11	x	x	Read/write	Read/write	All access types permitted in both modes
xx	1	1	Reserved	-	_a

a. Do not use this encoding. [S:R] = 11 generates a fault for any access.

7.7 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 7-14.
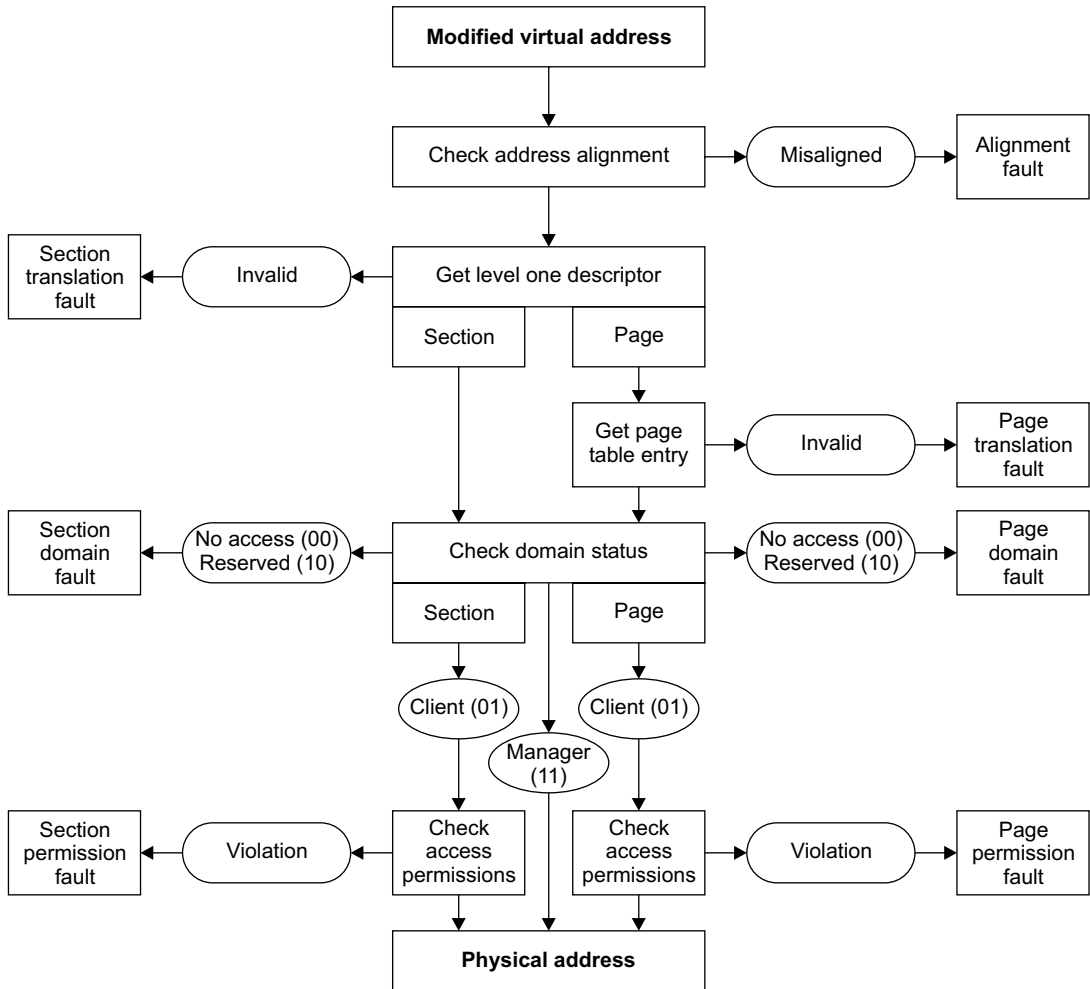
Figure 7-14 Sequence for checking faults

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

- *Alignment fault* on page 7-25
- *Translation fault* on page 7-25

- *Domain fault*
- *Permission fault* on page 7-26.

7.7.1 Alignment fault

If alignment fault is enabled (A bit in CP15 register 1 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, nor on any byte access.

—— **Note** ——

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

7.7.2 Translation fault

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 two descriptor is marked as invalid. This happens if bits [1:0] of the descriptor are both 0.

7.7.3 Domain fault

There are two types of domain fault:

Section The level one descriptor holds the 4-bit domain field, which selects one of the 16 2-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 7-11 on page 7-23. The domain is checked when the level one descriptor is returned.

Page The level one descriptor holds the 4-bit domain field, which selects one of the 16 2-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 7-11 on page 7-23. 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.

7.7.4   Permission fault

If the 2-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 7-11 on page 7-23. Their interpretation is dependent on the setting of the S and R bits (control register 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-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 7-11 on page 7-23). 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.

7.8   External aborts

In addition to the MMU-generated aborts, the ARM720T processor can be externally aborted by the AMBA bus. This can be used to flag an error on an external memory access. However, not all accesses can be aborted in this way and the *Bus Interface Unit* (BIU) ignores external aborts that cannot be handled.

The following accesses can be aborted:
*   noncached reads
*   unbuffered writes
*   read-lock-write sequence, to noncachable memory.

In the case of a read-lock-write (SWP) sequence, if the read aborts, the write is never attempted.

7.9 Interaction of the MMU and cache

The MMU is enabled and disabled using bit 0 of the CP15 control register as described in:

- *Enabling the MMU*
- *Disabling the MMU*.

7.9.1 Enabling the MMU

To enable the MMU:

1.   Program the TTB and domain access control registers.

2.   Program level 1 and level 2 page tables as required.

3.   Enable the MMU by setting bit 0 in the control register.

You must take care 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 (using physical = VA - flat translation).

