ARM7EJ-S

(Rev 1)

Technical Reference Manual

ARM®

ARM7EJ-S
Technical Reference Manual

Copyright © 2001 ARM Limited. All rights reserved.

Release Information

Contents
ARM7EJ-S Technical Reference Manual

ARM DDI 0214B

Appendix A   Signal Descriptions

Appendix B   Debug in Depth

Glossary

List of Tables
ARM7EJ-S Technical Reference Manual

List of Figures
ARM7EJ-S Technical Reference Manual

Preface

This preface introduces the ARM7EJ-S 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 the technical reference manual for the ARM7EJ-S processor.

Intended audience

This document has been written for hardware and software engineers who want to design or develop products based upon the ARM7EJ-S processor. It assumes no prior knowledge of ARM products.

Using this manual

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM7EJ-S processor and a summary of the instruction set.

Chapter 2 *Programmer's Model*

Read this chapter for a description of the ARM7EJ-S processor from the point of view of the programmer.

Chapter 4 *Interrupts*

Read this chapter for a description of interrupt operation, including interrupt latency.

Chapter 3 *Memory Interface*

Read this chapter for a description of the memory signals, including descriptions of the sequential, nonsequential, internal, and coprocessor register transfer bus cycles.

Chapter 5 *Coprocessor Interface*

Read this chapter for a description of the coprocessor interface, including timing diagrams for coprocessor operations.

Chapter 6 *Debug Interface and EmbeddedICE-RT*

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

Chapter 7 *Embedded Trace Macrocell Interface*

Read this chapter for a description of the Embedded Trace Macrocell interface and signals.

Chapter 8 *Device Reset*

> Read this chapter for a description of the reset behavior of the ARM7EJ-S processor.

Chapter 9 *Instruction Cycle Times*

> Read this chapter for a summary of instruction cycle timings and a description of interlocks.

Chapter 10 *AC Parameters*

> Read this chapter for the main AC timing parameters of the ARM7EJ-S processor.

Appendix A *Signal Descriptions*

> Read this chapter for a description of all the ARM7EJ-S processor interface signals.

Appendix B *Debug in Depth*

> Read this chapter for a detailed description of the debug interface and additional information about the EmbeddedICE-RT logic.

Typographical conventions

The following typographical conventions are used in this book:

bold   Highlights ARM processor signal names, and interface elements, such as menu names and buttons. 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 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 some timing diagrams. The following key explains the components used in these diagrams. Any variations are clearly labeled when they occur. Therefore, you must not attach any additional meaning 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 |
| Valid (correct) sampling point |
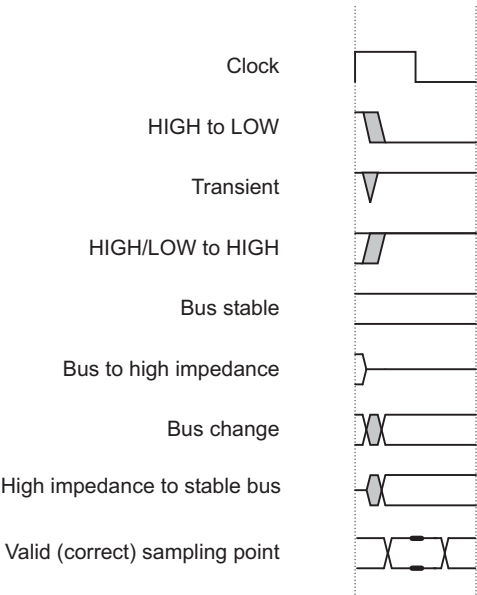
Key to timing diagram conventions

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

Further reading

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

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

 ARM DDI 0214B

ARM publications

This document contains information that is specific to the ARM7EJ-S processor. Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ETM9 (Rev 2a) Technical Reference Manual* (ARM DDI 0157)
- *ARM Software Development Kit User Guide* (ARM DUI 0040).

Other publications

This section lists relevant documents published by third parties.

- IEEE Std. 1149.1- 1990, *Standard Test Access Port and Boundary-Scan Architecture*.
- *The Java^{TM} Virtual Machine Specification*, Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303 U.S.A.

Feedback

ARM Limited welcomes feedback both on the ARM7EJ-S processor, and on the documentation.

Feedback on the ARM7EJ-S 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 Technical Reference Manual

If you have any comments about this document, please send email to errata@arm.com giving:
• the document title
• the document number
• the page number(s) to which your comments refer
• a concise explanation of your comments.

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM7EJ-S processor with Jazelle extensions. It contains the following sections:

- *About the ARM7EJ-S processor with Jazelle technology* on page 1-2
- *ARM7EJ-S processor architecture* on page 1-5
- *ARM7EJ-S processor block, core, and interface diagrams* on page 1-7
- *ARM7EJ-S processor instruction set summary* on page 1-10.

1.1 About the ARM7EJ-S processor with Jazelle technology

The ARM7EJ-S processor has the ARMv5TEJ architecture with Jazelle technology featuring an enhanced multiplier design for improved *Digital Signal Processing* (DSP) performance.The Jazelle technology enables direct execution of Java bytecodes on ARM processors, providing the performance for the next generation of Java-powered wireless and embedded devices.

The ARM7EJ-S processor is a member of the ARM family of general-purpose 32-bit microprocessors. The ARM family of processors offers high performance for very low power consumption and gate count.

The ARM architecture is based on *Reduced Instruction Set Computer* (RISC) principles. The reduced instruction set and related decode mechanism are much simpler than those of *Complex Instruction Set Computer* (CISC) designs. This simplicity gives:

- a high instruction throughput
- an excellent real-time interrupt response
- a small, cost-effective, processor macrocell.