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

MCR p15,0,R1,C1, C0,0            ; Enable MMUS

Fetch Flat

Fetch Flat

Fetch Translated
```

7.9.2 Disabling the MMU

To disable the MMU, clear bit 0 in the control register. The data cache must be disabled prior to, or at the same time as, the MMU is disabled by clearing bit 2 of the control register. See *Enabling the MMU* regarding prefetch effects.

───── **Note** ─────

If the MMU is enabled, then disabled and subsequently re-enabled, the contents of the TLB are preserved. If these are now invalid, you must invalidate the TLB before re-enabling the MMU. See *Register 8, TLB operations* on page 3-10.

Chapter 8
Coprocessor Interface

This chapter describes the coprocessor interface on the ARM720T processor. It contains the following sections:

- *About coprocessors* on page 8-2
- *Coprocessor interface signals* on page 8-4
- *Pipeline-following signals* on page 8-5
- *Coprocessor interface handshaking* on page 8-6
- *Connecting coprocessors* on page 8-11
- *Not using an external coprocessor* on page 8-13
- *STC operations* on page 8-14
- *Undefined instructions* on page 8-15
- *Privileged instructions* on page 8-16.

8.1   About coprocessors

The instruction set for the ARM720T processor enables you to implement specialized additional instructions using coprocessors. These are separate processing units that are tightly coupled to the ARM720T processor. A typical coprocessor contains:

- an instruction pipeline
- instruction decoding logic
- handshake logic
- a register bank
- special processing logic, with its own data path.

A coprocessor is connected to the same data bus as the ARM720T processor in the system, and tracks the pipeline in the ARM720T core. This means that the coprocessor can decode the instructions in the instruction stream, and execute those that it supports. Each instruction progresses down both the ARM720T processor pipeline and the coprocessor pipeline at the same time.

The execution of instructions is shared between the ARM720T core and the coprocessor, as follows:

The ARM720T core

1. Evaluates the condition codes to determine whether the instruction must be executed by the coprocessor, then signals this to any coprocessors in the system (using **CPnCPI**).

2. Generates any addresses that are required by the instruction, including prefetching the next instruction to refill the pipeline.

3. Takes the undefined instruction trap if no coprocessor accepts the instruction.

The coprocessor:

1. Decodes instructions to determine whether it can accept the instruction.

2. Indicates whether it can accept the instruction (by signaling on **EXTCPA** and **EXTCPB**).

3. Fetches any values required from its own register bank.

4. Performs the operation required by the instruction.

If a coprocessor cannot execute an instruction, the instruction takes the undefined instruction trap. You can choose whether to emulate coprocessor functions in software, or to design a dedicated coprocessor.

8.1.1 Coprocessor availability

You can connect up to 16 coprocessors into a system, each with a unique coprocessor ID number.

Some coprocessor numbers are reserved. For example, you cannot assign external coprocessors to coprocessor numbers 14 and 15, because these are internal to the ARM720T processor:

- CP14 is the communications channel coprocessor
- CP15 is the system control coprocessor for cache and MMU functions.

Coprocessor availability is shown in Table 8-1.

Table 8-1 Coprocessor availability

Coprocessor number	Allocation
15	System control
14	Debug controller
13:8	Reserved
7:4	Available to users
3:0	Reserved

——— **Note** ———

If you intend to design a coprocessor, send an E-mail with `coprocessor` in the subject line to `info@arm.com` for up to date information on coprocessor numbers that have already been allocated.

8.2   Coprocessor interface signals

The signals used to interface the ARM720T core to a coprocessor are grouped into four categories.

The clock and clock control signals include the main processor clock and bus reset:

*   **HCLK**
*   **EXTCPCLKEN**
*   **HRESETn**.

The pipeline-following signals are:

*   **CPnMREQ**
*   **CPnTRANS**
*   **CPnOPC**
*   **CPTBIT**.

The handshake signals are:

*   **CPnCPI**
*   **EXTCPA**
*   **EXTCPB**.

The data signals are:

*   **EXTCPDIN[31:0]**
*   **EXTCPDOUT[31:0]**
*   **EXTCPDBE**.

These signals and their use are described in:

*   *Pipeline-following signals* on page 8-5
*   *Coprocessor interface handshaking* on page 8-6
*   *Connecting coprocessors* on page 8-11
*   *Not using an external coprocessor* on page 8-13
*   *Undefined instructions* on page 8-15
*   *Privileged instructions* on page 8-16.

8.3   Pipeline-following signals

Every coprocessor in the system must contain a pipeline follower to track the instructions executing in the ARM720T processor pipeline. The coprocessors connect to the ARM720T processor input data bus, **RDATA[31:0]**, over which instructions are fetched, and to **HCLK** and **EXTCPCLKEN**.

It is essential that the two pipelines remain in step at all times. When designing a pipeline follower for a coprocessor, you must observe the following rules:

*   At reset (**HRESETn** LOW), the pipeline must either be marked as invalid, or filled with instructions that do not decode to valid instructions for that coprocessor.

*   The coprocessor state must only change when **EXTCPCLKEN** is HIGH (except for reset).

*   An instruction must be loaded into the pipeline on the rising edge of **HCLK**, and only when **CPnOPC**, **CPnMREQ**, and **CPTBIT** were *all* LOW in the previous bus cycle.

    These conditions indicate that this cycle is an ARM state opcode Fetch, so the new opcode must be sampled into the pipeline.

*   The pipeline must be advanced on the rising edge of **HCLK** when **CPnOPC**, **CPnMREQ**, and **CPTBIT** are all LOW in the current bus cycle.

    These conditions indicate that the current instruction is about to complete execution, because the first action of any instruction performing an instruction fetch is to refill the pipeline.

Any instructions that are flushed from the ARM720T processor pipeline never signal on **CPnCPI** that they have entered Execute, so they are automatically flushed from the coprocessor pipeline by the prefetches required to refill the pipeline.

There are no coprocessor instructions in the Thumb instruction set, so coprocessors must monitor the state of the **CPTBIT** signal to ensure that they do not try to decode pairs of Thumb instructions as ARM instructions.

8.4 Coprocessor interface handshaking

The ARM720T core and any coprocessors in the system perform a handshake using the signals shown in Table 8-2.

Table 8-2 Handshaking signals

Signal	Direction	Meaning
CPnCPI	ARM720T core to coprocessor	Not coprocessor instruction
EXTCPA	Coprocessor to ARM720T core	Coprocessor absent
EXTCPB	Coprocessor to ARM720T core	Coprocessor busy

These signals are explained in more detail in *Coprocessor signaling* on page 8-7.

8.4.1 The coprocessor

The coprocessor decodes the instruction currently in the Decode stage of its pipeline and checks whether that instruction is a coprocessor instruction. A coprocessor instruction has a coprocessor number that matches the coprocessor ID of the coprocessor.

If the instruction currently in the Decode stage is a coprocessor instruction:

1. The coprocessor attempts to execute the instruction.

2. The coprocessor signals back to the ARM720T core using **EXTCPA** and **EXTCPB**.

8.4.2 The ARM720T core

Coprocessor instructions progress down the ARM720T processor pipeline in step with the coprocessor pipeline. A coprocessor instruction is executed if the following are true:

1. The coprocessor instruction has reached the Execute stage of the pipeline. (It might not if it was preceded by a branch.)

2. The instruction has passed its conditional execution tests.

3. A coprocessor in the system has signalled on **EXTCPA** and **EXTCPB** that it is able to accept the instruction.

If all these requirements are met, the ARM720T processor signals by taking **CPnCPI** LOW. This commits the coprocessor to the execution of the coprocessor instruction.

8.4.3   Coprocessor signaling

The coprocessor signals as follows:

Coprocessor absent

If a coprocessor cannot accept the instruction currently in Decode it must leave **EXTCPA** and **EXTCPB** both HIGH.

Coprocessor present

If a coprocessor can accept an instruction, and can start that instruction immediately, it must signal this by driving both **EXTCPA** and **EXTCPB** LOW.

Coprocessor busy (busy-wait)

If a coprocessor can accept an instruction, but is currently unable to process that request, it can stall the ARM720T core by asserting busy-wait. This is signaled by driving **EXTCPA** LOW, but leaving **EXTCPB** HIGH. When the coprocessor is ready to start executing the instruction it signals this by driving **EXTCPB** LOW. This is shown in Figure 8-1.
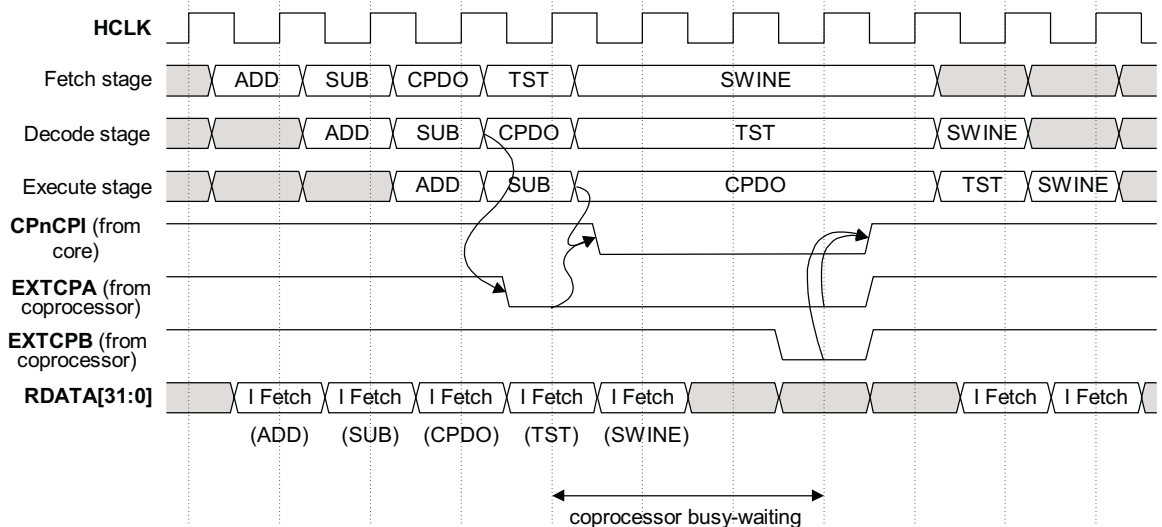
Figure 8-1 Coprocessor busy-wait sequence

8.4.4 Consequences of busy-waiting

A busy-waited coprocessor instruction can be interrupted. If a valid FIQ or IRQ occurs (the appropriate bit is cleared in the CSPR), the ARM720T processor abandons the coprocessor instruction, and signals this by taking **CPnCPI** HIGH. A coprocessor that is capable of busy-waiting must monitor **CPnCPI** to detect this condition. When the ARM720T core abandons a coprocessor instruction, the coprocessor also abandons the instruction and continues tracking the ARM720T processor pipeline.

──── **Caution** ────

It is essential that any action taken by the coprocessor while it is busy-waiting is idempotent. The actions taken by the coprocessor must not corrupt the state of the coprocessor, and must be repeatable with identical results. The coprocessor can only change its own state after the instruction has been executed.

8.4.5 Coprocessor register transfer instructions

The coprocessor register transfer instructions, MCR and MRC, transfer data between a register in the ARM720T processor register bank and a register in the coprocessor register bank. An example sequence for a coprocessor register transfer is shown in Figure 8-2.
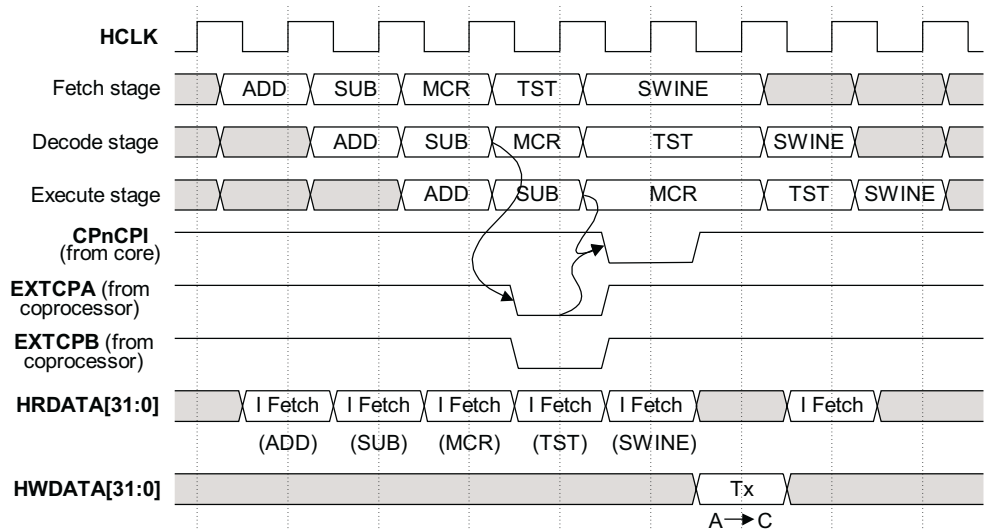
Figure 8-2 Coprocessor register transfer sequence

8.4.6 Coprocessor data operations

The coprocessor data processing instructions, CDP, perform processing operations on the data held in the coprocessor register bank. No information is transferred between the ARM720T core and the coprocessor as a result of this operation. An example sequence is shown in Figure 8-3.
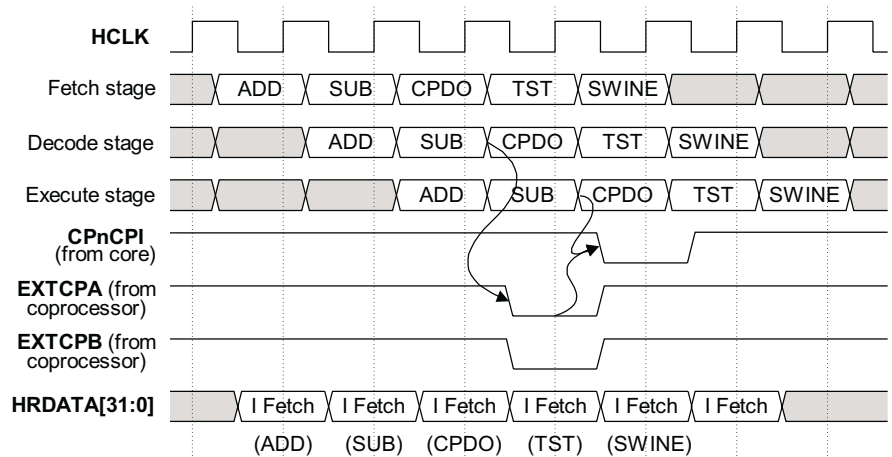
Figure 8-3 Coprocessor data operation sequence

8.4.7 Coprocessor load and store operations

The coprocessor load and store instructions, LDC and STC, are used to transfer data between a coprocessor and memory. They can be used to transfer either a single word of data or a number of the coprocessor registers. There is no limit to the number of words of data that can be transferred by a single LDC or STC instruction, but by convention a coprocessor must not transfer more than 16 words of data in a single instruction. An example sequence is shown in Figure 8-4 on page 8-10.

———— **Note** ————

If you transfer more than 16 words of data in a single instruction, the worst-case interrupt latency of the ARM720T processor increases.

———————————————
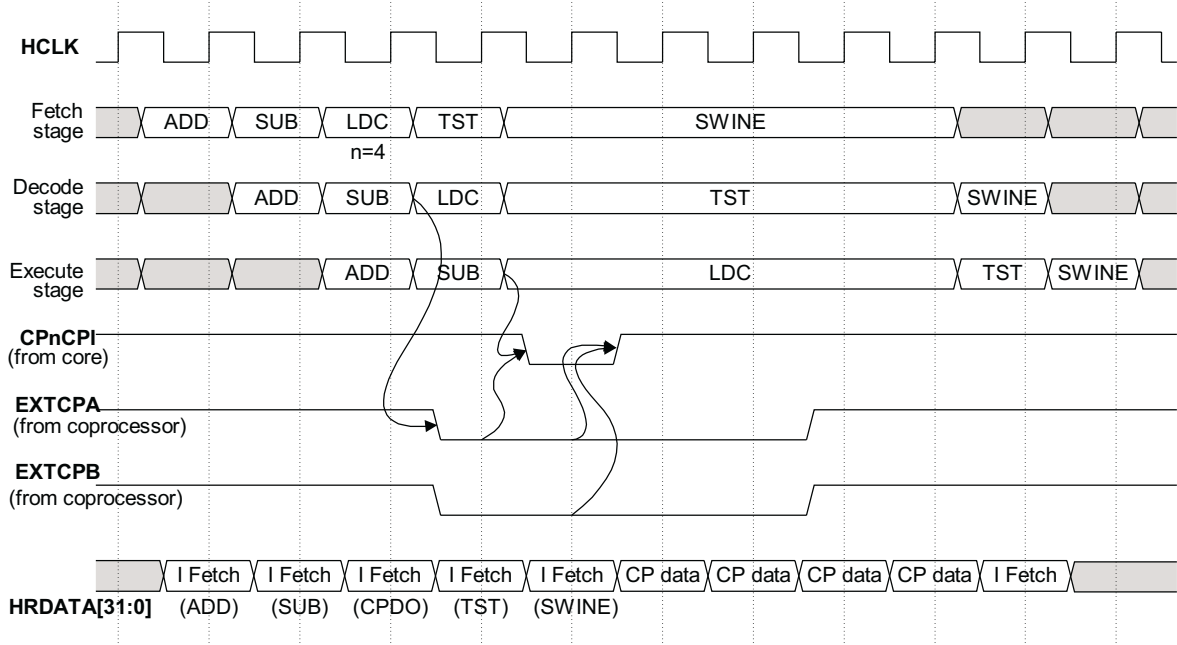
Figure 8-4 Coprocessor load sequence

 ARM DDI 0229A

8.5 Connecting coprocessors

A coprocessor in a system based on an ARM720T processor must have 32-bit connections to:

- transfer data from memory (instruction stream and LDC)
- write data from the ARM720T processor (MCR)
- read data to the ARM720T processor (MRC).

8.5.1 Connecting a single coprocessor

You can connect a single coprocessor directly to the coprocessor interface of the ARM720T processor without any additional logic, as shown in Figure 8-5. **EXTCPDBE** must be driven HIGH by the external coprocessor when it drives data on **EXTCPDOUT**.
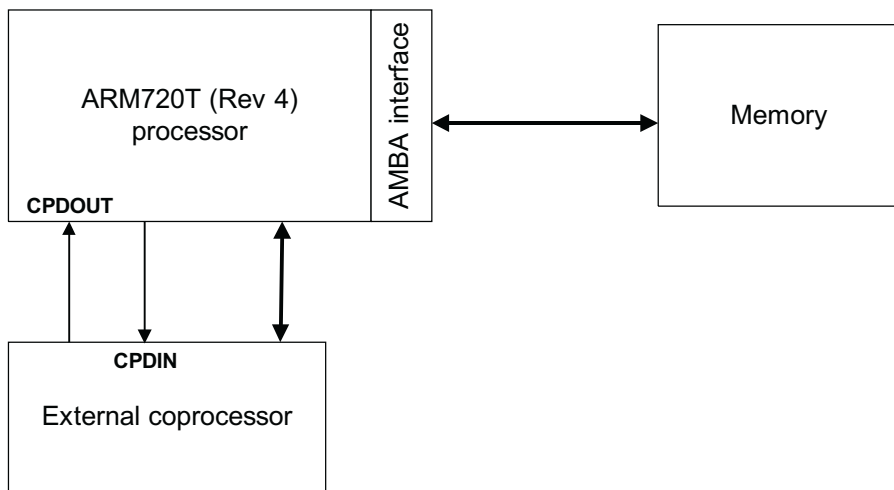
Figure 8-5 Example coprocessor connections

—— **Note** ——

If you are building a system with an ETM7 and an ARM720T core, you must directly connect the following buses:

- ETM7 input **RDATA[31:0]** to the ARM720T processor output **ETMRDATA[31:0]**

- ETM7 input **WDATA[31:0]** to the ARM720T processor output **ETMWDATA[31:0]**.

This enables the ETM to correctly trace coprocessor instructions.

8.5.2 Connecting multiple coprocessors

If you have multiple coprocessors in your system, connect the handshake signals as shown in Table 8-3.

Table 8-3 Handshake signal connections

Signal	Connection
CPnCPI	Connect this signal to all coprocessors present in the system
CPA and **CPB**	The individual **CPA** and **CPB** outputs from each coprocessor must be ANDed together, and connected to the **EXTCPA** and **EXTCPB** inputs on the ARM720T processor

You must also multiplex the output data from the coprocessors.

8.6   Not using an external coprocessor

If you are implementing a system that does not include any external coprocessors, you must tie both **EXTCPA** and **EXTCPB** HIGH. This indicates that no external coprocessors are present in the system. If any coprocessor instructions are received, they take the undefined instruction trap so that they can be emulated in software if required.

The coprocessor-specific outputs from the ARM720T core must be left unconnected:

- **CPnMREQ**
- **CPnTRANS**
- **CPnOPC**
- **CPnCPI**
- **CPTBIT**.

You must tie off **EXTCPDOUT**.

You must tie the external coprocessor data bus enable, **EXTCPDBE**, LOW.

8.7 STC operations

If you are using an external coprocessor, you can perform STC operations in cachable regions with the cache enabled. However, the STC operation is treated as a series of nonsequential transfers on the AMBA bus.

8.8 Undefined instructions

The ARM720T processor implements full ARM architecture v4T undefined instruction handling. This means that any instruction defined in the *ARM Architecture Reference Manual* as UNDEFINED, automatically causes the ARM720T processor to take the undefined instruction trap. Any coprocessor instructions that are not accepted by a coprocessor also result in the ARM720T processor taking the undefined instruction trap.

8.9   Privileged instructions

The output signal **CPnTRANS** enables you to implement coprocessors, or coprocessor instructions, that can only be accessed from privileged modes. The signal meanings are shown in Table 8-4.

Table 8-4 CPnTRANS signal meanings

CPnTRANS	Meaning
LOW	User mode instruction
HIGH	Privileged mode instruction

The **CPnTRANS** signal is sampled at the same time as the instruction, and is factored into the coprocessor pipeline Decode stage.

——— **Note** ———

If a User-mode process (**CPnTRANS** LOW) tries to access a coprocessor instruction that can only be executed in a privileged mode, the coprocessor must respond with **EXTCPA** and **EXTCPB** HIGH. This causes the ARM720T processor to take the undefined instruction trap.

Chapter 9
Debugging Your System

This chapter describes how to debug a system based on an ARM720T (Rev 4) processor. It contains the following sections:

- *The program counter during debug* on page 9-46
- *Priorities and exceptions* on page 9-50
- *Watchpoint unit registers* on page 9-51
- *Programming breakpoints* on page 9-56
- *Programming watchpoints* on page 9-59
- *Abort status register* on page 9-61
- *Debug control register* on page 9-62
- *Debug status register* on page 9-65
- *Coupling breakpoints and watchpoints* on page 9-67
- *EmbeddedICE-RT timing* on page 9-70.

9.1 About debugging your system

The advanced debugging features of the ARM720T (Rev 4) processor make it easier to develop application software, operating systems, and the hardware itself.

9.1.1 A typical debug system

The ARM720T (Rev 4) processor forms one component of a debug system that interfaces from the high-level debugging that you perform to the low-level interface supported by the ARM720T processor. Figure 9-1 shows a typical debug system.
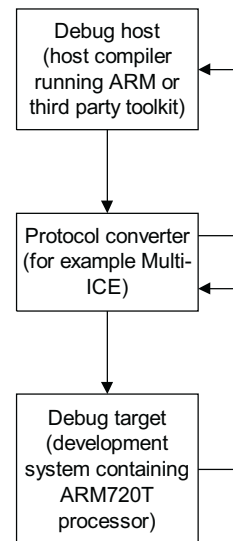
Figure 9-1 Typical debug system

A debug system usually has three parts:

Debug host   A computer that is running a software debugger such as the *ARM Debugger for Windows* (ADW). The debug host enables you to issue high-level commands such as setting breakpoints or examining the contents of memory.

Protocol converter This interfaces between the high-level commands issued by the debug host and the low-level commands of the ARM720T processor JTAG interface. Typically it interfaces to the host through an interface such as an enhanced parallel port.

Debug target   The ARM720T (Rev 4) processor has hardware extensions that ease debugging at the lowest level. These extensions enable you to:

- halt program execution
- examine and modify the internal state of the core
- examine the state of the memory system
- execute abort exceptions, allowing real-time monitoring of the core
- resume program execution.

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

       ARM DDI 0229A

9.2 Controlling debugging

The major blocks of the ARM720T (Rev 4) processor are:

ARM CPU core   This has hardware support for debug.

EmbeddedICE-RT macrocell

A set of registers and comparators that you use to generate debug exceptions (such as breakpoints). This unit is described in *The EmbeddedICE-RT macrocell* on page 9-14.

TAP controller   Controls the action of the scan chains using a JTAG serial interface. For more details, see *The TAP controller* on page 9-27.

These blocks are shown in Figure 9-2.
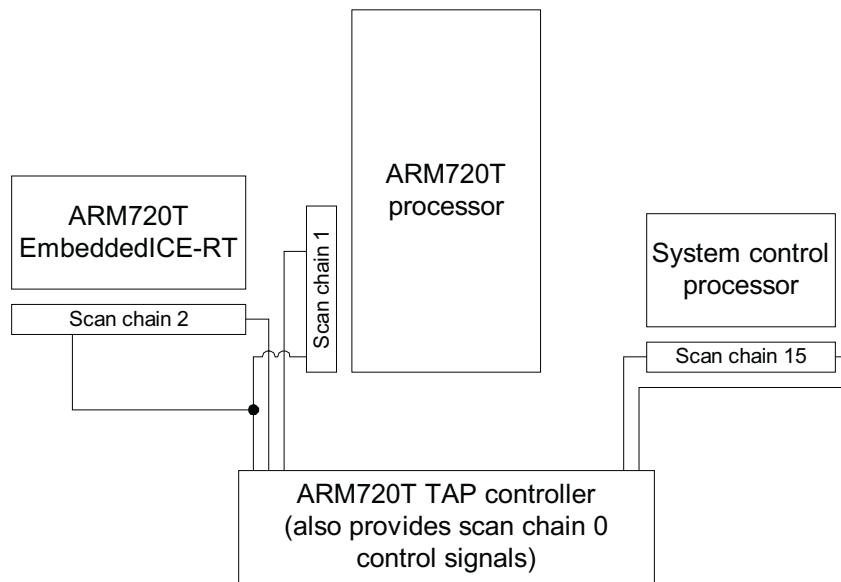
Figure 9-2 ARM720T (Rev 4) processor block diagram

9.2.1 Debug modes

You can perform debugging in either of the following modes:

Halt mode   When the system is in halt mode, the core enters *debug state* when it encounters a breakpoint or a watchpoint. In debug state, the core is stopped and isolated from the rest of the system. When debug has completed, the debug host restores the core and system state, and program execution resumes.