1.1.1 The instruction pipelines

The ARM7EJ-S processor uses a pipeline to increase the speed of the flow of instructions to the processor. This enables several operations to take place simultaneously, and the processing and memory systems to operate continuously.

A five-stage ARM state pipeline is used, consisting of Fetch, Decode, Execute, Memory, and Writeback stages. This is shown in Figure 1-1 on page 1-3.

A six-stage pipeline is used in Jazelle state, consisting of Fetch, Jazelle, Decode, Execute, Memory, and Writeback stages. This is shown in Figure 1-2 on page 1-4.

Instruction address in

ARM state	**Thumb state**		
PC	PC	Fetch	Instruction fetched from memory
PC - 4	PC - 2	Decode	Registers used in the instruction are decoded
PC - 8	PC - 4	Execute	Shift and ALU operation
PC - 12	PC - 6	Memory	Data access to/from memory
PC - 16	PC - 8	Writeback	Write back to register bank
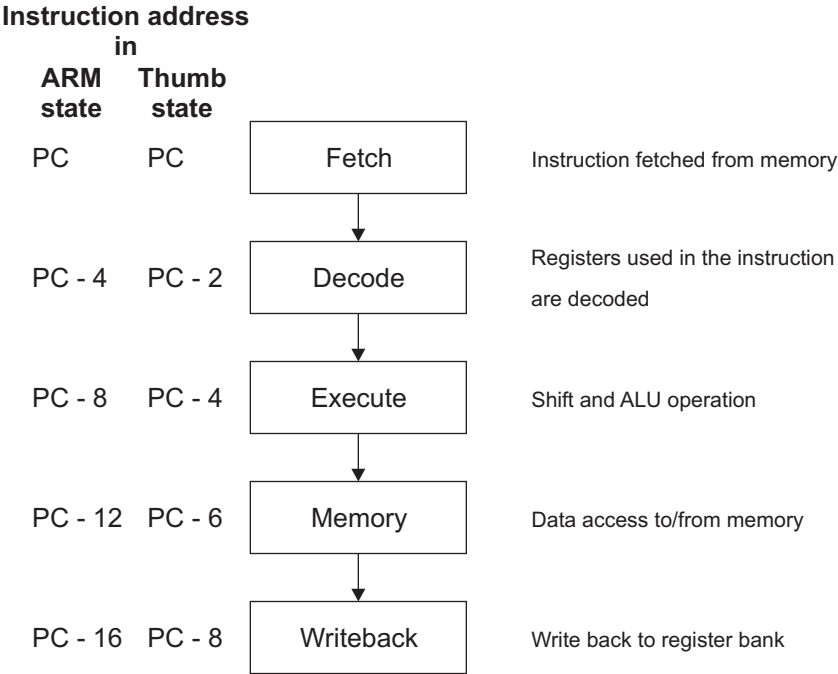
Figure 1-1 Five-stage pipeline

During normal operation:

- one instruction is being fetched from memory
- the previous instruction is being decoded
- the instruction before that is being executed
- the instruction before that is performing data accesses (if applicable)
- the instruction before that is writing its data back to the register bank.

——— **Note** ———

The program counter points to the instruction being fetched rather than to the instruction being executed.

Fetch	Instruction fetched from memory
↓	
Jazelle	Decoding of Java bytecode and operands
↓	
Decode	Registers used in the instruction are decoded
↓	
Execute	Shift and ALU operation
↓	
Memory	Data access to/from memory
↓	
Writeback	Write back to register bank
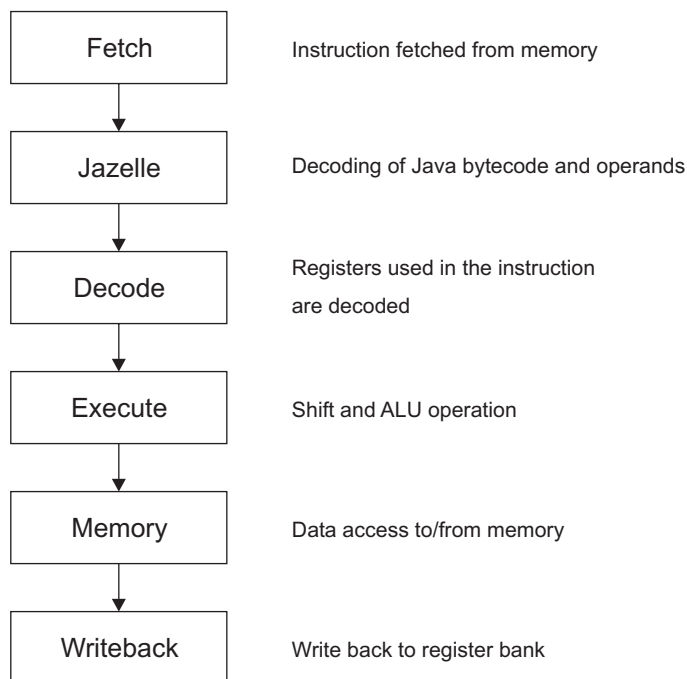
Figure 1-2 Six-stage Jazelle pipeline

1.1.2   Memory access

The ARM7EJ-S processor has a von Neumann architecture. This features a single 32-bit data bus that carries both instructions and data.

Only load, store, and swap instructions can access data from memory. Data can be 8-bit bytes, 16-bit halfwords, or 32-bit words. Words must be aligned to 4-byte boundaries. Halfwords must be aligned to 2-byte boundaries.

1.1.3   Forwarding, interlocking, and data dependencies

Due to the nature of the five-stage pipeline, a value can be required for use before it has been placed in the register bank by the actions of an earlier instruction. The ARM7EJ-S processor control logic automatically detects these cases and stalls the core or forwards data as applicable to overcome these hazards. No intervention is required by software in these cases, although you can improve software performance by re-ordering instructions in certain situations.

1.2 ARM7EJ-S processor architecture

The ARM7EJ-S processor has three instruction sets:

- the 32-bit ARM instruction set used in ARM state
- the 16-bit Thumb instruction set used in Thumb state
- the 8-bit Java bytecode used in Jazelle state.

The ARM7EJ-S processor uses the v5TEJ implementation of the ARM architecture. For details of both the ARM and Thumb instruction sets, refer to the *ARM Architecture Reference Manual*. For full details of the Jazelle instruction set, refer to `http://java.sun.com`.

This section describes:

- *Instruction compression*
- *The Thumb instruction set*.

1.2.1 Instruction compression

A typical 32-bit architecture can manipulate 32-bit integers with single instructions, and address a large address space much more efficiently than a 16-bit architecture. When processing 32-bit data, a 16-bit architecture takes at least two instructions to perform the same task as a single 32-bit instruction.

When a 16-bit architecture has only 16-bit instructions, and a 32-bit architecture has only 32-bit instructions, overall the 16-bit architecture has higher code density, and greater than half the performance of the 32-bit architecture.

Thumb implements a 16-bit instruction set on a 32-bit architecture, giving higher performance than on a 16-bit architecture and with higher code density than a 32-bit architecture.

The ARM7EJ-S processor gives you the choice of running in ARM state, or Thumb state, or a mix of the two. This enables you to optimize both code density and performance to best suit your application requirements.

1.2.2 The Thumb instruction set

The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on the processor model. Thumb instructions operate with the standard ARM register configuration, enabling excellent interoperability between ARM and Thumb states.

The Thumb instruction set has all the advantages of a 32-bit core:

- 32-bit address space
- 32-bit registers
- 32-bit shifter and *Arithmetic Logic Unit* (ALU)
- 32-bit memory transfer.

The Thumb instruction set therefore offers a long branch range, powerful arithmetic operations, and a large address space.

Thumb code is typically 65% of the size of the ARM code, and provides 160% of the performance of ARM code when running on a 16-bit memory system. The Thumb instruction set makes the ARM7EJ-S processor ideally suited to embedded applications with restricted memory bandwidth, where code density is important.

There is no performance penalty when moving between ARM and Thumb state.

The availability of both 16-bit Thumb and 32-bit ARM instruction sets, gives designers the flexibility to emphasize performance or code size on a subroutine level, according to the requirements of their applications. For example, critical loops for fast interrupts and DSP algorithms can be coded using the full ARM instruction set, and linked with Thumb code.

1.3   ARM7EJ-S processor block, core, and interface diagrams

The ARM7EJ-S processor architecture, core, and interface diagrams are shown in the following figures:

*   the *Block diagram* is shown in Figure 1-3
*   the *Core block diagram* on page 1-8 is shown in Figure 1-4 on page 1-8
*   the *Interface diagram* on page 1-9 is shown in Figure 1-5 on page 1-9.

Figure 1-3 Block diagram

Refer to Chapter 6 *Debug Interface and EmbeddedICE-RT* for a description of the EmbeddedICE-RT logic.

Figure 1-4 Core block diagram

 ARM DDI 0214B
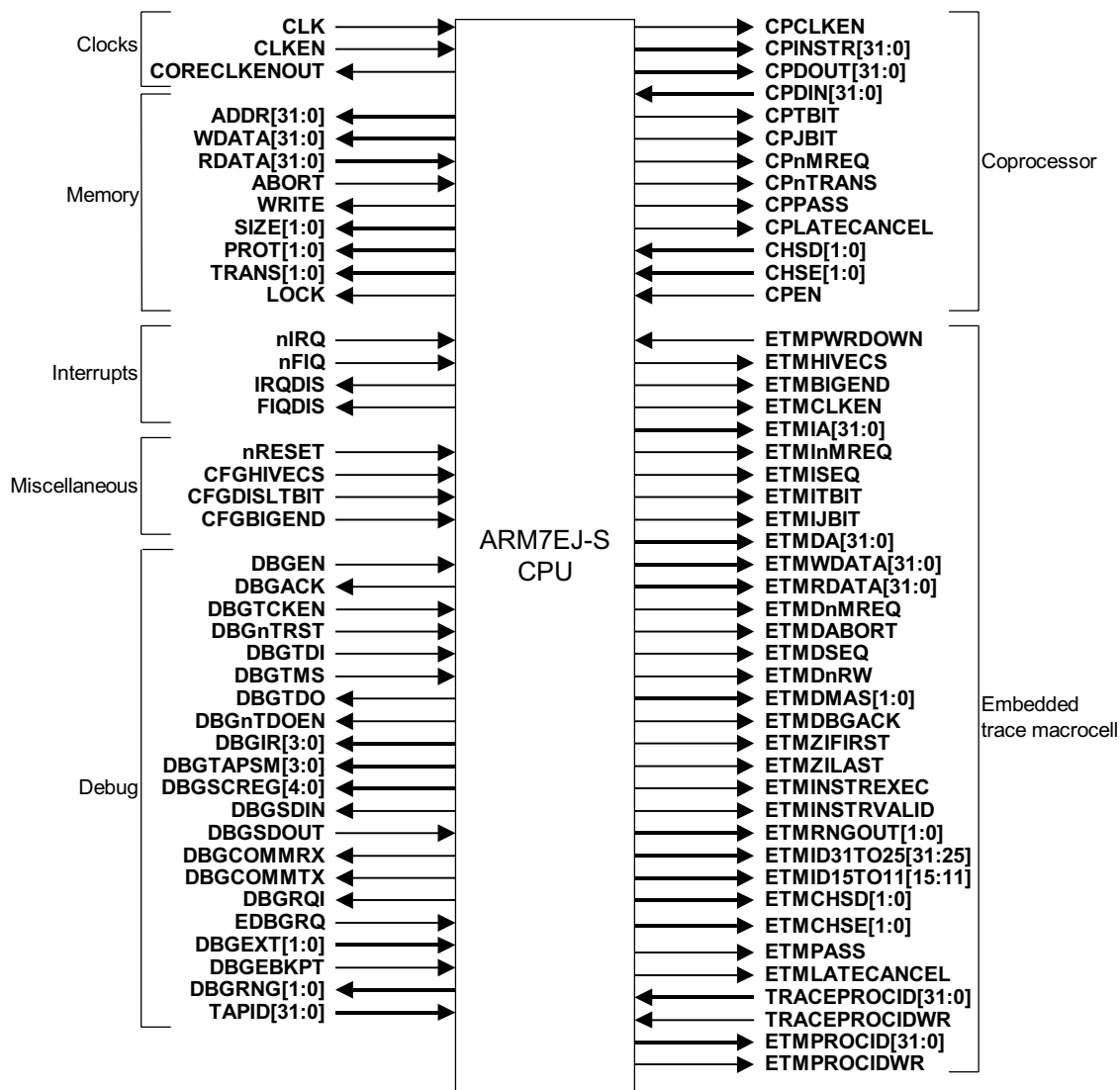
Figure 1-5 Interface diagram