For more information, see *Entry into debug state* on page 9-7.

Monitor mode   When the system is in monitor mode, the core does not enter debug state on a breakpoint or watchpoint. Instead, an Instruction Abort or Data Abort is generated and the core continues to receive and service interrupts as normal. You can use the abort status register to establish whether the exception was due to a breakpoint or watchpoint, or to a genuine memory abort.

For more information, see *Monitor mode debugging* on page 9-18.

9.2.2 Examining system state during debugging

In both halt mode and monitor mode, the JTAG-style serial interface enables you to examine the internal state of the core and the external state of the system while system activity continues.

In halt mode, this enables instructions to be inserted serially into the core pipeline without using the external data bus. For example, when in debug state, a *Store Multiple* (STM) can be inserted into the instruction pipeline to export the contents of the ARM720T processor registers. This data can be serially shifted out without affecting the rest of the system. For more information, see *Examining the core and the system in debug state* on page 9-40.

In monitor mode, the JTAG interface is used to transfer data between the debugger and a simple monitor program running on the ARM720T core.

For detailed information about the scan chains and the JTAG interface, see *Scan chains and the JTAG interface* on page 9-24.

                   ARM DDI 0229A

9.3 Entry into debug state

If the system is in halt mode, any of the following types of interrupt force the processor into debug state:

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

——— **Note** ———

In monitor mode, the processor continues to execute instructions in real time, and will take an abort exception. The abort status register enables you to establish whether the exception was due to a breakpoint or watchpoint, or to a genuine memory abort.

You can use the EmbeddedICE-RT logic to program the conditions under which a breakpoint or watchpoint can occur. Alternatively, you can use the **DBGBREAK** signal to enable external logic to flag breakpoints or watchpoints and monitor the following:

- address bus
- data bus
- control signals.

The timing is the same for externally-generated breakpoints and watchpoints. Data must always be valid around the rising edge of **HCLK**. When this data is an instruction to be breakpointed, the **DBGBREAK** signal must be HIGH around the rising edge of **HCLK**. Similarly, when the data is for a load or store, asserting **DBGBREAK** around the rising edge of **HCLK** marks the data as watchpointed.

When a breakpoint or watchpoint is generated, there might be a delay before the ARM720T core enters debug state. When it enters debug state, the **DBGACK** signal is asserted. The timing for an externally-generated breakpoint is shown in Figure 9-3 on page 9-8.
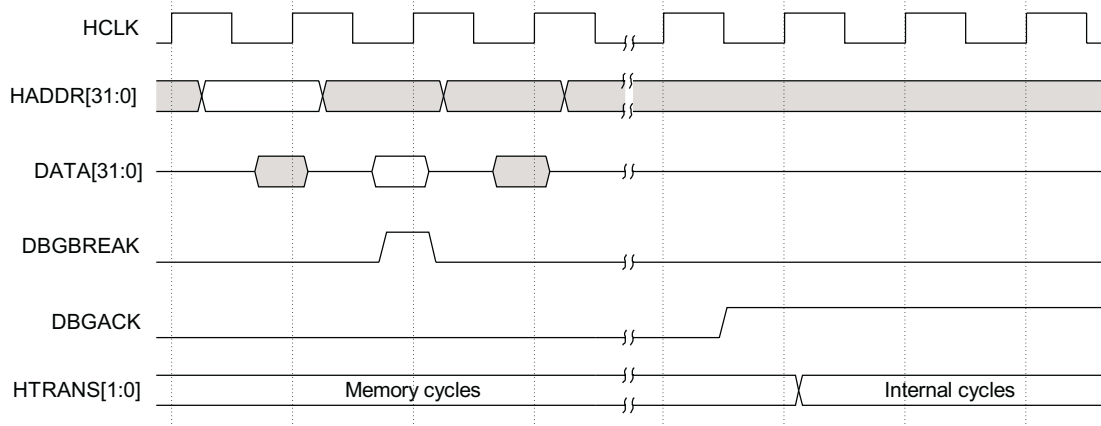
Figure 9-3 Debug state entry

9.3.1   Entry into debug state on breakpoint

The ARM720T (Rev 4) processor marks instructions as being breakpointed as they enter the instruction pipeline, but the core does not enter debug state until the instruction reaches the Execute stage.

Breakpointed instructions are not executed. Instead, the ARM720T core enters debug state. When you examine the internal state, you see the state before the breakpointed instruction. When your examination is complete, remove the breakpoint. Program execution restarts from the previously-breakpointed instruction.

When a breakpointed conditional instruction reaches the Execute stage of the pipeline, the breakpoint is always taken if the system is in halt mode. The ARM720T core enters debug state regardless of whether the instruction condition is met.

A breakpointed instruction does not cause the ARM720T core to enter debug state when:

- A branch or a write to the PC precedes the breakpointed instruction. In this case, when the branch is executed, the ARM720T processor flushes the instruction pipeline, so canceling the breakpoint.

- An exception occurs, causing the ARM720T processor to flush the instruction pipeline, and cancel the breakpoint. In normal circumstances, on exiting from an exception, the ARM720T core branches back to the instruction that would have been executed next before the exception occurred. In this case, the pipeline is refilled and the breakpoint is reflagged.

9.3.2 Entry into debug state on watchpoint

Watchpoints occur on data accesses. In halt mode, the core processing stops. In monitor mode, an abort exception is executed (see *Abort* on page 2-19). A watchpoint is always taken, but a core in halt mode might not enter debug state immediately because the current instruction always completes. If the current instruction is a multiword load or store (an LDM or STM), many cycles can elapse before the watchpoint is taken.

On a watchpoint, the following sequence occurs:

1. The current instruction completes.
2. All changes to the core state are made.
3. Load data is written into the destination registers.
4. Base write-back is performed.

——— **Note** ———

Watchpoints are similar to Data Aborts. The difference is that when a Data Abort occurs, although the instruction completes, the ARM720T core prevents all subsequent changes to the ARM720T processor state. This action enables the abort handler to cure the cause of the abort, so the instruction can be re-executed.

If a watchpoint occurs when an exception is pending, the core enters debug state in the same mode as the exception.

9.3.3 Entry into debug state on debug request

An ARM720T core in halt mode can be forced into debug state on debug request in either of the following ways:

• through EmbeddedICE-RT programming (see *Programming breakpoints* on page 9-56, and *Programming watchpoints* on page 9-59.)

• by asserting the **DBGRQ** pin.

When the **DBGRQ** pin has been asserted, the core normally enters debug state at the end of the current instruction. However, when the current instruction is a busy-waiting access to a coprocessor, the instruction terminates, and the ARM720T core enters debug state immediately. This is similar to the action of **nIRQ** and **nFIQ**.

9.3.4 Action of the ARM720T processor in debug state

When the ARM720T processor enters debug state, the core forces **HTRANS[1:0]** to indicate internal cycles. This action enables the rest of the memory system to ignore the ARM720T core and to function as normal. Because the rest of the system continues to operate, the ARM720T core is forced to ignore aborts and interrupts.

———— **Caution** ————

Do not reset the core while debugging, otherwise the debugger loses track of the core.

———— **Note** ————

The system must not change the **ETMBIGEND** signal during debug. From the point of view of the programmer, if **ETMBIGEND** changes, the ARM720T processor changes, with the debugger unaware that the core has reset. You must also ensure that **HRESETn** is held stable during debug. When the system applies reset to the ARM720T processor (that is, **HRESETn** is driven LOW), the ARM720T processor state changes with the debugger unaware that the core has reset.

9.3.5   Clocks

The system and test clocks must be synchronized externally to the macrocell. The ARM Multi-ICE debug agent directly supports one or more cores within an ASIC design. Synchronizing off-chip debug clocking with the ARM720T macrocell requires a three-stage synchronizer. The off-chip device (for example, Multi-ICE) issues a **TCK** signal and waits for the **RTCK** (Returned **TCK**) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next **TCK** until after **RTCK** is received.

Figure 9-4 on page 9-11 shows this synchronization.
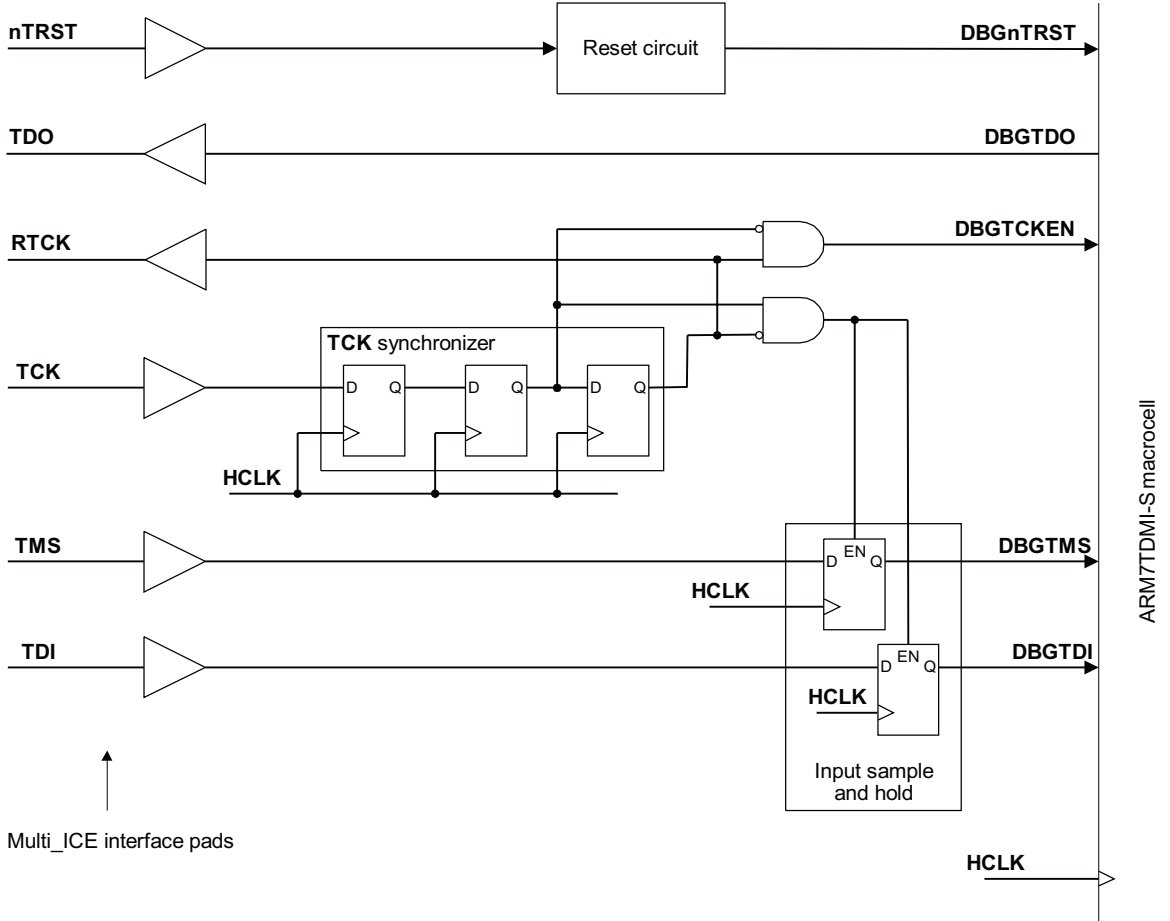
Figure 9-4 Clock synchronization

――――― **Note** ―――――
All the D-types shown in Figure 9-4 are reset by **DBGnTRST**.

9.4 Debug interface

The ARM720T processor debug interface is based on IEEE Std. 1149.1- 1990, *Standard Test Access Port and Boundary-Scan Architecture*. Refer to this standard for an explanation of the terms used in this chapter, and for a description of the TAP controller states.

9.4.1 Debug interface signals

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

*   **DBGBREAK** and **DBGRQ** are system requests for the ARM720T core to enter debug state

    ———— **Note** ————

    Both **DBGRQ** and **DBBREAK** must be LOW when the core has entered debug state. If they are not, these signals affect the use of the **DBGBREAK** flag on scan chain 1, which controls the way the core goes into and out of debug. The result is that the core performs an unexpected series of debug and system speed accesses, and the debugger loses control of the core.

    —————————————

*   **DBGACK** is used by the ARM720T core to flag back to the system that it is in debug state.

9.5 ARM720T core clock domains

The ARM720T processor has a single clock, **HCLK**, that is qualified by two clock enables:

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

When the ARM720T processor is in debug state, **DBGTCKEN** conditions **HCLK** to clock the core.

9.6   The EmbeddedICE-RT macrocell

The ARM720T processor EmbeddedICE-RT macrocell module provides integrated on-chip debug support for the ARM720T core.

The EmbeddedICE-RT module is connected directly to the core and therefore functions on the virtual address of the processor after relocation by the FCSE PID. You program the EmbeddedICE-RT macrocell serially using the ARM720T processor TAP controller.

Figure 9-5 shows the relationship between the core, EmbeddedICE-RT, and the TAP controller, showing only the signals that are pertinent to EmbeddedICE-RT.
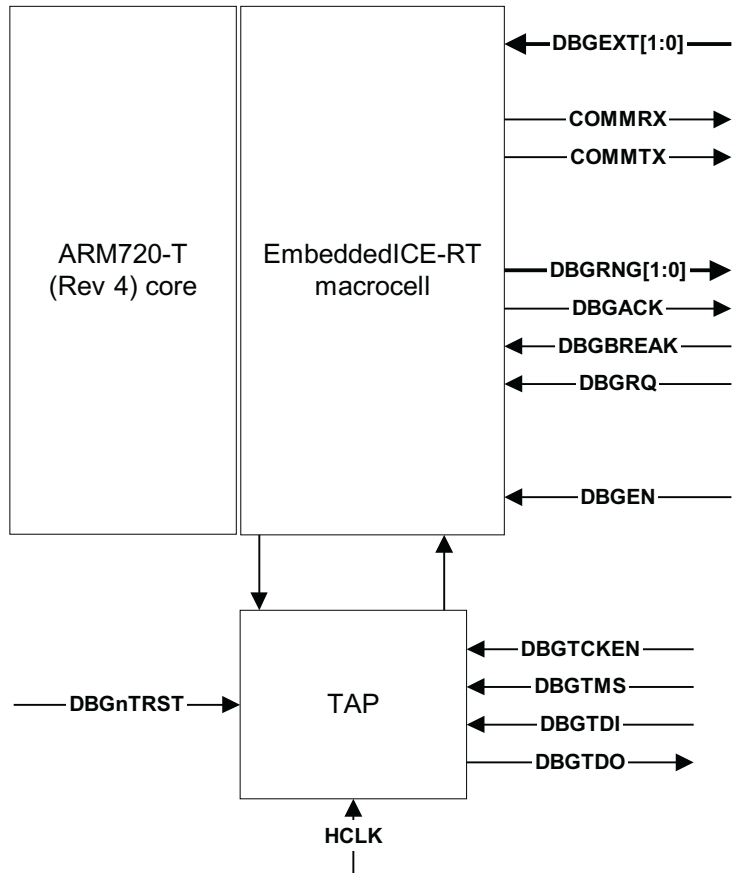
Figure 9-5 The ARM720T (Rev 4) core, TAP controller, and EmbeddedICE-RT macrocell

The EmbeddedICE-RT logic comprises the following:

Two real-time watchpoint units

You can program one or both watchpoint units to halt the execution of instructions by the core. Execution halts when the values programmed into the EmbeddedICE-RT logic match the values currently appearing on the address bus, data bus, and various control signals. You can mask any bit so that its value does not affect the comparison.

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

For more details, see *Watchpoint unit registers* on page 9-51.

Abort status register

This register identifies whether an abort exception entry was caused by a breakpoint, a watchpoint, or a real abort. For more information, see *Abort status register* on page 9-61.

Debug Communications Channel (DCC)

The DCC passes information between the target and the host debugger. For more information, see *The debug communications channel* on page 9-20.

In addition, two independent registers provide overall control of EmbeddedICE-RT operation. These are described in the following sections:

*   *Debug control register* on page 9-62
*   *Debug status register* on page 9-65.

The locations of the EmbeddedICE-RT registers are given in *EmbeddedICE-RT register map* on page 9-17.

9.7 Disabling EmbeddedICE-RT

You can disable EmbeddedICE-RT in two ways:

Permanently   By wiring the **DBGEN** input LOW.

When **DBGEN** is LOW:

- **DBGBREAK** and **DBGRQ** are ignored by the core
- **DBGACK** is forced LOW by the ARM720T core
- the **IFEN** input to the core is forced HIGH, so interrupts pass through to the processor uninhibited
- the EmbeddedICE-RT logic enters low-power mode.

———— **Caution** ————

Hard-wiring the **DBGEN** input LOW permanently disables debug access. However, you must not rely on this for system security.

Temporarily   By setting bit 5 in the debug control register (described in *Debug control register* on page 9-62). Bit 5 is also known as the EmbeddedICE-RT disable bit.

You must set bit 5 before doing either of the following:

- programming breakpoint or watchpoint registers
- changing bit 4 of the debug control register.

9.7.1 EmbeddedICE-RT timing

EmbeddedICE-RT samples the **DBGEXT[1]** and **DBGEXT[0]** inputs on the falling edge of **HCLK**. This means that you must allow sufficient set-up and hold time for these signals.

9.8   EmbeddedICE-RT register map

The locations of the EmbeddedICE-RT registers are shown in Table 9-1.

Table 9-1 Function and mapping of EmbeddedICE-RT registers

Address	Width	Function
b00000	6	Debug control
b00001	5	Debug status
b00100	32	Debug Communications Channel (DCC) control register
b00101	32	Debug Communications Channel (DCC) data register
b01000	32	Watchpoint 0 address value
b01001	32	Watchpoint 0 address mask
b01010	32	Watchpoint 0 data value
b01011	32	Watchpoint 0 data mask
b01100	9	Watchpoint 0 control value
b01101	8	Watchpoint 0 control mask
b10000	32	Watchpoint 1address value
b10001	32	Watchpoint 1 address mask
b10010	32	Watchpoint 1 data value
b10011	32	Watchpoint 1 data mask
b10100	9	Watchpoint 1 control value
b10101	8	Watchpoint 1 control mask

9.9 Monitor mode debugging

The ARM720T (Rev 4) processor contains logic that enables the debugging of a system without stopping the core entirely. This means that critical interrupt routines continue to be serviced while the core is being interrogated by the debugger.

9.9.1 Enabling monitor mode

The debugging mode is controlled by bit 4 of the debug control register (described in *Debug control register* on page 9-62). Bit 4 of this register is also known as the monitor mode enable bit:

Bit 4 set Enables the monitor mode features of the ARM720T processor. When this bit is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the ARM720T core to enter abort mode, taking the Prefetch or Data Abort vectors respectively.

Bit 4 clear Monitor mode debugging is disabled and the system is placed into halt mode. In halt mode, the core enters debug state when it encounters a breakpoint or watchpoint.

9.9.2 Restrictions on monitor-mode debugging

There are several restrictions you must be aware of when the ARM core is configured for monitor-mode debugging:

- Breakpoints and watchpoints cannot be data-dependent in monitor mode. No support is provided for use of the range functionality. Breakpoints and watchpoints can only be based on the following:
  — instruction or data addresses
  — external watchpoint conditioner (**DBGEXT[0]** or **DBGEXT[1]**)
  — User or privileged mode access (**CPnTRANS**)
  — read/write access for watchpoints (**HWRITE**)
  — access size (watchpoints **SIZE[1:0]**).

- External breakpoints or watchpoints are not supported.

- No support is provided to mix halt mode and monitor mode functionality.

The fact that an abort has been generated by the monitor mode is recorded in the abort status register in coprocessor 14 (see *Abort status register* on page 9-61).

The monitor mode enable bit does not put the ARM720T processor into debug state. For this reason, it is necessary to change the contents of the watchpoint registers while external memory accesses are taking place, rather than changing them when in debug state where the core is halted.

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

1.   Disable the watchpoint unit by setting bit 5 in the debug control register (also known as the EmbeddedICE-RT disable bit).

2.   Poll the debug control register until the EmbeddedICE-RT disable bit is read back as set.

3.   Change the other registers.

4.   Re-enable the watchpoint unit by clearing the EmbeddedICE-RT disable bit in the debug control register.

See *Debug control register* on page 9-62 for more information about controlling core behavior at breakpoints and watchpoints.

9.10 The debug communications channel

The ARM720T (Rev 4) EmbeddedICE-RT macrocell contains a *Debug Communication Channel* (DCC) for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The DCC comprises two registers, as follows:

DCC control register

A 32-bit register, used for synchronized handshaking between the processor and the asynchronous debugger. For more details, see *DCC control register*.

DCC data register

A 32-bit register, used for data transfers between the debugger and the processor. For more details, see *Communications through the DCC* on page 9-22.

These registers occupy fixed locations in the EmbeddedICE-RT memory map, as shown in Table 9-1 on page 9-17. They are accessed from the processor using `MCR` and `MRC` instructions to coprocessor 14.

The registers are accessed as follows:

By the debugger   Through scan chain 2 in the usual way.

By the processor   Through coprocessor register transfer instructions.

9.10.1 DCC control register

The DCC control register is read-only and enables synchronized handshaking between the processor and the debugger. The register format is shown in Figure 9-6.

31 30 29 28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02	01	00
0 0 0 1		W	R
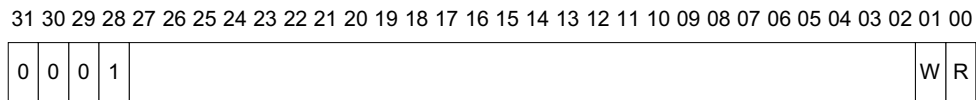
Figure 9-6 DCC control register

The DCC control register bit assignments are shown in Table 9-2.

Table 9-2 DCC control register bit assignments

Bit	Function
31:28	Contain a fixed pattern that denotes the EmbeddedICE-RT version number, in this case b0001.
27:2	Reserved.
1	The write control bit. If this bit is clear, the DCC data write register is ready to accept data from the processor. If this bit is set, there is data in the DCC data write register and the debugger can scan it out.
0	The read control bit. If this bit is clear, the DCC data read register is ready to accept data from the debugger. If this bit is set, the DCC data read register contains new data that has not been read by the processor, and the debugger must wait.

——— **Note** ———

If execution is halted, bit 0 might remain asserted. The debugger can clear it by writing to the DCC control register.

Writing to this register is rarely necessary, because in normal operation the processor clears bit 0 after reading it.

Instructions

The following instructions must be used:

```
MRC CP14, 0, Rd, C0, C0
```

Returns the value from the DCC control register into the destination register Rd.

```
MCR CP14, 0, Rn, C1, C0
```

Writes the value in the source register Rn to the DCC data write register.

```
MRC CP14, 0, Rd, C1, C0
```

Returns the value from the DCC data read register into the destination register Rd.

—— **Note** ——

The Thumb instruction set does not contain coprocessor instructions, so it is recommended that these are accessed using SWI instructions when in Thumb state.

9.10.2 Communications through the DCC

Messages can be sent and received through the DCC.

Sending a message to the debugger

Messages are sent from the processor to the debugger as follows:

1. When the processor wishes to send a message to EmbeddedICE-RT, it first checks that the communications data write register is free for use. The processor does this by reading the DCC control register to check the status of the W bit:

   a. If the W bit is clear, the DCC data write register is empty and a message is written by a register transfer to the coprocessor.

   b. If the W bit is set, this implies that previously-written data has not been read by the debugger. The processor must repeatedly read the DCC control register until the W bit is clear.

2. When the W bit is clear, a message is written by a register transfer to coprocessor 14. The data transfer occurs from the processor to the DCC data write register, so the W bit is set in the DCC control register.

3. When the debugger reads the DCC control register through the JTAG interface, it sees a synchronized version of both the R and W bits:

   a. When the debugger sees that the W bit is set, it can read the communications data write register and scan the data out.

   b. The action of reading this data register clears the W bit of the DCC control register. At this point, the communications process can begin again.

Receiving a message from the debugger

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

The sequence for receiving messages from the debugger is as follows:

1. The debugger reads the R bit of the DCC control register:

    a. If the R bit is clear, the data read register is free, and data can be placed there for the processor to read.

    b. If the R bit is set, previously-deposited data has not yet been collected, so the debugger must wait.

2. When the communications data read register is free, data is written there using the JTAG interface. The action of this write sets the R bit in the DCC control register.

3. The processor reads the DCC control register:

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

    b. If the R bit is clear, this indicates that the data has been taken and the process can now be repeated.

9.11   Scan chains and the JTAG interface

There are three JTAG-style scan chains within the ARM720T (Rev 4) processor. These allow debugging and EmbeddedICE-RT programming.

A JTAG-style *Test Access Port* (TAP) controller controls the scan chains. For more details of the JTAG specification, see IEEE Standard 1149.1 - 1990 *Standard Test Access Port and Boundary-Scan Architecture*.

9.11.1   Scan chain implementation

The three scan paths on the ARM720T (Rev 4) processor are referred to as scan chain 1, scan chain 2, and scan chain 15. They are shown in Figure 9-7.

Debug scan chain 0 is not implemented in the ARM720T (Rev 4) processor, but all the control signals are provided at the macrocell boundary. This enables you to design your own boundary scan chain wrapper if required.
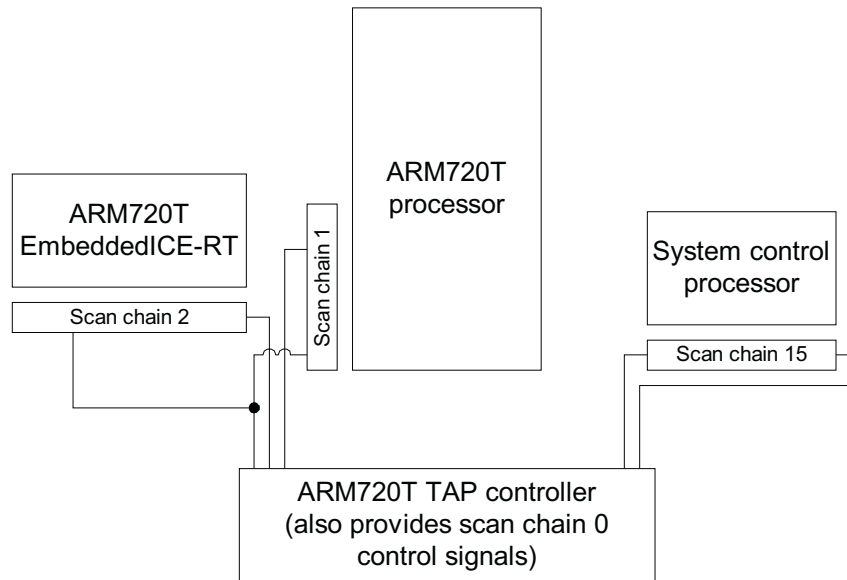
Figure 9-7 ARM720T processor scan chain arrangements

Scan chain 1

Scan chain 1 provides serial access to the core data bus **HRDATA/HWDATA** and the **DBGBREAK** signal.

There are 33 bits in this scan chain, the order being (from serial data in to out):

- data bus bits 0 through 31
- the **DBGBREAK** bit (the first to be shifted out).

Scan chain 2

Scan chain 2 enables access to the EmbeddedICE-RT registers. See *Test data registers* on page 9-32 for details.

Scan chain 15

Scan chain 15 is dedicated to the system control coprocessor registers (the CP15 registers).

There are 37 bits in scan chain 15. From **DBGTDI** to **DBGTDO**, the order of the bits is:

- read/write bit
- instruction encoding bits [3:0] (see Table 9-3)
- data bus bits 31 through 0.

Bit 0 of the data field is the first bit to be scanned in and the first to be scanned out.

The 4-bit instruction encodings for scan chain 15 are shown in Table 9-3.

Table 9-3 Instruction encodings for scan chain 15

Encoding	Instruction
b0000	ID register access (read only)
b0001	Control register access (read/write)
b0010	TTB register access (read/write)
b0011	DAC register access (read/write)
b0100	FSR register access (read/write)
b0101	FAR register access (read/write)
b0110	FCSE PID register access (read/write)
b0111	TRACE PROCID register access (read/write)

Table 9-3 Instruction encodings for scan chain 15 (continued)

Encoding	Instruction
b1000	Invalidate cache (write only)
b1001	Invalidate TLB (write only)
b1010	Invalidate TLB single entry (write only)

——— **Note** ———

The instructions shown in Table 9-3 on page 9-25 are only executed during update. To perform a read, the processor must return to capture state and then shift the result out. In the capture stage, the instruction field of scan chain 15 is RAZ.

9.11.2   Controlling the JTAG interface

The JTAG interface is driven by the currently-loaded instruction in the instruction register (described in *Instruction register* on page 9-33). The loading of instructions is controlled by the *Test Access Port* (TAP) controller.

For more information about the TAP controller, see *The TAP controller* on page 9-27.

                   ARM DDI 0229A

9.12 The TAP controller

The TAP controller is a state machine that determines the state of the boundary-scan test signals **DBGTDI** and **DBGTDO**. Figure 9-8 shows the state transitions that occur in the TAP controller.
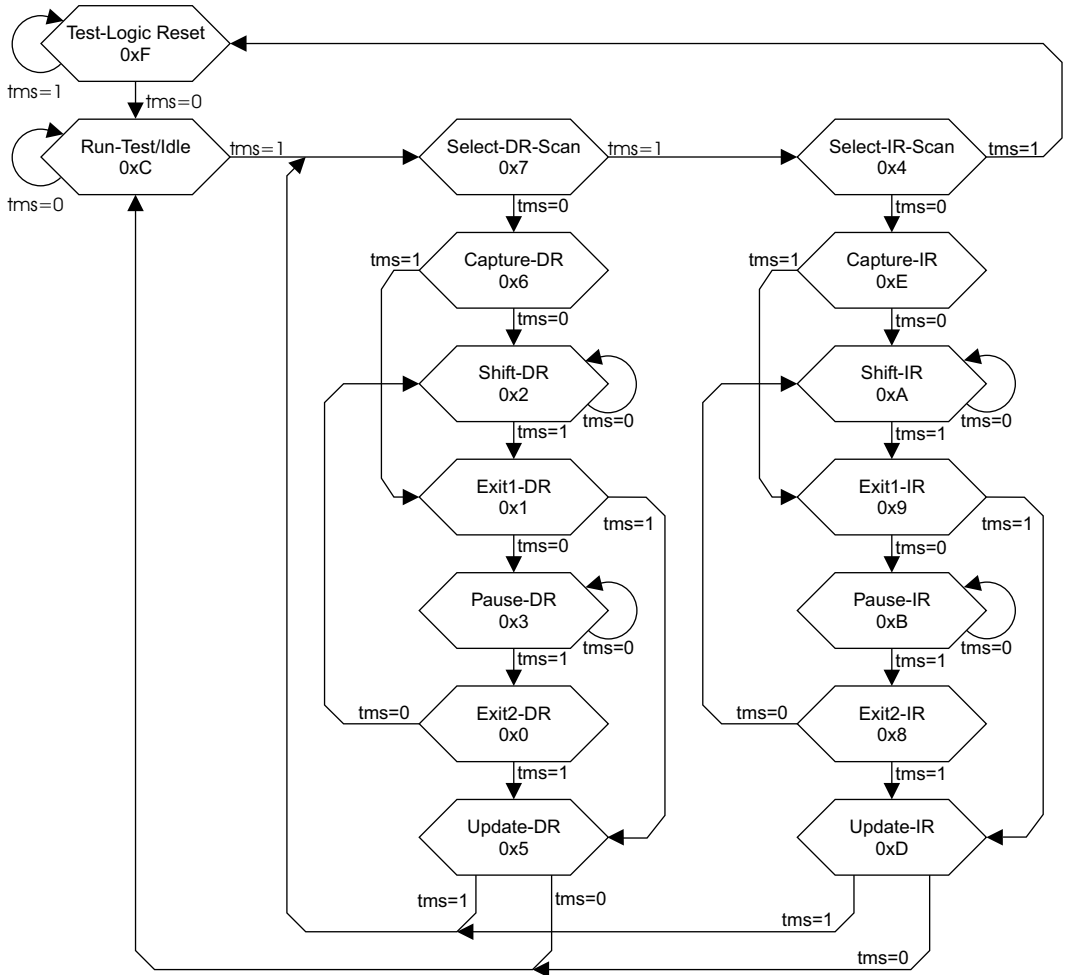
Figure 9-8 Test access port controller state transitions

From IEEE Std 1149.1-1990. Copyright 2001 IEEE. All rights reserved.

9.12.1 Resetting the TAP controller

To force the TAP controller into the correct state after power-up, you must apply a reset pulse to the **DBGnTRST** signal:

* When the boundary-scan interface is to be used, **DBGnTRST** must be driven LOW and then HIGH again.

* When the boundary-scan interface is not to be used, you can tie the **DBGnTRST** input LOW.

The action of reset is as follows:

1. System mode is selected. This means that the boundary-scan cells do not intercept any of the signals passing between the external system and the core.

2. The IDCODE instruction is selected.

   When the TAP controller is put into the SHIFT-DR state and **HCLK** is pulsed while enabled by **DBGTCKEN**, the contents of the ID register are clocked out of **DBGTDO**.

9.13 Public JTAG instructions

Table 9-4 shows the public JTAG instructions.

Table 9-4 Public instructions

Instruction	Binary code
SCAN_N	0010
INTEST	1100
IDCODE	1110
BYPASS	1111
RESTART	0100

In the following descriptions, the ARM720T (Rev 4) processor samples **DBGTDI** and **DBGTMS** on the rising edge of **HCLK** with **DBGTCKEN** HIGH. The TAP controller states are shown in Figure 9-8 on page 9-27.

9.13.1 SCAN_N (0010)

The SCAN_N instruction connects the scan path select register between **DBGTDI** and **DBGTDO**:

- In the CAPTURE-DR state, the fixed value 1000 is loaded into the register.

- In the SHIFT-DR state, the ID number of the desired scan path is shifted into the scan path select register.

- In the UPDATE-DR state, the scan register of the selected scan chain is connected between **DBGTDI** and **DBGTDO**, and remains connected until a subsequent SCAN_N instruction is issued.

- On reset, scan chain 0 is selected by default.

The scan path select register is 4 bits long in this implementation, although no finite length is specified.

9.13.2 INTEST (1100)

The INTEST instruction places the selected scan chain in test mode:

- The INTEST instruction connects the selected scan chain between **DBGTDI** and **DBGTDO**.

- When the INTEST instruction is loaded into the instruction register, all the scan cells are placed in their test mode of operation.

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

- In the SHIFT-DR state, the previously-captured test data is shifted out of the scan chain through the **DBGTDO** pin, while new test data is shifted in through the **DBGTDI** pin.

Single-step operation of the core is possible using the INTEST instruction.

9.13.3 IDCODE (1110)

The IDCODE instruction connects the device identification code register (or ID register) between **DBGTDI** and **DBGTDO**. The ID register is a 32-bit register that enables the manufacturer, part number, and version of a component to be read through the TAP. See *ARM720T processor device identification (ID) code register* on page 9-32 for the details of the ID register format.

When the IDCODE instruction is loaded into the instruction register, all the scan cells are placed in their normal (system) mode of operation:

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

- In the SHIFT-DR state, the previously captured device identification code is shifted out of the ID register through the **DBGTDO** pin, while data is shifted into the ID register through the **DBGTDI** pin.

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

9.13.4 BYPASS (1111)

The BYPASS instruction connects a 1-bit shift register (the bypass register) between **DBGTDI** and **DBGTDO**.

When the BYPASS instruction is loaded into the instruction register, all the scan cells assume their normal (system) mode of operation. The BYPASS instruction has no effect on the system pins:

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

- In the SHIFT-DR state, test data is shifted into the bypass register through **DBGTDI** and shifted out on **DBGTDO** after a delay of one **HCLK** cycle. The first bit to shift out is a zero.

- The bypass register is not affected in the UPDATE-DR state.

All unused instruction codes default to the BYPASS instruction.

9.13.5   RESTART (0100)

The RESTART instruction restarts the processor on exit from debug state. The RESTART instruction connects the bypass register between **DBGTDI** and **DBGTDO**. The TAP controller behaves as if the BYPASS instruction had been loaded.

The processor exits debug state when the RUN-TEST/IDLE state is entered.

For more information, see *Exit from debug state* on page 9-44.

9.14   Test data registers

The six test data registers that can connect between **DBGTDI** and **DBGTDO** are described in the following sections:

- *Bypass register*
- *ARM720T processor device identification (ID) code register*
- *Instruction register* on page 9-33
- *Scan path select register* on page 9-33
- *Scan chain 1* on page 9-35
- *Scan chain 2* on page 9-35.

In the following descriptions, data is shifted during every **HCLK** cycle when **DBGTCKEN** enable is HIGH.

9.14.1   Bypass register

Purpose   Bypasses the device during scan testing by providing a path between **DBGTDI** and **DBGTDO**.

Length   1 bit.

Operating mode   When the BYPASS instruction is the current instruction in the instruction register, serial data is transferred from **DBGTDI** to **DBGTDO** in the SHIFT-DR state with a delay of one **HCLK** cycle enabled by **DBGTCKEN**.
There is no parallel output from the bypass register.
A logic 0 is loaded from the parallel input of the bypass register in the CAPTURE-DR state.

9.14.2   ARM720T processor device identification (ID) code register

Purpose   Reads the 32-bit device identification code. No programmable supplementary identification code is provided.

Length   32 bits. The format of the ID code register is as shown in Figure 9-9.

31 28	27 12	11 1	0
Version	Part number	Manufacturer identity	1
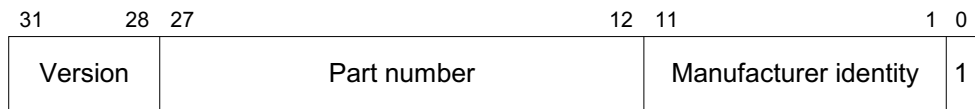
Figure 9-9 ID code register format

Copyright © 2001 ARM Limited. All rights reserved.   ARM DDI 0229A

The default device identification code is `0x7f1f0f0f`.

Operating mode   When the IDCODE instruction is current, the ID register is selected as the serial path between **DBGTDI** and **DBGTDO**.

There is no parallel output from the ID register.

The 32-bit device identification code is loaded into the ID register from its parallel inputs during the CAPTURE-DR state.

9.14.3 Instruction register

Purpose   Changes the current TAP instruction.

Length   4 bits.

Operating mode   In the SHIFT-IR state, the instruction register is selected as the serial path between **DBGTDI**, and **DBGTDO**.

During the CAPTURE-IR state, the binary value 0001 is loaded into this register. This value is shifted out during SHIFT-IR (least significant bit first), while a new instruction is shifted in (least significant bit first).

During the UPDATE-IR state, the value in the instruction register becomes the current instruction.

On reset, IDCODE becomes the current instruction.

There is no parity bit.

9.14.4 Scan path select register

Purpose   Changes the current active scan chain.

Length   4 bits.

Operating mode   SCAN_N as the current instruction in the SHIFT-DR state selects the scan path select register as the serial path between **DBGTDI**, and **DBGTDO**.

During the CAPTURE-DR state, the value 1000 binary is loaded into this register. This value is loaded out during SHIFT-DR (least significant bit first), while a new value is loaded in (least significant bit first). During the UPDATE-DR state, the value in the register selects a scan chain to become the currently active scan chain. All additional instructions, such as INTEST, then apply to that scan chain.

The currently-selected scan chain changes only when a SCAN_N instruction is executed, or when a reset occurs. On reset, scan chain 0 is selected as the active scan chain.

Table 9-5 shows the scan chain number allocation.

Table 9-5 Scan chain number allocation

Scan chain number	Function
0	(User-implemented)
1	Debug
2	EmbeddedICE-RT programming
3	Reserved[a]
4	Reserved[a]
8	Reserved[a]

a. When selected, reserved scan chains scan out zeros.

9.14.5   Scan chains 1 and 2

The scan chains allow serial access to the core logic, and to the EmbeddedICE-RT hardware for programming purposes. Each scan chain cell is simple and comprises a serial register and a multiplexor.

The scan cells perform three basic functions:
- capture
- shift
- update.

For input cells, the capture stage involves copying the value of the system input to the core into the serial register. During shift, this value is output serially. The value applied to the core from an input cell is either the system input, or the contents of the parallel register (loads from the shift register after UPDATE-DR state) under multiplexor control.

For output cells, capture involves placing the value of a core output into the serial register. During shift, this value is serially output as before. The value applied to the system from an output cell is either the core output, or the contents of the serial register.

All the control signals for the scan cells are generated internally by the TAP controller. The action of the TAP controller is determined by current instruction and the state of the TAP state machine.

Scan chain 1

Purpose	Scan chain 1 is used for communication between the debugger, and the ARM720T core. It is used to read and write data, and to scan instructions into the pipeline. The SCAN_N TAP instruction can be used to select scan chain 1.
Length	33 bits, 32 bits a for the data value and 1 bit for the scan cell on the **DBGBREAK** core input.
Scan chain order	From **DBGTDI** to **DBGTDO**, the ARM720T processor data bits, bits 0 to 31, then the 33rd bit, the **DBGBREAK** scan cell.

Scan chain 1, bit 33 serves three purposes:

- Under normal INTEST test conditions, it enables a known value to be scanned into the **DBGBREAK** input.

- While debugging, the value placed in the 33rd bit determines whether the ARM720T core synchronizes back to system speed before executing the instruction. See *System speed access* on page 9-48 for more details.

- After the ARM720T core has entered debug state, the value of the 33rd bit on the first occasion that it is captured, and scanned out tells the debugger whether the core entered debug state from a breakpoint (bit 33 LOW), or from a watchpoint (bit 33 HIGH).

Scan chain 2

Purpose	Scan chain 2 provides access to the EmbeddedICE-RT registers. To do this, scan chain 2 must be selected using the SCAN_N TAP controller instruction, and then the TAP controller must be put in INTEST mode.
Length	38 bits.
Scan chain order	From **DBGTDI** to **DBGTDO**, the read/write bit, the register address bits, bits 4 to 0, then the data bits, bits 0 to 31.

No action occurs during CAPTURE-DR.

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

During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read, 1 = write). See Figure 9-12 on page 9-52 for more details.

9.15   Scan timing

Figure 9-10 provides general scan timing information.
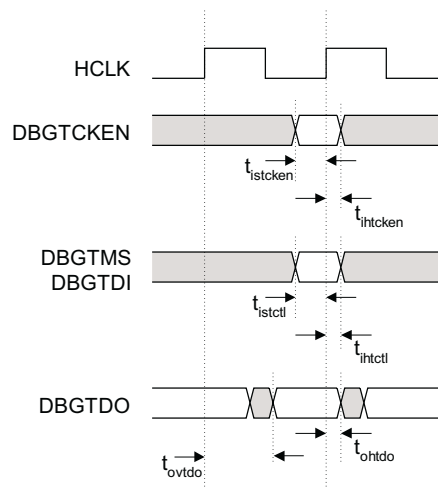
Figure 9-10 Scan timing

9.15.1   Scan chain 1 cells

The ARM720T (Rev 4) processor provides data for scan chain 1 cells as shown in Table 9-6.

Table 9-6 Scan chain 1 cells

Number	Signal	Type
1	**DATA[0]**	Input/output
2	**DATA[1]**	Input/output
3	**DATA[2]**	Input/output
4	**DATA[3]**	Input/output
5	**DATA[4]**	Input/output
6	**DATA[5]**	Input/output
7	**DATA[6]**	Input/output

Table 9-6 Scan chain 1 cells (continued)

Number	Signal	Type
8	**DATA[7]**	Input/output
9	**DATA[8]**	Input/output
10	**DATA[9]**	Input/output
11	**DATA[10]**	Input/output
12	**DATA[11]**	Input/output
13	**DATA[12]**	Input/output
14	**DATA[13]**	Input/output
15	**DATA[14]**	Input/output
16	**DATA[15]**	Input/output
17	**DATA[16]**	Input/output
18	**DATA[17]**	Input/output
19	**DATA[18]**	Input/output
20	**DATA[19]**	Input/output
21	**DATA[20]**	Input/output
22	**DATA[21]**	Input/output
23	**DATA[22]**	Input/output
24	**DATA[23]**	Input/output
25	**DATA[24]**	Input/output
26	**DATA[25]**	Input/output
27	**DATA[26]**	Input/output
28	**DATA[27]**	Input/output
29	**DATA[28]**	Input/output
30	**DATA[29]**	Input/output

Table 9-6 Scan chain 1 cells (continued)

Number	Signal	Type
31	**DATA[30]**	Input/output
32	**DATA[31]**	Input/output
33	**DBGBREAK**	Input

9.16 Examining the core and the system in debug state

When the ARM720T (Rev 4) processor is in debug state, you can examine the core and system state by forcing the load and store multiples into the instruction pipeline.

Before you can examine the core and system state, the debugger must determine whether the processor entered debug state from Thumb state or ARM state, by examining bit 4 of the EmbeddedICE-RT debug status register, as follows:

Bit 4 HIGH   The core has entered debug from Thumb state.

Bit 4 LOW   The core has entered debug from ARM state.

9.16.1 Determining the core state

When the processor has entered debug state from Thumb state, the simplest course of action is for the debugger to force the core back into ARM state. The debugger can then execute the same sequence of instructions to determine the processor state.

To force the processor into ARM state, execute the following sequence of Thumb instructions on the core:

```
STR R0, [R0]; Save R0 before use

MOV R0, PC  ; Copy PC into R0

STR R0, [R0]; Now save the PC in R0

BX  PC      ; Jump into ARM state

MOV R8, R8  ; NOP

MOV R8, R8  ; NOP
```

——— **Note** ———

Because all Thumb instructions are only 16 bits long, you can repeat the instruction when shifting scan chain 1. For example, the encoding for BX R0 is 0x4700, so when 0x47004700 shifts into scan chain 1, the debugger does not have to keep track of the half of the bus on which the processor expects to read the data.

You can use the sequences of ARM instructions below to determine the state of the processor.

With the processor in the ARM state, the first instruction to execute is typically:

```
STM R0, {R0-R15}
```

       ARM DDI 0229A

This instruction causes the contents of the registers to appear on the data bus. You can then sample and shift out these values.

───── **Note** ─────

The use of r0 as the base register for the STM is only for illustration, any register can be used.

─────────────────

After you have determined the values in the current bank of registers, you might wish to access the banked registers. To do this, you must change mode. Normally, a mode change can occur only if the core is already in a privileged mode. However, while in debug state, a mode change from one mode into any other mode can occur.

The debugger must restore the original mode before exiting debug state. For example, if the debugger was requested to return the state of the User mode registers, and FIQ mode registers, and debug state was entered in Supervisor mode, the instruction sequence might be:

```
STM R0, {R0-R15}; Save current registers

MRS R0, CPSR

STR R0, R0; Save CPSR to determine current mode

BIC R0, 0x1F; Clear mode bits

ORR R0, 0x10; Select user mode

MSR CPSR, R0; Enter USER mode

STM R0, {R13,R14}; Save register not previously visible

ORR R0, 0x01; Select FIQ mode

MSR CPSR, R0; Enter FIQ mode

STM R0, {R8-R14}; Save banked FIQ registers
```

All these instructions execute at debug speed. Debug speed is much slower than system speed. This is because between each core clock, 33 clocks occur in order to shift in an instruction, or shift out data. Executing instructions this slowly is acceptable for accessing the core state because the ARM720T processor is fully static. However, you cannot use this method for determining the state of the rest of the system.

While in debug state, only the following instructions can be scanned into the instruction pipeline for execution:

- all data processing operations
- all load, store, load multiple, and store multiple instructions
- MSR and MRS.

9.16.2 Determining system state

To meet the dynamic timing requirements of the memory system, any attempt to access system state must occur with the clock qualified by **HCLKEN**. To perform a memory access, **HCLKEN** must be used to force the ARM720T processor to run in normal operating mode. This is controlled by bit 33 of scan chain 1.

An instruction placed in scan chain 1 with bit 33, the **DBGBREAK** bit, LOW executes at debug speed. To execute an instruction at system speed, the instruction prior to it must be scanned into scan chain 1 with bit 33 set HIGH.

After the system speed instruction has scanned into the data bus and clocked into the pipeline, the RESTART instruction must be loaded into the TAP controller. RESTART causes the ARM720T processor to:

1. Switch automatically to **HCLKEN** control.

2. Execute the instruction at system speed.

3. Reenter debug state.

When the instruction has completed, **DBGACK** is HIGH and the core reverts to **DBGTCKEN** control. It is now possible to select INTEST in the TAP controller and resume debugging.

The debugger must look at both **DBGACK** and **HTRANS[1:0]** to determine whether a system speed instruction has completed. To access memory, the ARM720T core drives both bits of **HTRANS[1:0]** LOW after it has synchronized back to system speed. This transition is used by the memory controller to arbitrate whether the ARM720T core can have the bus in the next cycle. If the bus is not available, the ARM720T processor might have its clock stalled indefinitely. The only way to determine whether the memory access has completed is to examine the state of both **HTRANS[1:0]** and **DBGACK**. When both are HIGH, the access has completed.

The debugger usually uses EmbeddedICE-RT to control debugging, and so the state of **HTRANS[1:0]** and **DBGACK** can be determined by reading the EmbeddedICE-RT status register. See *Debug status register* on page 9-65 for more details.

The state of the system memory can be fed back to the debug host by using system speed load multiples and debug speed store multiples.

There are restrictions on which instructions can have bit 33 set. The valid instructions on which to set this bit are:

- loads
- stores
- load multiple
- store multiple.

See also *Exit from debug state* on page 9-44.

When the ARM720T processor returns to debug state after a system speed access, bit 33 of scan chain 1 is set HIGH. The state of bit 33 gives the debugger information about why the core entered debug state the first time this scan chain is read.

9.17 Exit from debug state

Leaving debug state involves:

- restoring the ARM720T processor internal state
- causing the execution of a branch to the next instruction
- returning to normal operation.

After restoring the internal state, a branch instruction must be loaded into the pipeline. See *The program counter during debug* on page 9-46 for details on calculating the branch.

Bit 33 of scan chain 1 forces the ARM720T processor to resynchronize back to **HCLKEN**, clock enable. The penultimate instruction of the debug sequence is scanned in with bit 33 set HIGH. The final instruction of the debug sequence is the branch, which is scanned in with bit 33 LOW. The core is then clocked to load the branch instruction into the pipeline, and the RESTART instruction is selected in the TAP controller.

When the state machine enters the RUN-TEST/IDLE state, the scan chain reverts back to System mode. The ARM720T processor then resumes normal operation, fetching instructions from memory. This delay, until the state machine is in the RUN-TEST/IDLE state, enables conditions to be set up in other devices in a multiprocessor system without taking immediate effect. When the state machine enters the RUN-TEST/IDLE state, all the processors resume operation simultaneously.

DBGACK informs the rest of the system when the ARM720T processor is in debug state. This information can be used to inhibit peripherals, such as watchdog timers, that have real-time characteristics. **DBGACK** can also mask out memory accesses caused by the debugging process.

For example, when the ARM720T processor enters debug state after a breakpoint, the instruction pipeline contains the breakpointed instruction, and two other instructions that have been prefetched. On entry to debug state the pipeline is flushed. On exit from debug state the pipeline must therefore revert to its previous state.

Because of the debugging process, more memory accesses occur than are expected normally. **DBGACK** can inhibit any system peripheral that might be sensitive to the number of memory accesses. For example, a peripheral that counts the number of memory cycles must return the same answer after a program has been run with and without debugging. Figure 9-11 on page 9-45 shows the behavior of the ARM720T processor on exit from the debug state.
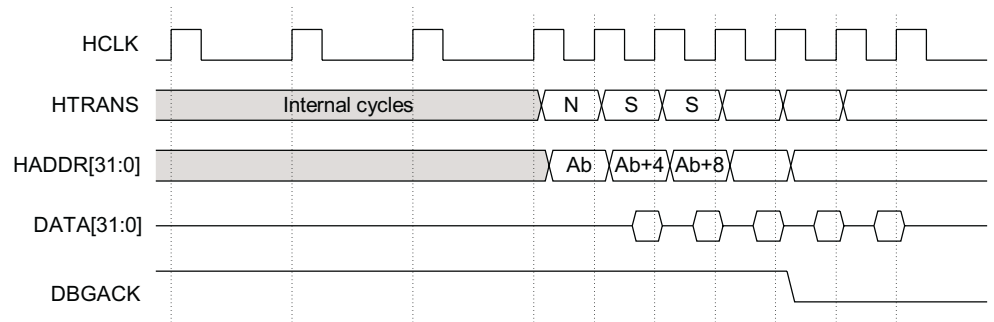
Figure 9-11 Debug exit sequence

Figure 9-3 on page 9-8 shows that the final memory access occurs in the cycle after **DBGACK** goes HIGH. This is the point at which the cycle counter must be disabled. Figure 9-11 shows that the first memory access that the cycle counter has not previously seen occurs in the cycle after **DBGACK** goes LOW. This is the point at which to re-enable the counter.

——— **Note** ———

When a system speed access from debug state occurs, the ARM720T processor temporarily drops out of debug state, so **DBGACK** can go LOW. If there are peripherals that are sensitive to the number of memory accesses, they must be led to believe that the ARM720T processor is still in debug state. You can do this by programming the EmbeddedICE-RT control register to force the value on **DBGACK** to be HIGH. See *Debug status register* on page 9-65 for more details.

9.18   The program counter during debug

The debugger must keep track of what happens to the PC, so that the ARM720T core can be forced to branch back to the place at which program flow was interrupted by debug. Program flow can be interrupted by any of the following:

*   *Breakpoints*
*   *Watchpoints*
*   *Watchpoint with another exception* on page 9-47
*   *Debug request* on page 9-47
*   *System speed access* on page 9-48.

9.18.1   Breakpoints

Entry into debug state from a breakpoint advances the PC by four addresses or 16 bytes. Each instruction executed in debug state advances the PC by one address or 4 bytes.

The usual way to exit from debug state after a breakpoint is to remove the breakpoint and branch back to the previously-breakpointed address.

For example, if the ARM720T processor entered debug state from a breakpoint set on a given address, and two debug speed instructions were executed, a branch of –7 addresses must occur (4 for debug entry, plus 2 for the instructions, plus 1 for the final branch).

The following sequence shows the data scanned into scan chain 1, most significant bit first. The value of the first digit goes to the **DBGBREAK** bit, and then the instruction data into the remainder of scan chain 1:

```
0 E0802000; ADD R2, R0, R0

1 E1826001; ORR R6, R2, R1

0 EAFFFFF9; B –7 (2's complement)
```

After the ARM720T processor enters debug state, it must execute a minimum of two instructions before the branch, although these can both be NOPs (MOV R0, R0). For small branches, you can replace the final branch with a subtract, with the PC as the destination (SUB PC, PC, #28 in the above example).

9.18.2   Watchpoints

The return to program execution after entry to debug state from a watchpoint is made in the same way as the procedure described in *Breakpoints*.

Debug entry adds four addresses to the PC, and every instruction adds one address. The difference from breakpoint is that the instruction that caused the watchpoint has executed, and the program must return to the next instruction.

9.18.3 Watchpoint with another exception

If a watchpointed access simultaneously causes a Data Abort, the ARM720T processor enters debug state in abort mode. Entry into debug is held off until the core changes into abort mode and has fetched the instruction from the abort vector.

A similar sequence follows when an interrupt, or any other exception, occurs during a watchpointed memory access. The ARM720T processor enters debug state in the mode of the exception. The debugger must check to see whether an exception has occurred by examining the current and previous mode (in the CPSR, and SPSR), and the value of the PC. When an exception has taken place, you are given the choice of servicing the exception before debugging.

Entry to debug state when an exception has occurred causes the PC to be incremented by three instructions rather than four, and this must be considered in return branch calculation when exiting debug state. For example, suppose that an abort occurs on a watchpointed access, and ten instructions have been executed to determine this eventuality. You can use the following sequence to return to program execution.

```
0 E1A00000; MOV R0, R0

1 E1A00000; MOV R0, R0

0 EAFFFFF0; B -16
```

This code forces a branch back to the abort vector, causing the instruction at that location to be refetched and executed.

——— **Note** ———

After the abort service routine, the instruction that caused the abort, and watchpoint is refetched and executed. This triggers the watchpoint again and the ARM720T processor reenters debug state.

9.18.4 Debug request

Entry into debug state using a debug request is similar to a breakpoint. However, unlike a breakpoint, the last instruction has completed execution and so must not be refetched on exit from debug state. Therefore, you can assume that entry to debug state adds three addresses to the PC and every instruction executed in debug state adds one address.

For example, suppose you have invoked a debug request, and decide to return to program execution straight away. You could use the following sequence:

```
0 E1A00000; MOV R0, R0

1 E1A00000; MOV R0, R0

0 EAFFFFFA; B –6
```

This code restores the PC and restarts the program from the next instruction.

9.18.5   System speed access

When a system speed access is performed during debug state, the value of the PC increases by three addresses. System speed instructions access the memory system and so it is possible for aborts to take place. If an abort occurs during a system speed memory access, the ARM720T processor enters abort mode before returning to debug state.

This scenario is similar to an aborted watchpoint, but the problem is much harder to fix because the abort was not caused by an instruction in the main program, and so the PC does not point to the instruction that caused the abort. An abort handler usually looks at the PC to determine the instruction that caused the abort and also the abort address. In this case, the value of the PC is invalid, but because the debugger can determine which location was being accessed, the debugger can be written to help the abort handler fix the memory system.

9.18.6   Summary of return address calculations

To determine whether entry to debug state was due to a breakpoint, watchpoint, or debug request (**DBGRQ**), bit 33 (**DBGBREAK**) of scan chain 1 must be consulted together with bit 12 (**DBGMOE**) of the debug status register (register 1 of scan chain 2).

Table 9-7 on page 9-49 shows how **DBGMOE** and **DBGBREAK** vary according to the reason for entry to debug state.

───── **Note** ─────

 DBGMOE and **DBGBREAK** must be read after entry into debug state and before any other accesses to scan chain 1.

Table 9-7 Determining the cause of entry to debug state

DBGMOE	DBGBREAK	Description
0	0	Breakpoint
0	1	Watchpoint
1	X	Debug Request (**DBGRQ**)

The calculation of the branch return address is as follows:

- for normal breakpoint and watchpoint, the branch is:

  - $(4 + N + 3S)$

- for entry through debug request (**DBGRQ**) or watchpoint with exception, the branch is:

  - $(3 + N + 3S)$

where N is the number of debug speed instructions executed (including the final branch) and S is the number of system speed instructions executed.

9.19 Priorities and exceptions

When a breakpoint, or a debug request occurs, the normal flow of the program is interrupted. Therefore, debug can be treated as another type of exception. The interaction of the debugger with other exceptions is described in *The program counter during debug* on page 9-46. This section covers the following priorities:

* *Breakpoint with Prefetch Abort*
* *Interrupts*
* *Data Aborts*.

9.19.1 Breakpoint with Prefetch Abort

When a breakpointed instruction fetch causes a Prefetch Abort, the abort is taken, and the breakpoint is disregarded. Normally, Prefetch Aborts occur when, for example, an access is made to a virtual address that does not physically exist, and the returned data is therefore invalid. In such a case, the normal action of the operating system is to swap in the page of memory, and to return to the previously-invalid address. This time, when the instruction is fetched, and providing the breakpoint is activated (it can be data-dependent), the ARM720T processor enters debug state.

The Prefetch Abort, therefore, takes higher priority than the breakpoint.

9.19.2 Interrupts

When the ARM720T processor enters debug state, interrupts are automatically disabled.

If an interrupt is pending during the instruction prior to entering debug state, the ARM720T processor enters debug state in the mode of the interrupt. On entry to debug state, the debugger cannot assume that the ARM720T processor is in the mode expected by the program of the user. The ARM720T core must check the PC, the CPSR, and the SPSR to determine accurately the reason for the exception.

Debug, therefore, takes higher priority than the interrupt, but the ARM720T processor does remember that an interrupt has occurred.

9.19.3 Data Aborts

When a Data Abort occurs on a watchpointed access, the ARM720T processor enters debug state in abort mode. The watchpoint, therefore, has higher priority than the abort, but the ARM720T processor remembers that the abort happened.

9.20 Watchpoint unit registers

There are two watchpoint units, known as *watchpoint 0* and *watchpoint 1*. You can configure either to be a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). You can make watchpoints and breakpoints data-dependent.

Each watchpoint unit contains three pairs of registers:
- address value and address mask
- data value and data mask
- control value and control mask.

Each register is independently programmable and has a unique address. The function and mapping of the watchpoint unit register is shown in Table 9-1 on page 9-17.

9.20.1 Programming and reading watchpoint registers

A watchpoint register is programmed by shifting data into the EmbeddedICE-RT scan chain (scan chain 2). The scan chain is a 38-bit shift register comprising:
- a 32-bit data field
- a 5-bit address field
- a read/write bit.

This setup is shown in Figure 9-12 on page 9-52.
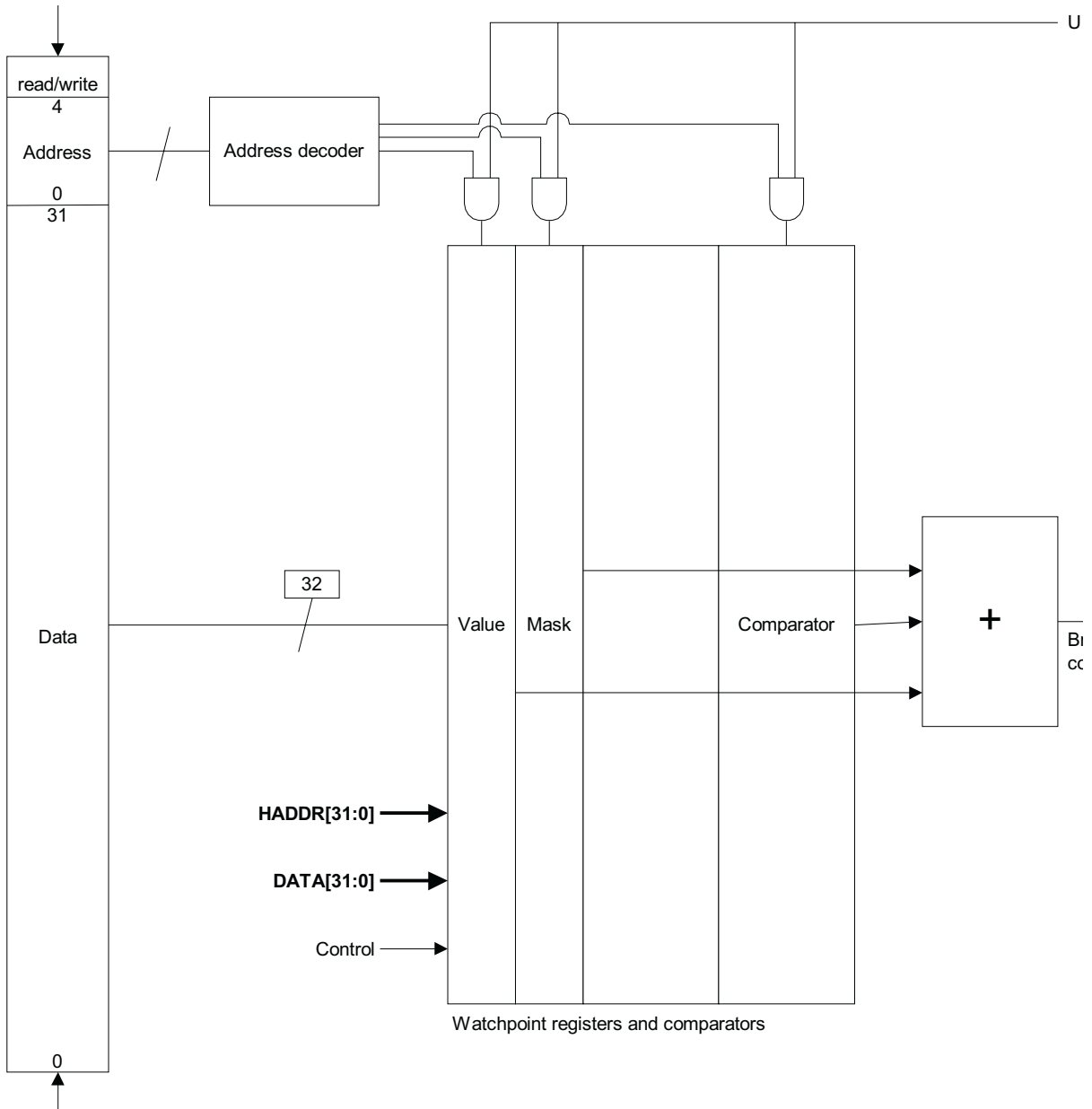
Figure 9-12 EmbeddedICE-RT block diagram

The data to be written is shifted into the 32-bit data field, the address of the register is shifted into the 5-bit address field, and the read/write bit is set.

 ARM DDI 0229A

The data to be written is scanned into the 32-bit data field, the address of the register is scanned into the 5-bit address field, and the read/write bit is set.

A register is read by shifting its address into the address field, and by shifting a 0 into the read/write bit. The 32-bit data field is ignored.

The register addresses are shown in Table 9-1 on page 9-17.

——— **Note** ———
A read or write takes place when the TAP controller enters the UPDATE-DR state.

9.20.2 Using the data, and address mask registers

For each value register in a register pair, there is a mask register of the same format. Setting a bit to 1 in the mask register has the effect of making the corresponding bit in the value register disregarded in the comparison.

For example, when a watchpoint is required on a particular memory location, but the data value is irrelevant, the data mask register can be programmed to 0xffffffff (all bits set) to ignore the entire data bus field.

——— **Note** ———
The mask is an XNOR mask rather than a conventional AND mask. When a mask bit is set to 1, the comparator for that bit position always matches, irrespective of the value register or the input value.

Clearing the mask bit means that the comparator matches only if the input value matches the value programmed into the value register.

9.20.3 The watchpoint unit control registers

The control value and control mask registers are mapped identically in the lower eight bits, as shown in Figure 9-13.

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	DBGEXT	PROT[1]	PROT[0]	SIZE[1]	SIZE[0]	WRITE

Figure 9-13 Watchpoint control value, and mask format

Bit 8 of the control value register is the ENABLE bit and cannot be masked.

The bits have the following functions:

WRITE   Compares against the write signal from the core in order to detect the direction of bus activity. **WRITE** is 0 for a read cycle, and 1 for a write cycle.

SIZE[1:0]   Compares against the **HSIZE[1:0]** signal from the core in order to detect the size of bus activity.

The encoding is shown in Table 9-8.

Table 9-8 SIZE[1:0] signal encoding

bit 1	bit 0	Data size
0	0	Byte
0	1	Halfword
1	0	Word
1	1	(Reserved)

PROT[0]   Is used to detect whether the current cycle is an instruction fetch (**PROT[0]** = 0), or a data access (**PROT[0]** = 1).

PROT[1]   Is used to compare against the not translate signal from the core in order to distinguish between user mode (**PROT[1]** = 0), and non-User mode (**PROT[1]** = 1) accesses.

DBGEXT[1:0]   Is an external input to EmbeddedICE-RT logic that enables the watchpoint to be dependent on some external condition.

The **DBGEXT** input for Watchpoint 0 is labeled **DBGEXT[0]**.

The **DBGEXT** input for Watchpoint 1 is labeled **DBGEXT[1]**.

CHAIN   Can be connected to the chain output of another watchpoint in order to implement, for example, debugger requests of the form `breakpoint on address YYY only when in process XXX`.

In the ARM720T processor EmbeddedICE-RT macrocell, the **CHAINOUT** output of Watchpoint 1 is connected to the **CHAIN** input of Watchpoint 0.

The **CHAINOUT** output is derived from a register. The address/control field comparator drives the write enable for the register. The input to the register is the value of the data field comparator.

The **CHAINOUT** register is cleared when the control value register is written, or when **DBGnTRST** is LOW.

RANGE   In the ARM720T processor EmbeddedICE-RT logic, the **DBGRNG** output of Watchpoint 1 is connected to the **RANGE** input of Watchpoint 0. Connection enables the two watchpoints to be coupled for detecting conditions that occur simultaneously, for example in range checking.

ENABLE   When a watchpoint match occurs, the internal **DBGBREAK** signal is asserted only when the **ENABLE** bit is set. This bit exists only in the value register. It cannot be masked.

For each of the bits [7:0] in the control value register, there is a corresponding bit in the control mask register. This removes the dependency on particular signals.

9.21 Programming breakpoints

Breakpoints are classified as hardware breakpoints or software breakpoints:

- *Hardware breakpoints* typically monitor the address value and can be set in any code, even in code that is in ROM or code that is self-modifying. See *Hardware breakpoints* for more details.

- *Software breakpoints* monitor a particular bit pattern being fetched from any address. One EmbeddedICE-RT watchpoint can therefore be used to support any number of software breakpoints. See *Software breakpoints* on page 9-57 for more details.

   Software breakpoints can usually be set only in RAM because a special bit pattern chosen to cause a software breakpoint has to replace the instruction.

9.21.1 Hardware breakpoints

To make a watchpoint unit cause hardware breakpoints (on instruction fetches):

1. Program its address value register with the address of the instruction to be breakpointed.

2. Program the breakpoint bits for each state as follows:

   For an ARM-state breakpoint

   Set bits [1:0] of the address mask register.

   For a Thumb state breakpoint

   Set bit 0 of the address mask register.

   In either case, clear the remaining bits.

3. Program the data value register only when you require a data-dependent breakpoint, that is only when you have to match the actual instruction code fetched as well as the address. If the data value is not required, program the data mask register to 0xFFFFFFFF (all bits set). Otherwise program it to 0x00000000.

4. Program the control value register with **PROT[0]** = 0.

5. Program the control mask register with **PROT[0]** = 0, all other bits set.

6. When you have to make the distinction between User and non-User mode instruction fetches, program the **PROT[1]** value and mask bits appropriately.

7. If required, program the **DBGEXT**, **RANGE**, and **CHAIN** bits in the same way.

8. Set the mask bits for all unused control values.

—— **Note** ——

In monitor mode, you must set the EmbeddedICE-RT disable bit (bit 5 in the debug control register) before changing the register values, and clear it on completion of the programming.

9.21.2   Software breakpoints

To make a watchpoint unit cause software breakpoints (on instruction fetches of a particular bit pattern):

1.   Program the address mask register of the watchpoint unit to 0xFFFFFFFF (all bits set) so that the address is disregarded.

2.   Program the data value register with the particular bit pattern that has been chosen to represent a software breakpoint.

     If you are programming a Thumb software breakpoint, repeat the 16-bit pattern in both halves of the data value register. For example, if the bit pattern is 0xDFFF, program 0xDFFFDFFF. When a 16-bit instruction is fetched, EmbeddedICE-RT compares only the valid half of the data bus against the contents of the data value register. In this way, you can use a single watchpoint register to catch software breakpoints on both the upper and lower halves of the data bus.

3.   Program the data mask register to 0x00000000.

4.   Program the control value register with **PROT[0]** = 0.

5.   Program the control mask register with **PROT[0]** = 0 and all other bits set.

6.   If you want to make the distinction between User and non-User mode instruction fetches, program the **PROT[1]** bit in the control value register and control mask register accordingly.

7.   If required, program the **DBGEXT**, **RANGE**, and **CHAIN** bits in the same way.

—— **Note** ——

You do not have to program the address value register.

Setting the breakpoint

To set the software breakpoint:

1.   Read the instruction at the desired address and store it.

2.   Write the special bit pattern representing a software breakpoint at the address.

Clearing the breakpoint

To clear the software breakpoint, restore the instruction to the address.

 ARM DDI 0229A

9.22   Programming watchpoints

This section contains examples of how to program the watchpoint unit to generate breakpoints and watchpoints. Many other ways of programming the watchpoint unit registers are possible. For example, simple range breakpoints can be provided by setting one or more of the address mask bits.

To make a watchpoint unit cause watchpoints (on data accesses):

1.   Program its address value register with the address of the data access to be watchpointed.

2.   Program the address mask register to `0x00000000`.

3.   Program the data value register only if you require a data-dependent watchpoint, that is, only if you have to match the actual data value read or written as well as the address. If the data value is irrelevant, program the data mask register to `0xFFFFFFFF` (all bits set). Otherwise program the data mask register to `0x00000000`.

4.   Program the control value register as follows:

PROT[0]	Set.
HWRITE	Clear for a read.
	Set for a write.
SIZE[1:0]	Program with the value corresponding to the appropriate data size.

5.   Program the control mask register as follows:

PROT[0]	Clear.
HWRITE	Clear.

——— **Note** ———

You can set this bit if both reads and writes are to be watchpointed.

SIZE[1:0]	Clear.

——— **Note** ———

You can set these bits if data size accesses are to be watchpointed.

All other bits	Set.

6. If you have to make the distinction between User and non-User mode data accesses, program the **PROT[1]** bit in the control value and control mask registers accordingly.

7. If required, program the **DBGEXT**, **RANGE**, and **CHAIN** bits in the same way.

9.23   Abort status register

Only bit 0 of this 32 bit read/write register is used. It determines whether an abort exception entry was caused by a breakpoint, a watchpoint, or a real abort. The format is shown in Figure 9-14.

31:1	0
SBZ/RAZ	**DbgAbt**

Figure 9-14 Debug abort status register

Bit 0 is set when the ARM720T core takes a Prefetch or Data Abort as a result of a breakpoint or watchpoint. If, on a particular instruction or data fetch, both the Debug Abort and the external Abort signal are asserted, the external Abort takes priority, and the DbgAbt bit is not set. Once set, DbgAbt remains set until reset by the user. The register is accessed by MRC and MCR instructions.

9.24   Debug control register

The debug control register is six bits wide. Writes to the debug control register occur when a watchpoint unit register is written. Reads of the debug control register occur when a watchpoint unit register is read. See *Watchpoint unit registers* on page 9-51 for more information.

Figure 9-15 shows the function of each bit in the debug control register.

5	4	3	2	1	0
EmbeddedICE-RT disable	Monitor mode enable	SBZ/RAZ	**INTDIS**	**DBGRQ**	**DBGACK**

Figure 9-15 Debug control register format

The debug control register bit assignments are shown in Table 9-9.

Table 9-9 Debug control register bit assignments

Bit	Function
5	Used to disable the EmbeddedICE-RT comparator outputs while the watchpoint and breakpoint registers are being programmed. This bit can be read and written through JTAG. Set bit 5 when: • programming breakpoint or watchpoint registers • changing bit 4 of the debug control register. You must clear bit 5 after you have made the changes, to re-enable the EmbeddedICE-RT logic and make the new breakpoints and watchpoints operational.
4	Used to determine the behavior of the core when breakpoints or watchpoints are reached: • If clear, the core enters debug state when a breakpoint or watchpoint is reached. • If set, the core performs an abort exception when a breakpoint or watchpoint is reached. This bit can be read and written from JTAG.
3	This bit must be clear.

Table 9-9 Debug control register bit assignments (continued)

Bit	Function
2	Used to disable interrupts: • If set, the interrupt enable signal of the core (**IFEN**) is forced LOW. The **IFEN** signal is driven as shown in Table 9-10. • If clear, interrupts are enabled.
1	Used to force the value on **DBGRQ**.
0	Used to force the value on **DBGACK**.

9.24.1 Disabling interrupts

IRQs and FIQs are disabled under the following conditions:

- during debugging (**DBGACK** HIGH)
- when the **INTDIS** bit is set.

The core interrupt enable signal, **IFEN**, is driven as shown in Table 9-10.

Table 9-10 Interrupt signal control

DBGACK	INTDIS	IFEN	Interrupts
0	0	1	Permitted
1	x	0	Inhibited
x	1	0	Inhibited

9.24.2 Forcing DBGRQ

Figure 9-17 on page 9-66 shows that the value stored in bit 1 of the debug control register is synchronized and then ORed with the external **DBGRQ** before being applied to the processor. The output of this OR gate is the signal **DBGRQI** which is brought out externally from the macrocell.

The synchronization between debug control register bit 1 and **DBGRQI** assists in multiprocessor environments. The synchronization latch only opens when the TAP controller state machine is in the RUN-TEST-IDLE state. This enables an enter-debug

condition to be set up in all the processors in the system while they are still running. When the condition is set up in all the processors, it can be applied to them simultaneously by entering the RUN-TEST-IDLE state.

9.24.3   Forcing DBGACK

Figure 9-17 on page 9-66 shows that the value of the internal signal **DBGACKI** from the core is ORed with the value held in bit 0 of the debug control register, to generate the external value of **DBGACK** seen at the periphery of the ARM720T core. This enables the debug system to signal to the rest of the system that the core is still being debugged even when system-speed accesses are being performed (when the internal **DBGACK** signal from the core is LOW).

9.25 Debug status register

The debug status register is 13 bits wide. If it is accessed for a write (with the read/write bit set), the status bits are written. If it is accessed for a read (with the read/write bit clear), the status bits are read. The format of the debug status register is shown in Figure 9-16.

12	11	5	4	3	2	1	0
DBGMOE			TBIT	TRANS[1]	IFEN	DBGRQ	DBGACK

Figure 9-16 Debug status register format

The function of each bit in this register is shown in Table 9-11.

Table 9-11 Debug status register bit assignments

Bit	Function
12	Enables the debugger to determine whether the core has entered debug state due to the assertion of **DBGRQ**.
4	Enables **TBIT** to be read. This enables the debugger to determine what state the processor is in, and which instructions to execute.
3	Enables the state of the **HTRANS[1]** signal from the core to be read. This enables the debugger to determine whether a memory access from the debug state has completed.
2	Enables the state of the core interrupt enable signal, **IFEN**, to be read.
1	Enables the values on the synchronized version of **DBGRQ** to be read.
0	Enables the values on the synchronized versions of **DBGACK** to be read.

The structure of the debug control and status registers is shown in Figure 9-17 on page 9-66.
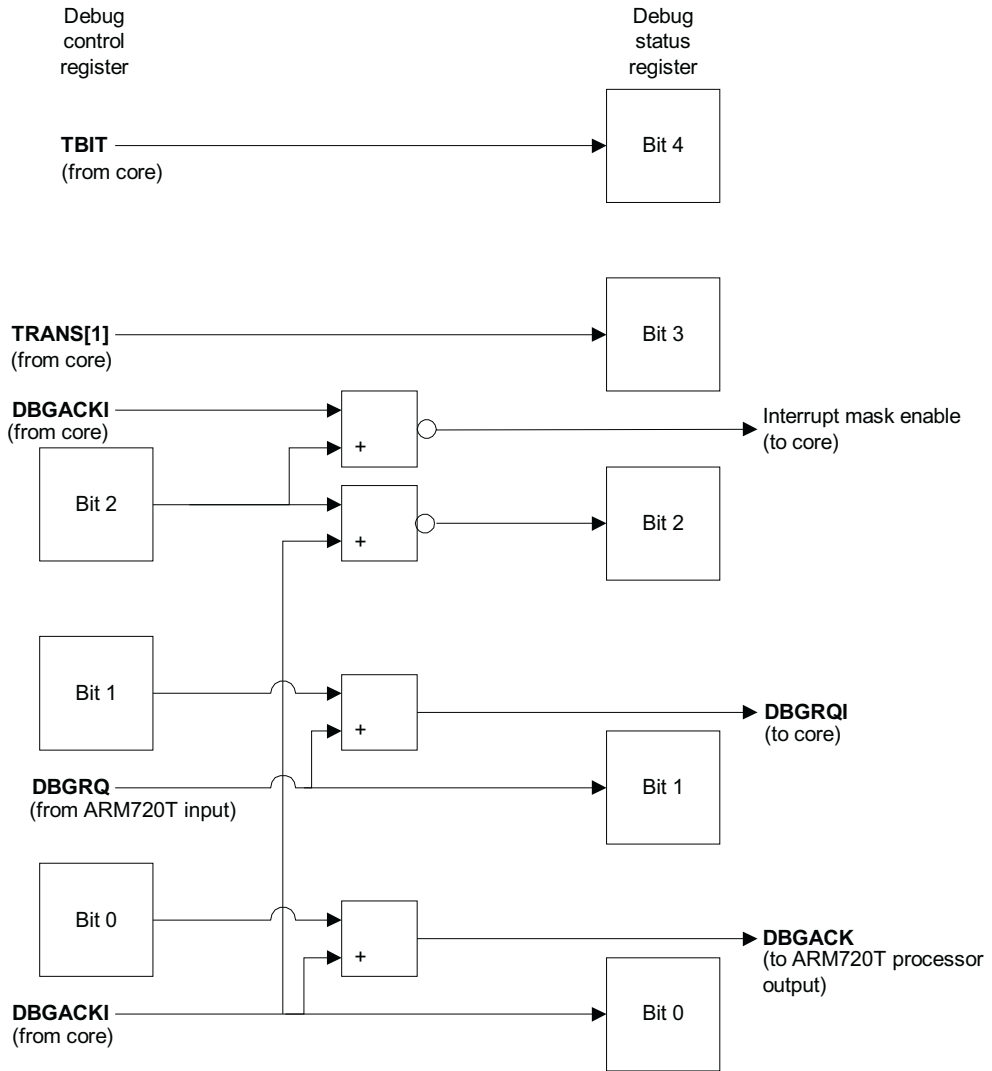
Figure 9-17 Debug control and status register structure

9.26 Coupling breakpoints and watchpoints

You can couple watchpoint units 1 and 0 together using the **CHAIN** and **RANGE** inputs. The use of **CHAIN** enables Watchpoint 0 to be triggered only if Watchpoint 1 has previously matched. The use of **RANGE** enables simple range checking to be performed by combining the outputs of both watchpoints.

9.26.1 Breakpoint and watchpoint coupling example

Let:

Av[31:0]	Be the value in the address value register
Am[31:0]	Be the value in the address mask register
A[31:0]	Be the address bus from the ARM720T (Rev 4) processor
Dv[31:0]	Be the value in the data value register
Dm[31:0]	Be the value in the data mask register
D[31:0]	Be the data bus from the ARM720T (Rev 4) processor
Cv[8:0]	Be the value in the control value register
Cm[7:0]	Be the value in the control mask register
C[9:0]	Be the combined control bus from the ARM720T core, other watchpoint registers, and the **DBGEXT** signal.

CHAINOUT signal

The **CHAINOUT** signal is derived as follows:

```
WHEN (({Av[31:0],Cv[4:0]} XNOR {A[31:0],C[4:0]}) OR {Am[31:0],Cm[4:0]} == 0xFFFFFFFFF)

CHAINOUT = ((({Dv[31:0],Cv[6:4]} XNOR {D[31:0],C[7:5]}) OR {Dm[31:0],Cm[7:5]}) == 0x7FFFFFFFF)
```

The **CHAINOUT** output of watchpoint register 1 provides the **CHAIN** input to Watchpoint 0. This **CHAIN** input enables you to use quite complicated configurations of breakpoints and watchpoints.

——— **Note** ———
There is no **CHAIN** input to Watchpoint 1 and no **CHAIN** output from Watchpoint 0.

For example, consider the request by a debugger to breakpoint on the instruction at location YYY when running process XXX in a multiprocess system. If the current process ID is stored in memory, you can implement the above function with a watchpoint and breakpoint chained together. The watchpoint address points to a known memory location containing the current process ID, the watchpoint data points to the required process ID and the **ENABLE** bit is cleared.

The address comparator output of the watchpoint is used to drive the write enable for the **CHAINOUT** latch. The input to the latch is the output of the data comparator from the same watchpoint. The output of the latch drives the **CHAIN** input of the breakpoint comparator. The address YYY is stored in the breakpoint register, and when the **CHAIN** input is asserted, the breakpoint address matches and the breakpoint triggers correctly.

9.26.2 DBGRNG signal

The **DBGRNG** signal is derived as follows:

```
DBGRNG = (((({Av[31:0],Cv[4:0]} XNOR {A[31:0],C[4:0]}) OR {Am[31:0],Cm[4:0]}) == 0xFFFFFFFFF) AND
(((({Dv[31:0],Cv[7:5]} XNOR {D[31:0],C[7:5]}) OR
Dm[31:0],Cm[7:5]}) == 0x7FFFFFFFF)
```

The **DBGRNG** output of watchpoint register 1 provides the **RANGE** input to watchpoint register 0. This **RANGE** input enables you to couple two breakpoints together to form range breakpoints.

———— **Note** ————

Selectable ranges are restricted to being powers of 2.

For example, if a breakpoint is to occur when the address is in the first 256 bytes of memory, but not in the first 32 bytes, program the watchpoint registers as follows:

For Watchpoint 1:

1. Program Watchpoint 1 with an address value of `0x00000000` and an address mask of `0x0000001F`.

2. Clear the **ENABLE** bit.

3. Program all other Watchpoint 1 registers as normal for a breakpoint.

    An address within the first 32 bytes causes the **RANGE** output to go HIGH but does not trigger the breakpoint.

For Watchpoint 0:

1.   Program Watchpoint 0 with an address value of `0x00000000`, and an address mask of `0x000000FF`.

2.   Set the ENABLE bit.

3.   Program the RANGE bit to match a 0.

4.   Program all other Watchpoint 0 registers as normal for a breakpoint.

If Watchpoint 0 matches but Watchpoint 1 does not (that is, the **RANGE** input to Watchpoint 0 is 0), the breakpoint is triggered.

9.27   EmbeddedICE-RT timing

EmbeddedICE-RT samples the **DBGEXT[1]** and **DBGEXT[0]** inputs on the rising edge of **HCLK**.

Chapter 10
ETM Interface

This chapter describes the ETM interface that is provided on the ARM720T (Rev 4) processor. It contains the following sections:

- *About the ETM interface* on page 10-2
- *Enabling and disabling the ETM7 interface* on page 10-3
- *Connections between the ETM7 macrocell and the ARM720T (Rev 4) processor* on page 10-4
- *Clocks and resets* on page 10-6
- *Debug request wiring* on page 10-7
- *TAP interface wiring* on page 10-8.

10.1 About the ETM interface

You can connect an external *Embedded Trace Macrocell* (ETM) to the ARM720T processor, so that you can perform real-time tracing of the code that the processor is executing.

In general, little or no glue logic is required to connect the ETM7 to the ARM720T (Rev 4) processor. You program the ETM through a JTAG interface. The interface is an extension of the ARM TAP controller, and is assigned scan chain 6.

——— **Note** ———

If you have more than one ARM processor in your system, each processor must have its own dedicated ETM.

See the *ETM7 (Rev 1) Technical Reference Manual* for detailed information about integrating an ETM7 with an ARM720T processor.

10.2   Enabling and disabling the ETM7 interface

Under the control of the ARM debug tools, the ETM7 **PWRDOWN** output is used to enable and disable the ETM. When **PWRDOWN** is HIGH, this indicates that the ETM is not currently enabled, so you can stop the **CLK** input and hold the other ETM signals stable. This enables you to reduce power consumption when you are not performing tracing.

When a TAP reset (**nTRST**) occurs, **PWRDOWN** is forced HIGH until the ETM7 control register has been programmed (see the *Embedded Trace Macrocell Specification* for details of this register).

PWRDOWN is automatically cleared at the start of a debug session.

On the ARM720T processor, the ETM interface pins are gated by the **ETMEN** input. This means that if the **ETMEN** input is LOW, all the output pins of the ETM interface remain stable. You can control this **ETMEN** input by connecting it with either of the following:

•   the **ETMEN** output on the ETM7

•   the inverted **PWRDOWN** output on the ETM7.

10.3 Connections between the ETM7 macrocell and the ARM720T (Rev 4) processor

Table 10-1 shows the connections that you must make between the ETM7 macrocell and the ARM720T processor.

Table 10-1 Connections between the ETM7 macrocell and the ARM720T (Rev 4) processor

ETM7 macrocell signal name	ARM720T (Rev 4) processor signal name
A[31:0]	ETMADDR[31:0]
ABORT	ETMABORT
ARMTDO	DBGTDO
BIGEND	ETMBIGEND
CLK[a]	HCLK[a]
CLKEN	ETMCLKEN
CPA	ETMCPA
CPB	ETMCPB
DBGACK	ETMDBGACK
DBGRQ[b]	DBGRQ[b]
nMREQ	ETMnMREQ
SEQ	ETMSEQ
MAS[1:0]	ETMSIZE[1:0]
nCPI	ETMnCPI
nEXEC	ETMnEXEC
nOPC	ETMnOPC
nRESET	HRESETn
nRW	ETMnRW
nTRST[a]	DBGnTRST[a]
PROCID[31:0]	ETMPROCID[31:0]

Table 10-1 Connections between the ETM7 macrocell and the ARM720T (Rev 4) processor (continued)

ETM7 macrocell signal name	ARM720T (Rev 4) processor signal name
PROCIDWR	**ETMPROCIDWR**
ETMEN or inverted **PWRDOWN**	**ETMEN**[c]
-	**ETMHIVECS**[d]
RANGEOUT[0]	**DBGRNG[0]**
RANGEOUT[1]	**DBGRNG[1]**
RDATA[31:0]	**ETMRDATA[31:0]**
TBIT	**ETMTBIT**
TCK[a]	**HCLK**[a]
TCKEN	**DBGTCKEN**
TDI	**DBGTDI**
TDO[e]	**DBGTDO**
TMS	**DBGTMS**
WDATA[31:0]	**ETMWDATA[31:0]**
INSTRVALID	**ETMINSTRVALID**

a. See *Clocks and resets* on page 10-6.

b. See *Debug request wiring* on page 10-7.

c. See *Enabling and disabling the ETM7 interface* on page 10-3.

d. Leave this pin unconnected.

e. See *TAP interface wiring* on page 10-8.

10.4   Clocks and resets

The ARM720T (Rev 4) processor uses a single clock, **HCLK**, as both the main system clock and the JTAG clock. You must connect the processor clock to both **HCLK** and **TCK** on the ETM. You can then use **TCKEN** to control the JTAG interface.

To trace through a warm reset of the ARM720T processor, use the TAP reset (connect **nTRST** to **DBGnTRST**) to reset the ETM7 state.

For more information about ETM7 clocks and resets, see the *ETM7 Technical Reference Manual*.

10.5   Debug request wiring

It is recommended that you connect together the **DBGRQ** output of the ETM7 to the **DBGRQ** input of the ARM720T processor. If this input is already in use, you can OR the **DBGRQ** inputs together. See the *ETM7 Technical Reference Manual* for more details.

10.6   TAP interface wiring

The ARM720T (Rev 4) processor does not provide a scan chain expansion input. ARM Limited recommends that you connect the ARM720T processor and the ETM7 TAP controllers in parallel. For more details, see the *ETM7 (Rev 1) Technical Reference Manual*.

Chapter 11
Test Support

This chapter describes the test methodology and the CP15 test registers for the ARM720T (Rev 4) processor synthesized logic and TCM. It contains the following sections:

- *About the ARM720T (Rev 4) test registers* on page 11-2
- *Automatic Test Pattern Generation (ATPG)* on page 11-3
- *Test state register* on page 11-4
- *Cache test registers and operations* on page 11-5
- *MMU test registers and operations* on page 11-12.

11.1 About the ARM720T (Rev 4) test registers

Coprocessor 15 register 15 (c15) of the ARM720T processor is used to provide device-specific test operations. You can use it to access and control the following:

- *Test state register* on page 11-4
- *Cache test registers and operations* on page 11-5
- *MMU test registers and operations* on page 11-12.

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 operations is:

```
MCR/MRC   p15,opcode_1,Rd,c15,CRm,opcode_2
```

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

Cond	1 1 1 0	opcode_1	L	CRn	Rd	1 1 1 1	opcode_2	1	CRm
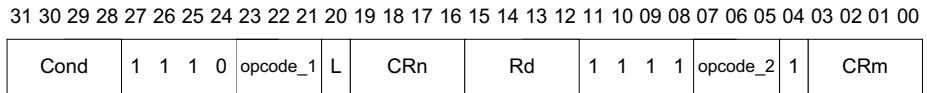
Figure 11-1 CP15 MRC and MCR bit pattern

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

 ARM DDI 0229A

11.2 Automatic Test Pattern Generation (ATPG)

Scan insertion is already performed and fixed for the ARM720T (Rev 4) processor. You can use *Automatic Test Pattern Generation* (ATPG) tools to create the necessary scan patterns to test the logic outputs from all registers.

11.2.1 ARM720T (Rev 4) processor INTEST/EXTEST wrapper

In addition to the auto-inserted scan chains, the ARM720T (Rev 4) macrocell includes all the signals for an optional INTEST/EXTEST scan chain, scan chain 0.

ATPG

Seven balanced scan chains are provided for ATPG, along with a test enable and a single scan enable.

11.3   Test state register

The test state register contains only one bit, bit 0:

Bit 0 set   Enable MMU and cache test.

Bit 0 clear   Disable MMU and cache test.

At reset (**HRESETn** LOW), bit 0 is cleared.

The test state register operations are shown in Table 11-1.

Table 11-1 Test state register operations

Operation	Instruction
Write test register	`MCR p15, 7, Rd, c15, c15, 7`
Read test register	`MRC p15, 7, Rd, c15, c15, 7`

11.4   Cache test registers and operations

The cache is maintained using MCR and MRC instructions to CP15 registers 7 and 9, defined by the ARM v4T programmer's model. Additional operations are available using MCR and MRC instructions to CP15 register 15. These operations are combined with those using registers 7 and 9 to enable testing of the caches entirely in software.

CP15 register 7 (c7) is write-only, and provides only one function:

•   invalidate cache.

The CP15 register 9 (c9) operations are read and write. The operations available are:
•   write victim and lockdown base
•   write victim.

The CP15 register 15 (c15) operations are:
•   write to register C15.C
•   read from register C15.C
•   CAM read to C15.C
•   CAM write
•   RAM read to C15.C
•   RAM write from C15.C
•   CAM match, RAM read to C15.C.

——— **Note** ———

For the CAM Match, RAM Read operation the respective MMU does not perform a lookup and a cache miss does not cause a linefill.

The register 15 operations are all issued as MCR. The Rd field defines the address for the operation. Therefore, the data is either supplied from, or latched into, CP15.C in CP15. These 32-bit registers are accessed with CP15 MCR and MRC instructions.

Table 11-2 summarizes c7, c9, and c15 operations.

Table 11-2 Summary of CP15 register 7, 9, and 15 operations

Function	Rd	Instruction
Invalidate cache	SBZ	MCR p15,0,Rd,c7,c7,0
Write cache victim and lockdown base	Victim=Base	MCR p15,0,Rd,c9,c0,0
Write cache victim	Victim, Seg	MCR p15,0,Rd,c9,c1,0

Table 11-2 Summary of CP15 register 7, 9, and 15 operations (continued)

Function	Rd	Instruction
CAM read to C15.C	Seg	MCR p15,2,Rd,c15,c7,2
CAM write	Tag, Seg, Dirty	MCR p15,2,Rd,c15,c7,6
RAM read to C15.C	Seg, Word	MCR p15,2,Rd,c15,c11,2
RAM write from C15.C	Seg, Word	MCR p15,2,Rd,c15,c11,6
CAM match, RAM read to C15.C	Tag, Seg, Word	MCR p15,2,Rd,c15,c7,5
Write to register C15.C	Data	MCR p15,3,Rd,c15,c3,0
Read from register C15.C	Data read	MRC p15,3,Rd,c15,c3,0

The CAM read format for Rd is shown in Figure 11-2.

Figure 11-2 Rd format, CAM read

The CAM write format for Rd is shown in Figure 11-3.

Figure 11-3 Rd format, CAM write

In Figure 11-3 on page 11-6, bit labels have the following meanings:

V   Valid.

De   Dirty even (words [3:0]).

Do   Dirty odd (words [7:4]).

WB   Writeback.

The RAM read format for Rd is shown in Figure 11-4.

31			7	6	5	4		2	1	0
SBZ				Seg		Word			SBZ	

Figure 11-4 Rd format, RAM read

The RAM write format for Rd is shown in Figure 11-5.

31			7	6	5	4		2	1	0
SBZ				Seg		Word			SBZ	

Figure 11-5 Rd format, RAM write

The CAM match, RAM read format for Rd is shown in Figure 11-6.

31			7	6	5	4		2	1	0
MVA TAG				Seg		Word			SBZ	

Figure 11-6 Rd format, CAM match RAM read

The CAM read format for data is shown in Figure 11-7.

Figure 11-7 Data format, CAM read

The RAM read format for data is shown in Figure 11-8.
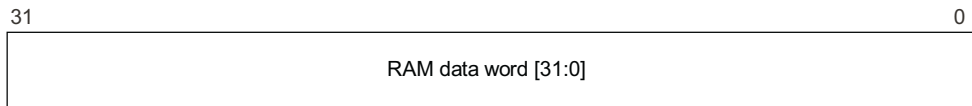
Figure 11-8 Data format, RAM read

The CAM match, RAM read format for data is shown in Figure 11-9.
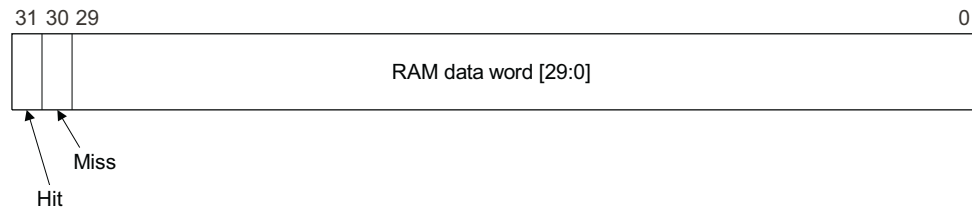
Figure 11-9 Data format, CAM match RAM read

11.4.1 Addressing the CAM and RAM

For the CAM read or write, and RAM read or write operations you must specify the segment, index, and word (for the RAM operations). The CAM and RAM operations use the value in the victim pointer for that segment, so you must ensure that the value is written in the victim pointer before any CAM or RAM operation.

If the MCR *write victim and lockdown base* is used, then the victim pointer is incremented after every CAM read or write, and every RAM read or write. If the MCR *write victim* is used, then the victim pointer is only incremented after every CAM read or write. This enables efficient reading or writing of the CAM and RAM for an entire segment. The write cache victim and lockdown operations are shown in Table 11-3.

Table 11-3 Write cache victim and lockdown operations

Operation	Instructions
Write cache victim and lockdown base	MCR p15,0,Rd,c9,c0,0 MCR p15,0,Rd,c9,c0,1
Write cache victim	MCR p15,0,Rd,c9,c1,0 MCR p15,0,Rd,c9,c1,1

The write cache victim and lockdown base format for Rd is shown in Figure 11-10.

31	26 25	0
Index	SBZ	

Figure 11-10 Rd format, write cache victim and lockdown base

The write cache victim format for Rd is shown in Figure 11-11.

31	26 25	7 6 5 4	0
Index	SBZ	Seg	SBZ

Figure 11-11 Rd format, write cache victim

Another cache test register, C15.C, is written with the current victim of the addressed segment whenever an MCR CAM read is executed. This is intended for use in debug to establish the value of the current victim pointer of each segment before reading the values of the CAM and RAM, so that the value can be restored afterwards.

Example 11-1 shows sample code for performing software test of the cache. It contains typical operations with register C15.C.

Example 11-1 Cache test operations

```
; CAM write, read and check for segment 2

; Write cache victim pointer with index 0, segment 2
        MOV r0,#0
        ORR r1,r0,#2 :SHL: 0x5
        MCR p15,0,r1,c9,c1,0

; Write pattern in 0xFFFFFF9E in all 64 CAM lines
        MVN r2,#1          ; bit 0 should be '0'
        BIC r2,r2,#0x20    ; write segment 2
        MOV r8,#64
loop0   MCR p15,2,r2,c15,c7,6   ; write CAM, index auto-incremented
        SUBS r8,r8,#1
        BNE loop0

; Now read and check
; Reset victim pointer to index 0, segment 2
        MOV r0,#0
        ORR r1,r0 :SHL: 0x5
        MCR p15,0,r1,c9,c1,0
        MOV r8,#64
        MOV r3,#0x40       ; read segment 2
        BIC r2,r2,#0x60    ; clear bit 5 and 6 (always read as '0')
loop1   MCR p15,3,r0,c15,c3,0   ; write C15.C to '0'
        MCR p15,2,r3,c15,c7,2   ; read CAM to C15.C
        MRC p15,3,r4,c15,c3,0   ; read C15.C to R4
        BIC r4,r4,#1            ; clear LFSR bit
        CMP r4,r2
        BNE TEST_FAIL
        SUBS r8,r8,#1
        BNE loop1
        B TEST_PASS

; RAM write, read and check for segment 1

; Write cache victim pointer with index 0, segment 1
        MOV r0,#0
```

```
        ORR r1,r0,#1 :SHL: 0x5
        MCR p15,0,r1,c9,c1,0

; Write pattern 0x5A5A5A5A in RAM line (eight words)
        LDR r0,=0x5A5A5A5A
        MOV r8,#8
        MOV r2,#0x20          ; write segment 2, word 0
        MCR p15,3,r0,c15,c3,0     ; write RAM data in C15.C
loop0   MCR p15,2,r2,c15,c11,6   ; write RAM
        ADD r2,r2,#0x04          ; next word
        SUBS r8,r8,#1
        BNE loop0

; Now read and check

        MOV r8,#8
        MOV r2,#0x20
        MOV r1,#0
loop1   MCR p15,3,r1,c15,c3,0    ; write C15.C to '0'
        MCR p15,2,r2,c15,c11,2   ; read RAM to C15.C
        MRC p15,3,r5,c15,c3,0    ; read C15.C to R4
        CMP r5,r0
        BNE TEST_FAIL
        SUBS r8,r8,#1
        BNE loop1
        B TEST_PASS
```

11.5   MMU test registers and operations

The TLB is maintained using MCR and MRC instructions to CP15 registers 2, 3, 5, 6, 8, and 10, defined by the ARM v4T programmer's model.

The CP15 register 2 (c2) operations control the *Translation Table Base* (TTB). These operations are:
- write TTB registers
- read TTB register.

The CP15 register 3 (c3) operations control the *Domain Access Control* (DAC) register. These operations are:
- write DAC registers
- read DAC register.

The CP15 register 5 (c5) operations control the *Fault Status Register* (FSR). These operations are:
- write FSR
- read FSR.

The CP15 register 6 (c6) operations control the *Fault Address Register* (FAR). These operations are:
- write FAR
- read FAR.

The CP15 register 8 (c8) operations control the TLB and are all write-only. These operations are:
- invalidate TLB
- invalidate single entry using MVA.

The CP15 register 10 (c10) operations control TLB lockdown. These operations are:
- read victim, lockdown base and preserve bit
- write victim, lockdown base and preserve bit.

The CP15 register 15 (c15) operations that operate on the CAM, RAM1, and RAM2 are shown in Table 11-4.

Table 11-4 CAM, RAM1, and RAM2 register 15 operations

Function	Rd	Data
CAM read to C15.M	SBZ	Tag, Size, V, P
CAM write	Tag, Size, V, P	
RAM1 read to C15.M	SBZ	Protection
RAM1 write	Protection	
RAM2 read to C15.M	SBZ	PA Tag, Size
RAM2 write	PA Tag, Size	PA Tag, Size
CAM match RAM1 read to C15.M	MVA	Fault, Miss, Protection

––––– **Note** –––––––

For the CAM match, RAM1 read operation a TLB miss will not cause a page walk.

These register 15 operations are all issued as MCR, which means that the read and match operations have to be latched into register CP15.M in CP15. This is a 32-bit register that is read with the following CP15 MRC instruction:

```
Read from register CP15.M
```

Table 11-5 summarizes c2, c3, c5, c6, c8, c10, and c15 operations.

Table 11-5 Register 2, 3, 5, 6, 8, 10, and 15 operations

Function	Rd	Instruction(s)
Read TTB register	TTB	MRC p15,0,Rd,c2,c0,0
Write TTB register	TTB	MCR p15,0,Rd,c2,c0,0
Read domain [15:0] access control	DAC	MRC p15,0,Rd,c3,c0,0
Write domain [15:0] access control	DAC	MCR p15,0,Rd,c3,c0,0
Read FSR	FSR	MRC p15,0,Rd,c5,c0,0
Write FSR	FSR	MCR p15,0,Rd,c5,c0,0
Read FAR	FAR	MRC p15,0,Rd,c6,c0,0

Table 11-5 Register 2, 3, 5, 6, 8, 10, and 15 operations (continued)

Function	Rd	Instruction(s)
Write FAR	FAR	`MCR p15,0,Rd,c6,c0,0`

Invalidate TLB	SBZ	`MCR p15,0,Rd,c8,c5,0`
		`MCR p15,0,Rd,c8,c6,0`
		`MCR p15,0,Rd,c8,c7,0`

Invalidate TLB single entry (using MVA)	MVA format	`MCR p15,0,Rd,c8,c5,1`
		`MCR p15,0,Rd,c8,c6,1`
		`MCR p15,0,Rd,c8,c7,1`

Read TLB lockdown	TLB lockdown	`MRC p15,0,Rd,c10,c0,0`
Write TLB lockdown	TLB lockdown	`MCR p15,0,Rd,c10,c0,0`
CAM read to C15.M	SBZ	`MCR p15,4,Rd,c15,c7,4`
CAM write	Tag, Size, V, P	`MCR p15,4,Rd,c15,c7,0`
RAM1 read to C15.M	SBZ	`MCR p15,4,Rd,c15,c11,4`
RAM1 write	Protection	`MCR p15,4,Rd,c15,c11,0`
RAM2 read to C15.M	SBZ	`MCR p15,4,Rd,c15,c3,5`
RAM2 write	PA Tag, Size	`MCR p15,4,Rd,c15,c3,1`
CAM match, RAM1 read to C15.M	MVA	`MCR p15,4,Rd,c15,c13,4`
Read C15.M	Data	`MRC p15,4,Rd,c15,c3,0`

Figure 11-12 shows the format of Rd for CAM writes and data for CAM reads.

31		10 9	6 5	4	3 2 1 0
	MVA TAG		SIZE_C	V	P	SBZ

Figure 11-12 Rd format, CAM write and data format, CAM read

In Figure 11-12 on page 11-14, V is the Valid bit, P is the Preserve bit, and SIZE_C sets the memory region size. The allowed values of SIZE_C are shown in Table 11-6.

Table 11-6 CAM memory region size

SIZE_C[3:0]	Memory region size
0b1111	1MB
0b0111	64KB
0b0011	16KB
0b0001	4KB
0b0000	1KB

Figure 11-13 shows the format of Rd for RAM1 writes.
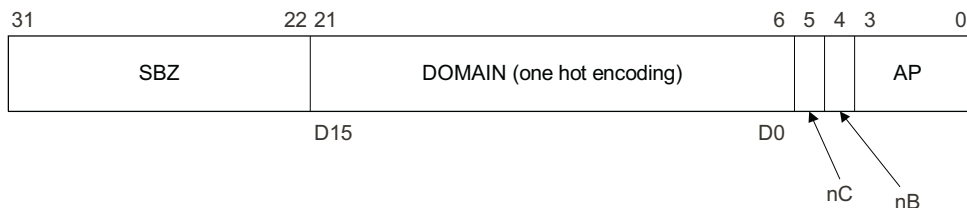
Figure 11-13 Rd format, RAM1 write

In Figure 11-13, AP[3:0] determines the setting of the access permission bits for a memory region. The allowed values are shown in Table 11-7.

Table 11-7 Access permission bit setting

AP[3:0]	Access permission bits
0b1000	0b11
0b0100	0b10
0b0010	0b01
0b0001	0b00

Figure 11-14 shows the data format for RAM1 reads.
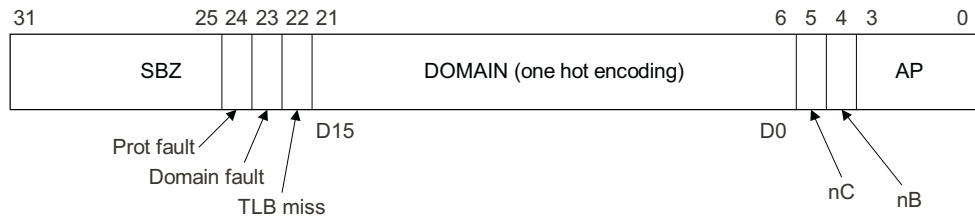
Figure 11-14 Data format, RAM1 read

In Figure 11-14, bits [24:22] are only valid for a match operation. In this case the values shown in Table 11-8 apply.

Table 11-8 Miss and fault encoding

Prot fault	Domain fault	TLB miss	Function
0	0	0	Hit, OK
0	1	0	Hit, domain fault
1	0	0	Hit, protection fault
1	1	0	Hit, protection and domain fault
-	-	1	TLB miss

Figure 11-15 shows the Rd format for RAM2 writes, and the data format for RAM2 reads.

Figure 11-15 Rd format, RAM2 write and data format, RAM2 read

 ARM DDI 0229A

In Figure 11-15 on page 11-16, SIZE_R2 sets the memory region size. The allowed values of SIZE_R2 are shown in Table 11-9.

Table 11-9 RAM2 memory region size

SIZE_R2[3:0]	Memory region size
0b1111	1MB
0b0111	64KB
0b0011	16KB
0b0000	4KB
0b0001	1KB

———— **Note** ————

The encoding for SIZE_R2 is different from SIZE_C.

11.5.1 Addressing the CAM, RAM1, and RAM2

For the CAM read or write, RAM1 read or write, and RAM2 read or write operations, you must specify the index. The CAM and RAM1 operations use the value in the victim pointer, so you must write this before any CAM or RAM1 operation. RAM2 uses a pipelined version of the victim pointer used for the CAM or RAM1 operation. This means that to read from index N in the RAM2 array, you must first perform an access to index N in either the CAM or RAM1.

The write TLB lockdown operation is:

```
MCR p15,0,Rd,c10,c0,0
```

The write TLB lockdown format for Rd is shown in Figure 11-16.

31 26	25 20	19 1	0
Base	Victim	SBZ	P
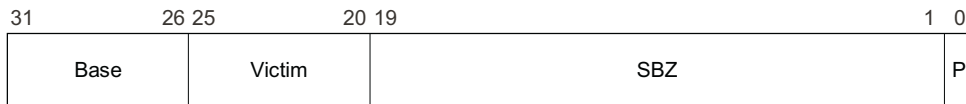
Figure 11-16 Rd format, write TLB lockdown

Example 11-2 on page 11-18 shows sample code for performing software test of the MMU. It contains typical operations with C15.M.

Example 11-2 MMU test operations

```
; MMU write, read and check for CAM, RAM1 and RAM2

; Load victim pointer with 0
        MOV r0,#0
        MCR p15,0,r0,c10,c0,0

; Write pattern 0x5A5A5A50 in CAM
; Write pattern 0x0025A5A5 in RAM1
; Write pattern 0xF0F0F0C0 in RAM2
        LDR r2,=0x5A5A5A50
        LDR r3,=0x0025A5A5
        LDR r4,=0xF0F0F0C0
        MOV r5,#64
; Write all 64 lines
loop0   MCR p15,4,r2,c15,c7,0       ; write CAM
        MCR p15,4,r3,c15,c11,0      ; write RAM1
        MCR p15,4,r4,c15,c3,1       ; write RAM2, pointer auto-incremented here
        SUBS r5,r5,#1
        BNE loop0

; Now read and check
; Reset victim pointer
        MOV r0,#0
        MCR p15,0,r0,c10,c0,0
        MOV r8,#64
loop1   MCR p15,4,r5,c15,c7,4       ; read CAM to C15.M
        MRC p15,4,r5,c15,c3,6       ; read C15.M to R5
        MCR p15,4,r6,c15,c11,4
        MRC p15,4,r6,c15,c3,6       ; read RAM1 to R6
        BIC r5,r5,#0x01c00000       ; mask fault/miss bits

        MCR p15,4,r7,c15,c3,5
        MRC p15,4,r7,c15,c3,6       ; read RAM2 to R7

        CMP r5,r2
        CMP r6,r3
        CMP r7,r4
        BNE TEST_FAIL

        SUBS r8,r8,#1
        BNE loop1
        B TEST_PASS
```

 ARM DDI 0229A

Appendix A
Signal Descriptions

This chapter describes the interface signals of the ARM720T processor. It contains the following sections:

- *AMBA interface signals* on page A-2
- *Coprocessor interface signals* on page A-3
- *JTAG and test signals* on page A-4
- *Debugger signals* on page A-6
- *Embedded trace macrocell interface signals* on page A-7
- *Miscellaneous signals* on page A-9.

A.1 AMBA interface signals

The AMBA interface signals are shown in Table A-1.

Table A-1 AMBA interface signals

Signal name	Type	Description
HCLK	Input	Bus clock. This is the only clock on the ARM720T (Rev 4) macrocell.
HADDR[31:0]	Output	32-bit system address bus.
HTRANS[1:0]	Output	Indicates type of current transfer.
HBURST[2:0]	Output	Indicates burst length of current transfer.
HWRITE	Output	Indicates direction of current transfer.
HSIZE[2:0]	Output	Indicates size of current transfer.
HPROT[3:0]	Output	Protection control signals
HGRANT	Input	Bus transfer granted.
HREADY	Input	Indicates that the current transfer has finished.
HRESP[1:0]	Input	Indicates transfer status.
HWDATA[31:0]	Output	Write data bus.
HRDATA[31:0]	Input	Read data bus.
HBUSREQ	Output	Bus transfer request.
HLOCK	Output	Indicates locked access.
HCLKEN	Input	Bus clock enable.
HRESETn	Input	Global reset.

A.2 Coprocessor interface signals

The coprocessor interface signals are shown in Table A-2.

Table A-2 Coprocessor interface signal descriptions

Name	Type	Description
EXTCPA	In	External coprocessor absent. This signal must be HIGH if no external coprocessor is present.
EXTCPB	In	External coprocessor busy.
EXTCPCLKEN	Out	External coprocessor clock enable.
EXTCPDIN[31:0]	Out	External coprocessor data in.
EXTCPDOUT[31:0]	In	External coprocessor data out.
CPnCPI	Out	Not coprocessor instruction. When LOW, this signal indicates that the ARM720T processor is executing a coprocessor instruction.
CPnOPC	Out	Not opcode fetch. When LOW, this signal indicates that the processor is fetching an instruction from memory. When HIGH, data, if present, is being transferred. This signal is used by the coprocessor to track the ARM pipeline.
CPTBIT	Out	Thumb state. This signal, when HIGH, indicates that the processor is executing the THUMB instruction set. When LOW, the processor is executing the ARM instruction set.
CPnTRANS	Out	Not coprocessor translate. When HIGH, the coprocessor interface is in a nonprivileged mode. When LOW, the coprocessor interface is in a privileged mode. The coprocessor samples this signal on every cycle when determining the coprocessor response.
CPnMREQ	Out	Not coprocessor memory request.
EXTCPDBE	In	External coprocessor data bus enable. This signal when HIGH, indicates that the coprocessor intends to drive the coprocessor data bus, **CPDATA**. If the coprocessor interface is not to be used then this signal must be tied LOW.

A.3 JTAG and test signals

JTAG and test signal descriptions are shown in Table A-3.

Table A-3 JTAG and test signal descriptions

Name	Type	Description
DBGIR[3:0]	Out	TAP instruction register. These signals reflect the current instruction loaded into the TAP controller instruction register. The signals change on the falling edge of **HCLK** when the TAP state machine is in the UPDATE-DR state. You can use these signals to enable more scan chains to be added using the ARM720T processor TAP controller.
DBGSREG[3:0]	Out	Scan chain register. These signals reflect the ID number of the scan chain currently selected by the TAP controller. These signals change on the falling edge of **XTCK** when the TAP state machine is in the UPDATE-DR state.
DBGSDIN	Out	Boundary scan serial data in. This signal is the serial data to be applied to an external scan chain.
DBGSDOUT	In	Boundary scan serial data out. This signal is the serial data from an external scan chain. It enables a single **DBGTDO** port to be used. If an external scan chain is not connected, this input must be tied LOW.
DBGTAPSM[3:0]	Out	Tap controller status. These signals represent the current state of the TAP controller machine. These signals change on the rising edge of **XTCK** and can be used to allow more scan chains to be added using the ARM720T processor TAP controller.
DBGCAPTURE[a]	Out	CAPTURE state signal. When HIGH, this indicates that the TAP controller state machine is in a CAPTURE state (see Figure 9-8 on page 9-27).
DBGSHIFT[a]	Out	SHIFT state signal. When HIGH, this indicates that the TAP controller state machine is in a SHIFT state (see Figure 9-8 on page 9-27).
DBGUPDATE[a]	Out	UPDATE state signal. When HIGH, this indicates that the TAP controller state machine is in an UPDATE state (see Figure 9-8 on page 9-27).
DBGINTEST[a]	Out	INTEST state signal.
DBGEXTEST[a]	Out	EXTEST state signal.

Table A-3 JTAG and test signal descriptions (continued)

Name	Type	Description
DBGnTDOEN	Out	Test data out enable.
DBGnTRST	In	Not test reset. When LOW, this signal resets the JTAG interface.
DBGTCKEN	In	Test clock enable.
DBGTDI	In	Test data in. JTAG test data in signal.
DBGTDO	Out	Test data out. JTAG test data out signal.
DBGTMS	In	Test mode select. JTAG test mode select signal.

a.   These signals are only active when scan chain 0 is selected.

A.4 Debugger signals

The debugger signal descriptions are shown in Table A-4.

Table A-4 Debugger signal descriptions

Name	Type	Description
DBGBREAK	In	Breakpoint.
		This signal enables external hardware to halt execution of the processor for debug purposes. When HIGH, this causes the current memory access to be breakpointed. If memory access is an instruction Fetch, the core enters debug state if the instruction reaches the Execute stage of the core pipeline. If the memory access is for data, the core enters the debug state after the current instruction completes execution. This enables extension of the internal breakpoints provided by the EmbeddedICE-RT module.
		In most systems, this input is tied LOW.
COMMRX	Out	Communication receive full.
		When HIGH, this signal denotes that the comms channel receive buffer contains data for the core to read.
COMMTX	Out	Communication transmit empty.
		When HIGH, this signal denotes that the comms channel transmit buffer is empty.
DBGACK	Out	Debug acknowledge.
		When HIGH, this signal denotes that the ARM is in debug state.
DBGEN	In	Debug enable.
		When **DBGEN** is LOW, it inhibits **BREAKPOINT** and **DBGRQ** to the core. **DBGACK** from the ARM720T processor is held LOW when **DGBEN** is LOW.
		DBGEN must be HIGH to enable the EmbeddedICE-RT logic to be used.
DBGRQ	In	Debug request.
		This signal causes the core to enter debug state after executing the current instruction. This enables external hardware to force the core into debug state, in addition to the debugging features provided by the EmbeddedICE-RT Logic.
		In most systems, this input is tied LOW.
DBGEXT[1:0]	In	External condition.
		These signals allow breakpoints and watchpoints to depend on an external condition.
DBGRNG[1:0]	Out	Range out.
		These signals indicate that the relevant EmbeddedICE-RT watchpoint register has matched the conditions currently present on the address, data, and control buses. These signals are independent of the state of the watchpoint enable control bits.

A.5 Embedded trace macrocell interface signals

The ETM interface signals are shown in Table A-5.

Table A-5 ETM interface signal descriptions

Output name	Type	Description
ETMnMREQ	Out	Not memory request. When LOW, indicates that the processor requires memory access during the following cycle.
ETMSEQ	Out	Sequential address. When HIGH, indicates that the address of the next memory cycle is related to that of the last memory cycle. The new address is one of the following: • the same as the previous one • four greater in ARM state • two greater in Thumb state. This signal can be used, with the low order address lines, to indicate that the next cycle can use a fast memory mode and bypass the address translation system.
ETMnEXEC	Out	Not executed. When HIGH, indicates that the instruction in the execution unit is not being executed. For example it might have failed the condition check code.
ETMnCPI	Out	Not coprocessor instruction. When the ARM720T processor executes a coprocessor instruction, it takes the **ETMnCPI** LOW and waits for a response from the coprocessor. The actions taken depend on this response, which the coprocessor signals on the **CPA** and **CPB** inputs.
ETMADDR[31:0]	Out	Addresses. This is the retimed internal address bus.
ETMnOPC	Out	Not opcode fetch. When LOW, indicates that the processor is fetching an instruction from memory. When HIGH, indicates that data, if present, is being transferred.
ETMDBGACK	Out	Debug acknowledge. When HIGH, indicates that the processor is in debug state. When LOW, indicates that the processor is in normal system state.
ETMABORT	Out	Memory abort or bus error. Indicates that a requested access has been disallowed.
ETMCPA	Out	Coprocessor absent handshake. The coprocessor absent signal. It is a buffered version of the coprocessor absent signal.
ETMCPB	Out	Coprocessor busy handshake. The coprocessor busy signal. It is a buffered version of the coprocessor absent signal.
ETMPROCID[31:0]	Out	Trace PROCID bus.
ETMPROCIDWR	Out	Trace PROCID write. Indicates to the ETM7 that the Trace PROCID, CP15 register 13, has been written.

Output name	Type	Description
ETMTBIT	Out	Thumb state. This signal, when HIGH, indicates that the processor is executing the THUMB instruction set. When LOW, the processor is executing the ARM instruction set.
ETMBIGEND	Out	Big-endian format. When this signal is HIGH, the processor treats bytes in memory as being in big-endian format. When it is LOW, memory is treated as little-endian.
ETMEN	In	The ETM7 enable signal.
ETMHIVECS	Out	When LOW, this signal indicates that the exception vectors start at address 0x00000000. When HIGH, the exception vectors start at address 0xFFFF0000.
ETMSIZE[1:0]	Out	The memory access size bus driven by the ARM720T (Rev 4) processor.
ETMRDATA[31:0]	Out	The processor read data bus.
ETMWDATA[31:0]	Out	The processor write data bus.
ETMINSTRVALID	Out	The instruction valid signal driven by the ARM720T processor. When HIGH, it indicates that the instruction in the Execute stage is valid and has not been flushed.
ETMnRW	Out	Not read/write. When HIGH, indicates a processor write cycle. When LOW, indicates a processor read cycle.
ETMCLKEN	Out	This signal is used to indicate to the ETM that the core is in a wait state. It is not a true clock enable for the ETM.

A.6 Miscellaneous signals

Miscellaneous signals used by the ARM720T processor are shown in Table A-6.

Table A-6 Miscellaneous signal descriptions

Name	Type	Description
BIGENDOUT	Out	Big-endian format. When this signal is HIGH, the processor treats bytes in memory as being in big-endian format. When it is LOW, memory is treated as little-endian.
nFIQ	In	ARM fast interrupt request signal.
nIRQ	In	ARM interrupt request signal.
VINITHI	In	Determines the state of the V bit in CP15 register 1 at reset. When HIGH, the V bit is set coming out of rest. When LOW, the V bit is clear coming out of reset.

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   Is caused by an illegal memory access. Abort can be caused by the external memory system, an external MMU, or the EmbeddedICE-RT logic.

Addressing modes   A procedure 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.

Arithmetic Logic Unit

The part of a computer that performs all arithmetic computations, such as addition and multiplication, and all comparison operations.

ALU   *See* Arithmetic Logic Unit.

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

Big-endian   Memory organization where the least significant byte of a word is at a higher address than the most significant byte.

Banked registers Register numbers whose physical register is defined by the current processor mode. The banked registers are registers R8 to R14, or R13 to R14, depending on the processor mode.

Breakpoint A location in the program. If execution reaches this location, the debugger halts execution of the code image.

See also Watchpoint.

CISC *See* Complex Instruction Set Computer.

Complex Instruction Set Computer

A microprocessor that recognizes a large number of instructions.

See also Reduced Instruction Set Computer.

CPSR *See* Program Status Register.

Control bits The bottom eight bits of a program status register. The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Current Program Status Register

See Program Status Register.

DCC Debug Communications Channel.

Debug state A condition that allows the monitoring and control of the execution of a processor. Usually used to find errors in the application program flow. A processor enters debug state from *halt mode* and not from *monitor mode*.

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

EmbeddedICE The EmbeddedICE logic is controlled via the JTAG test access port, using a protocol converter such as MultiICE: an extra piece of hardware that allows software tools to debug code running on a target processor.

See also ICE and JTAG.

EmbeddedICE-RT A version of EmbeddedICE logic that has improved support for real-time debugging.

Exception modes Privileged modes that are entered when specific exceptions occur.

Exception Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

External abort An abort that is generated by the external memory system.

FIQ Fast interrupt.

Halt mode One of two debugging modes. When debugging is performed in halt mode, the core stops when it encounters a watchpoint or breakpoint, and is isolated from the rest of the system. See also *Monitor mode*.

ICE *See* In-circuit emulator.

Idempotent A mathematical quantity that when applied to itself under a given binary operation equals itself.

In-circuit emulator An *In-Circuit Emulator* (ICE), is a device that aids the debugging of hardware and software. Debuggable ARM processors such as the ARM720T (Rev 4) processor have extra hardware to assist this process.

See also EmbeddedICE-RT.

IRQ Interrupt request.

Joint Test Action Group

The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices.

JTAG See *Joint Test Action Group*.

Link register This register holds the address of the next instruction after a branch with link instruction.

Little-endian memory

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

LR *See* Link register

Macrocell A complex logic block with a defined interface and behavior. A typical VLSI system will comprise several macrocells (such as an ARM7TDMI-S core, an ETM7, and a memory block) plus application-specific logic.

Memory Management Unit

Allows control of a memory system. Most of the control is provided through translation tables held in memory.

MMU *See* Memory Management Unit

Monitor mode   One of two debugging modes. When debugging is performed in monitor mode, the core does not stop when it encounters a watchpoint or breakpoint, but enters an abort exception routine. See also *Halt mode*.

PC   *See* Program Counter.

Privileged mode   Any processor mode other than User mode. Memory systems typically check memory accesses from privileged modes against supervisor access permissions rather than the more restrictive user access permissions. The use of some instructions is also restricted to privileged modes.

Processor Status Register

See Program Status Register

Program Counter   Register 15, the Program Counter, is used in most instructions as a pointer to the instruction that is two instructions after the current instruction.

Program Status Register

Contains some information about the current program and some information about the current processor. Also referred to as Processor Status Register.

Also referred to as *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR). The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned.

PSR   *See* Program Status Register.

RAZ   Read as zero.

Reduced Instruction Set Computer

A type of microprocessor that recognizes a lower number of instructions in comparison with a Complex Instruction Set Computer. The advantages of RISC architectures are:
- they can execute their instructions very fast because the instructions are so simple
- they require fewer transistors, this makes them cheaper to produce and more power efficient.

See also Complex Instruction Set Computer.

RISC   *See* Reduced Instruction Set Computer

Saved Program Status Register

The Saved Program Status Register which is associated with the current processor mode and is undefined if there is no such Saved Program Status Register, as in User mode or System mode.

See also Program Status Register.

SBO   *See* Should Be One fields.

SBZ   *See* Should Be Zero fields.

Should Be One fields

Should be written as one (or all ones for bit fields) by software. Values other than one produces unpredictable results.

See also Should Be Zero fields.

Should Be Zero fields

Should be written as zero (or all 0s for bit fields) by software. Values other than zero produce unpredictable results.

See also Should Be One fields.

Software Interrupt Instruction

This instruction (SWI) enters Supervisor mode to request a particular operating system function.

SPSR   *See* Saved Program Status Register.

Stack pointer   A register or variable pointing to the top of a stack. If the stack is full stack the SP points to the most recently pushed item, else if the stack is empty, the SP points to the first empty location, where the next item will be pushed.

Status registers *See* Program Status Register.

SP   *See* Stack pointer

SWI   *See* Software Interrupt Instruction.

TAP   See *Test access port*.

Test Access Port The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **nTRST**.

Thumb instruction A halfword which 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) instructions is operating in Thumb state.

UND   *See* Undefined.

Undefined   Indicates an instruction that generates an undefined instruction trap.

UNP	*See* Unpredictable
Unpredictable	Means the result of an instruction cannot be relied upon. Unpredictable instructions must not halt or hang the processor, or any parts of the system.
Unpredictable fields	Do not contain valid data, and a value can vary from moment to moment, instruction to instruction, and implementation to implementation.
Watchpoint	A location in the image that is monitored. If the value stored there changes, the debugger halts execution of the image.
	See also Breakpoint.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.