1.4 ARM7EJ-S processor instruction set summary

This section provides a description of the instruction sets used on the ARM7EJ-S processor:

- *Format summary*
- *Extended ARM instruction set summary* on page 1-11
- *Thumb instruction set summary* on page 1-19.

1.4.1 Format summary

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

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

The ARM7EJ-S processor uses an implementation of the ARMv5TE with ARM Jazelle technology (ARMv5TEJ). For a description of the ARM Thumb refer to the *ARM Architecture Reference Manual*. For a description of the Jazelle technology, refer to `http://java.sun.com`.

Table 1-1 Key to instruction set tables

Symbol	Description
{cond}	See Table 1-6 on page 1-18.
<Oprnd2>	See Table 1-4 on page 1-17.
{field}	See Table 1-5 on page 1-18.
S	Sets condition codes (optional).
B	Byte operation (optional).
H	Halfword operation (optional).
T	Forces address translation. Cannot be used with pre-indexed addresses.
Addressing modes	See Table 1-3 on page 1-15.
#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 }).
x	Selects HIGH or LOW 16 bits of register Rm. T selects the HIGH 16 bits. (T = top) B selects the LOW 16 bits. (B = bottom).
y	Selects HIGH or LOW 16 bits of register Rs. T selects the HIGH 16 bits. (T = top) B selects the LOW 16 bits. (B = bottom).

1.4.2   Extended ARM instruction set summary

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

Table 1-2 ARM instruction set 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_flg, #32bit_Imm`
	Move immediate to CPSR flags	`MSR{cond} CPSR_flg, #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>`
	Reverse subtract	`RSB{cond}{S} Rd, Rn, <Oprnd2>`
	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>`
	Saturating add	`QADD{cond} Rd, Rn, Rs`

Table 1-2 ARM instruction set summary (continued)

Operation		Assembler
	Saturating add with double	`QDADD{cond} Rd, Rn, Rs`
	Saturating subtract	`QSUB{cond} Rd, Rn, Rs`
	Saturating subtract with double	`QDSUB{cond} Rd, Rn, Rs`
	Multiply 16x16	`SMULxy{cond} Rd, Rm, Rs`
	Multiply accumulate 16x16+32	`SMULAxy{cond} Rd, Rm, Rs, Rn`
	Multiply 32x16	`SMULWx{cond} Rd, Rm, Rs`
	Multiply accumulate 32x16+32	`SMLAWx{cond} Rd, Rm, Rs, Rn`
	Multiply signed accumulate long 16x16+64	`SMLALx{cond} RdLo, RdHi, Rm, Rs`
	Count leading zeros	`CLZ{cond} Rd, Rm`
Logical	Test	`TST{cond} Rn, <Oprnd2>`
	Test equivalence	`TEQ{cond} Rn, <Oprnd2>`
	AND	`AND{cond}{S} Rd, Rn, <Oprnd2>`
	XOR	`EOR{cond}{S} Rd, Rn, <Oprnd2>`
	OR	`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	`BX{cond} Rn`
	Branch, link and exchange	`BLX{cond} label`
	Branch, link and exchange	`BLX{cond} Rn`
	Branch and exchange to Jazelle state	`BXJ{cond} Rm`
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>`

Table 1-2 ARM instruction set summary (continued)

Operation		Assembler
	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	`LDM{cond}<a_mode4L> Rd{!}, <reglist>`
	• 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 and restore CPSR	`LDM{cond}<a_mode4L> Rd{!}, <reglist+PC>^`
	• User registers	`LDM{cond}<a_mode4L> Rd{!}, <reglist>^`
	• Load double	`LDR{cond}D Rd, <a_mode3>`
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>`
	Byte with User mode privilege	`STR{cond}BT Rd, <a_mode2P>`
	Halfword	`STR{cond}H Rd, <a_mode3>`
	Multiple block data operations	`STM{cond}<a_mode4S> Rd{!}, <reglist>`
	• 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>{^}`
	• User registers	`STM{cond}<a_mode4S> Rd{!}, <reglist>^`
	• Store double	`STR{cond}D Rd, <a_mode3>`
Soft preload	Memory may prepare to load from address	`PLD <a_mode2>`
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`

Table 1-2 ARM instruction set summary (continued)

Operation		Assembler
	Byte	`SWP{cond}B Rd, Rm, [Rn]`
Coprocessors	Data operations	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`
	Move to ARM reg from coprocessor	`MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Move to coprocessor from ARM register	`MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>`
	Move double to ARM register from coprocessor	`MRRC{cond} p<cpnum>, <op1>, Rd, Rn, CRm`
	Move double to coprocessor from ARM register	`MCRR{cond} p<cpnum>, <op1>, Rd, Rn, CRm`
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode5>`
	Store	`STC{cond} p<cpnum>, CRd, <a_mode5>`
Software interrupt	-	`SWI{cond} 24bit_Imm`
Software breakpoint	-	`BKPT<immediate>`

Addressing modes

The addressing modes enable all load and store addresses to be determined from register contents and instruction fields only. The five addressing modes used by the ARM7EJ-S processor are:

Mode 1   Shifter operands for data processing instructions.

Mode 2   Load and store word or unsigned byte.

Mode 3   Load and store halfword or load signed byte.

Mode 4   Load and store multiple.

Mode 5   Load and store coprocessor.

The addressing modes are shown with their types and mnemonics in Table 1-3.

Table 1-3 Addressing modes

Addressing mode	Type or addressing mode	Mnemonic or stack type
Mode 2 <a_mode2>	Immediate offset	`[Rn, #+/-12bit_Offset]`
	Register offset	`[Rn, +/-Rm]`
	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]`
	Pre-indexed offset	-
	Immediate	`[Rn, #+/-12bit_Offset]!`
	Register	`[Rn, +/-Rm]!`
	Scaled register	`[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 offset	-
	Immediate	`[Rn], #+/-12bit_Offset`
	Register	`[Rn], +/-Rm`
	Scaled register	`[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`

Table 1-3 Addressing modes (continued)

Addressing mode	Type or addressing mode	Mnemonic or stack type
Mode 2 (privileged <a_mode2P>)	Immediate offset	`[Rn, #+/-12bit_Offset]`
	Register offset	`[Rn, +/-Rm]`
	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 offset	-
	Immediate	`[Rn], #+/-12bit_Offset`
	Register	`[Rn], +/-Rm`
	Scaled register	`[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`
Mode 3 <a_mode3>	Immediate offset	`[Rn, #+/-8bit_Offset]`
	Pre-indexed	`[Rn, #+/-8bit_Offset]!`
	Post-indexed	`[Rn], #+/-8bit_Offset`
	Register offset	`[Rn, +/-Rm]`
	Pre-indexed	`[Rn, +/-Rm]!`
	Post-indexed	`[Rn], +/-Rm`
Mode 4 (load) <a_mode4L>	`IA` Increment after	`FD` Full descending
	`IB` Increment before	`ED` Empty descending
	`DA` Decrement after	`FA` Full ascending

 ARM DDI 0214B

Table 1-3 Addressing modes (continued)

Addressing mode	Type or addressing mode	Mnemonic or stack type
	DB Decrement before	EA Empty ascending
Mode 4 (store) <a_mode4S>	IA Increment after	EA Empty ascending
	IB Increment before	FA Full ascending
	DA Decrement after	ED Empty descending
	DB Decrement before	FD Full descending
Mode 5 <a_mode5>)	Immediate offset	[Rn, #+/-(8bit_Offset*4)]
	Pre-indexed	[Rn, #+/-(8bit_Offset*4)]!
	Post-indexed	[Rn], #+/-(8bit_Offset*4)

Operand 2

An operand is the part of the instruction that references data or a peripheral device. Operand2 is shown in Table 1-4.

Table 1-4 Operand2

Operand	Type	Mnemonic
Operand 2 <oprnd2>	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

Fields are shown in Table 1-5.

Type	Suffix	Sets	Bit
Field {field}	_c	Control field mask bit (bit 0)	3
	_x	Extension field mask bit (bit 1)	0
	_s	Status field mask bit (bit 2)	1
	_f	Flags field mask bit (bit 3)	2

Condition fields

Condition fields are shown in Table 1-6.

Table 1-6 Condition fields

Field type	Suffix	Description	Condition
Condition {cond}	EQ	Equal	Z set
	NE	Not equal	Z clear
	HS/CS	Unsigned higher or same	C set
	LO/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 and V set, or N and V clear
	LT	Less than	N set and V clear, or N clear and V set

Table 1-6 Condition fields (continued)

Field type	Suffix	Description	Condition
	GT	Greater than	Z clear and (N and V set, or N and V clear)
	LE	Less than or equal	Z set or (N set and V clear) or (N clear and V set)
	AL	Always	Flag ignored

1.4.3 Thumb instruction set summary

The Thumb instruction set summary is shown in Table 1-7.

Table 1-7 Thumb instruction set 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

Table 1-7 Thumb instruction set summary (continued)

Operation		Assembler
	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`
	XOR	`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`

Table 1-7 Thumb instruction set summary (continued)

Operation		Assembler
	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 or 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 and V set, or If Z clear, and N and V clear	`BGT label`
	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`
	Long branch, link and exchange instruction	`BLX label`
Branch and exchange	To address held in Low reg	`BX Rs`
	To address held in High reg	`BX Hs`
Branch, link, and exchange	To address held in Low reg	`BLX Rs`
	To address held in High reg	`BLX 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	-

Table 1-7 Thumb instruction set summary (continued)

Operation		Assembler
	• Word	`LDR Rd, [Rb, Ro]`
	• Halfword	`LDRH Rd, [Rb, Ro]`
	• Halfword signed	`LDRSH Rd, [Rb, Ro]`
	• Byte	`LDRB Rd, [Rb, Ro]`
	• Byte signed	`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]`
	• 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>`

Table 1-7 Thumb instruction set summary (continued)

Operation		Assembler
	Pop registers and PC from stack	`POP <reglist, PC>`
Software interrupt	-	`SWI 8bit_Imm`
Software breakpoint	-	`BKPT<immediate>`

Chapter 2
Programmer's Model

This chapter describes the ARM7EJ-S processor programmer's model. It contains the following sections:

- *About the programmer's model* on page 2-2
- *Processor operating states* on page 2-3
- *Memory formats* on page 2-4
- *Data types* on page 2-7
- *Operating modes* on page 2-8
- *Registers* on page 2-9
- *The program status registers* on page 2-15
- *Exceptions* on page 2-20.

2.1   About the programmer's model

The ARM7EJ-S processor implements the ARMv5TE architecture with Jazelle technology extensions. This includes the 32-bit ARM instruction set, 16-bit Thumb instruction set, and the 8-bit Jazelle instruction set. For details of both the ARM and Thumb instruction sets, refer to the *ARM Architecture Reference Manual*. For the Jazelle instruction set, contact ARM Limited.

                   ARM DDI 0214B

2.2 Processor operating states

The ARM7EJ-S processor has three operating states:

ARM state   32-bit, word-aligned ARM instructions are executed in this state.

Thumb state   16-bit, halfword-aligned Thumb instructions.

Jazelle state   Variable length, byte-aligned Jazelle instructions.

In Thumb state, bit 1 of the *Program Counter* (PC) is used to select between alternate halfwords.

In Jazelle state, all instruction fetches are in words.

——— **Note** ———

Transition between ARM, Thumb, and Jazelle states does not affect the processor mode or the register contents.

2.2.1 Switching state

You can switch the operating state of the ARM7EJ-S core between:

- ARM state and Thumb state using the BX and BLX instructions, and loads to the PC. Switching state is described in the *ARM Architecture Reference Manual*.
- ARM state and Jazelle state using the BXJ instruction.

  All exceptions are entered, handled, and exited in ARM state. If an exception occurs in Thumb state or Jazelle state, the processor reverts to ARM state. The transition back to Thumb or Jazelle states occurs automatically on return from the exception handler.

2.2.2 Interworking ARM and Thumb state

The ARM7EJ-S processor enables you to mix ARM and Thumb code as you want. For details see Chapter 7 *Interworking ARM and Thumb* in the *Software Development Kit User Guide*.

2.3   Memory formats

The processor views memory as a linear collection of bytes numbered in ascending order from zero. For example:

*   bytes 0 to 3 hold the first stored word
*   bytes 4 to 7 hold the second stored word.

The processor is bi-endian and can treat words in memory as being stored in either:

*   *Little-endian format*
*   *Big-endian format*.

2.3.1   Little-endian format

In little-endian format, the lowest addressed byte in a word is considered the least-significant byte of the word and the highest addressed byte is the most significant. So the byte at address 0 of the memory system connects to data lines 7 through 0.

For a word-aligned address A, Figure 2-1 shows how the word at address A, the halfword at addresses A and A+2, and the bytes at addresses A, A+1, A+2, and A+3 map on to each other when the core is configured as little-endian.

31 24	23 16	15 8	7 0
Word at address A			
Halfword at address A+2		Halfword at address A	
Byte at address A+3	Byte at address A+2	Byte at address A+1	Byte at address A
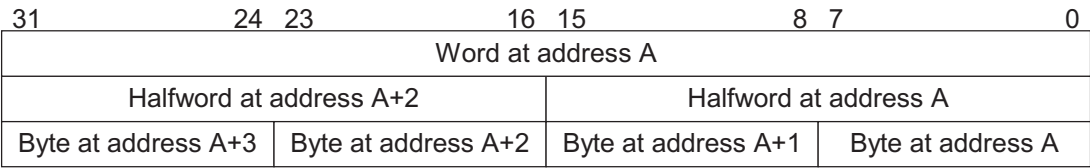
Figure 2-1 Little-endian addresses of bytes and halfwords within words

2.3.2   Big-endian format

In big-endian format, the processor stores the most significant byte of a word at the lowest-numbered byte, and the least significant byte at the highest-numbered byte. So the byte at address 0 of the memory system connects to data lines 31 through 24.

For a word-aligned address A, Figure 2-2 on page 2-5 shows how the word at address A, the halfword at addresses A and A+2, and the bytes at addresses A, A+1, A+2, and A+3 map on to each other when the core is configured as big-endian.

31	24	23	16	15	8	7	0
Word at address A							
Halfword at address A				Halfword at address A+2			
Byte at address A		Byte at address A+1		Byte at address A+2		Byte at address A+3	
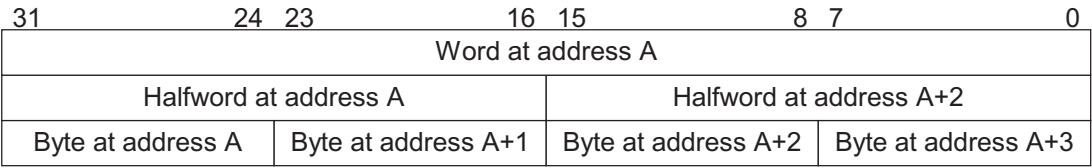
Figure 2-2 Big-endian addresses of bytes and halfwords within words

2.4   Instruction length

Instructions are either:

- 32 bits long (in ARM state)
- 16 bits long (in Thumb state)
- variable length, multiples of 8 bits (in Jazelle state).

2.5   Data types

The ARM7EJ-S processor supports the following data types:
- word (32-bit)
- halfword (16-bit)
- byte (8-bit).

You must align these as follows:
- word quantities must be aligned to four-byte boundaries
- halfword quantities must be aligned to two-byte boundaries
- byte quantities can be placed on any byte boundary.

2.6   Operating modes

In all states there are seven modes of operation:

- User mode is the usual ARM program execution state, and is used for executing most application programs
- *Fast interrupt* (FIQ) mode is used for handling fast interrupts
- *Interrupt* (IRQ) mode is used for general-purpose interrupt handling
- Supervisor mode is a protected mode for the operating system
- Abort mode is entered after a data or instruction Prefetch Abort
- System mode is a privileged user mode for the operating system
- Undefined mode is entered when an undefined instruction exception occurs.

Modes other than User mode are collectively known as privileged modes. Privileged modes are used to service interrupts or exceptions, or to access protected resources.

Each register has an operating mode as shown in Table 2-1.

Table 2-1 Register mode identifiers

Mode	Mode identifier
User	usr[a]
Fast interrupt	fiq
Interrupt	irq
Supervisor	svc
Abort	abt
System	usr[a]
Undefined	und

a. The usr identifier is usually omitted from register names. It is only used in descriptions where the User or System mode register is specifically accessed from another operating mode.

2.7 Registers

The ARM7EJ-S processor has a total of 37 registers:

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

These registers are not all accessible at the same time. The processor state and operating mode determine which registers are available to the programmer.

2.7.1 The ARM state register set

In ARM state, 16 general registers and one or two status registers are accessible at any one time. In privileged modes, mode-specific banked registers become available. Figure 2-3 on page 2-11 shows which registers are available in each mode.

The ARM state register set contains 16 directly-accessible registers, r0 to r15. A further register, the *Current Program Status Register* (CPSR), contains condition code flags and the current mode bits. Registers r0 to r13 are general-purpose registers used to hold either data or address values. Registers r14, r15, and the CPSR have the following special functions:

Link register   Register r14 is used as the subroutine *Link Register* (LR).

Register r14 receives a copy of r15 when a *Branch with Link* (BL or BLX) instruction is executed.

You can treat r14 as a general-purpose register at all other times. 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 or BLX instructions are executed within interrupt or exception routines.

Program counter   Register r15 holds the PC. In:

- ARM state this is word-aligned
- Thumb state this is halfword-aligned
- Jazelle state this is byte-aligned.

In privileged modes, another register, the *Saved Program Status Register* (SPSR), is accessible. This contains the condition code flags and the mode bits saved as a result of the exception that caused entry to the current mode.

Banked registers are discrete physical registers in the core that are mapped to the available registers depending on the current processor operating mode. Banked register contents are preserved across operating mode changes.

Banked registers have a mode identifier that indicates which User mode register they are mapped to. These mode identifiers are shown in Table 2-1 on page 2-8.

FIQ mode has seven banked registers mapped to r8–r14 (r8_fiq–r14_fiq). As a result many FIQ handlers do not have to save any registers.

The Supervisor, Abort, IRQ, and Undefined modes each have alternative mode-specific registers mapped to r13 and r14, enabling a private stack pointer and link register for each mode.

Figure 2-3 on page 2-11 shows the ARM state registers.

 ARM DDI 0214B

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

Indicates that the normal register used by the User or System mode
has been replaced by an alternative register specific to the exception mode.

Figure 2-3 Register organization in ARM state

2.7.2   The Thumb state register set

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

*   eight general registers, r0–r7 (for details of high register access in Thumb state see *Accessing high registers in Thumb state* on page 2-13)
*   the PC
*   a stack pointer, SP (ARM r13)
*   an LR (ARM r14)
*   the CPSR.

There are banked SPs, LRs, and SPSRs for each privileged mode. This register set 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

2.7.3   Relationships between ARM state and Thumb state registers

The relationships between the ARM state and Thumb state registers are shown in Figure 2-5.

Thumb state		ARM state
r0	→	r0
r1	→	r1
r2	→	r2
r3	→	r3
r4	→	r4
r5	→	r5
r6	→	r6
r7	→	r7
		r8
		r9
		r10
		r11
		r12
Stack pointer (SP)	→	Stack pointer (r13)
Link register (LR)	→	Link register (r14)
Program counter (PC)	→	Program counter (r15)
CPSR	→	CPSR
SPSR	→	SPSR
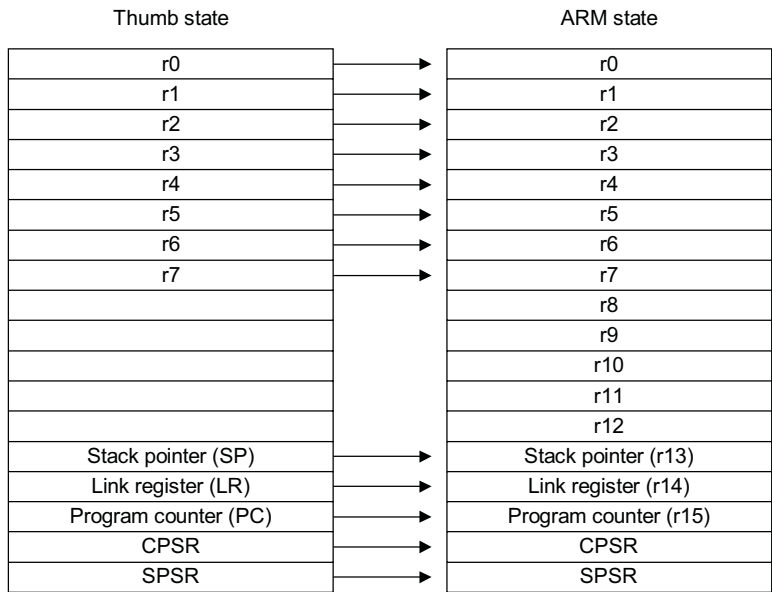
Figure 2-5 Relationships between the ARM state and Thumb state registers

——— **Note** ———

Registers r0–r7 are known as the low registers. Registers r8–r15 are known as the high registers.

2.7.4   Accessing high registers in Thumb state

In Thumb state, the high registers (r8–r15) are not part of the standard register set. With assembly language programming you have limited access to them, but can use them for fast temporary storage.

You can use special variants of the MOV instruction to transfer a value from a low register (in the range r0–r7) to a high register, and from a high register to a low register. The CMP instruction enables you to compare high register values with low register values. The ADD instruction enables you to add high register values to low register values. For more details, refer to the *ARM Architecture Reference Manual*.

2.8   The program status registers

The ARM7EJ-S processor contains one CPSR, and five SPSRs for exception handlers to use. The program status 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 in the status registers is shown in Figure 2-6, and described in:

*   *The condition code flags* on page 2-16
*   *The Q flag* on page 2-16
*   *The J bit* on page 2-16
*   *The control bits* on page 2-17
*   *Reserved bits* on page 2-19.

Condition code flags   **Reserved**   **Control bits**

31	30	29	28	27	26	25	24 23	8	7	6	5	4	0
N	Z	C	V	Q			J	Reserved	I	F	T		Mode	

Jazelle state bit

Reserved

Reserved

Sticky overflow

Overflow

Carry/Borrow/Extend

Zero

Negative/Less than

Mode bits

Thumb state bit

FIQ disable

IRQ disable
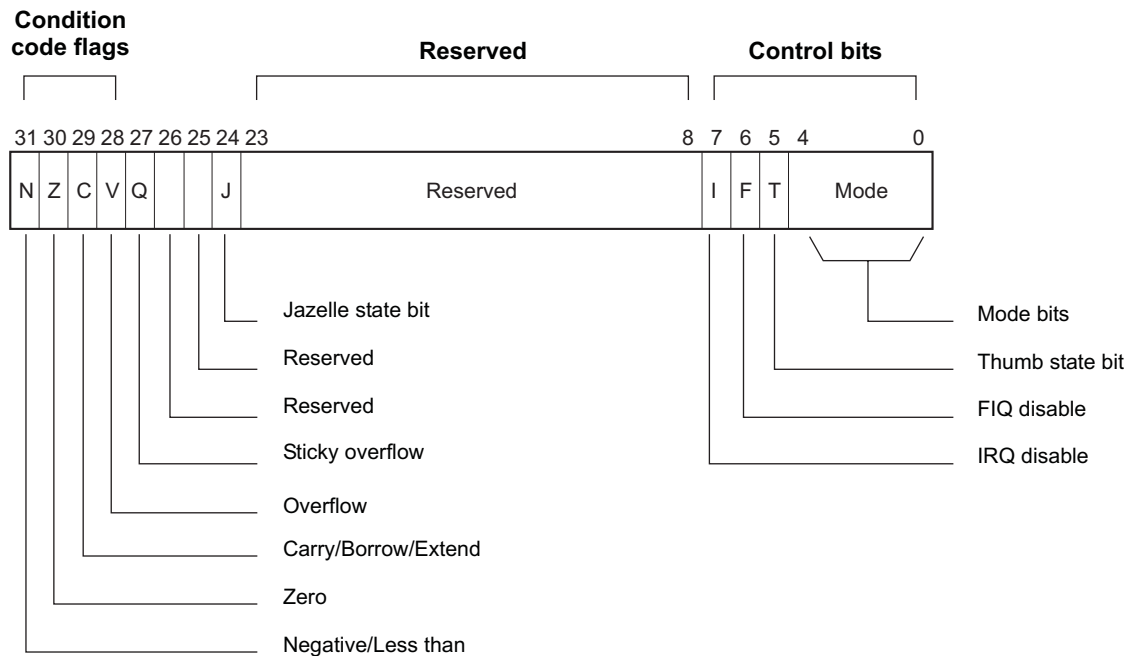
Figure 2-6 Program status register

───── **Note** ─────

The unused bits of the status registers might be used in future ARM architectures, and must not be modified by software. The unused bits of the status registers are readable, to enable the processor state to be preserved (for example, during process context switches) and writable, to enable the processor state to be restored. To maintain compatibility with future ARM processors, and as good practice, you are strongly advised to use a read-modify-write strategy when changing the CPSR.

2.8.1 The condition code flags

The N, Z, C, and V bits are the condition code flags. They can be set by arithmetic and logical operations, and also by MSR and LDM instructions. The ARM7EJ-S processor tests these flags to determine whether to execute an instruction.

All instructions can execute conditionally on the state of the N, Z, C, and V bits in ARM state. In Thumb state, only the Branch instruction can be executed conditionally. For more information about conditional execution, refer to the *ARM Architecture Reference Manual*.

2.8.2 The Q flag

The Sticky Overflow (Q) flag can be set by certain multiply and fractional arithmetic instructions:
- QADD
- QDADD
- QSUB
- QDSUB
- SMLAxy
- SMLAWy.

The Q flag is *sticky* in that, once set by an instruction, it remains set until explicitly cleared by an MSR instruction writing to CPSR. Instructions cannot execute conditionally on the status of the Q flag. To determine the status of the Q flag you must read the PSR into a register and extract the Q flag from this. For details of how the Q flag is set and cleared, see individual instruction definitions in the *ARM Architectural Reference Manual*.

2.8.3 The J bit

The J bit in the CPSR indicates when the ARM7EJ-S processor is in Jazelle state.

When:

J = 0   The processor is in ARM or Thumb state, depending on the T bit.

J = 1   The processor is in Jazelle state.

> ──── **Note** ────
>
> - The combination of J = 1 and T = 1 causes similar effects to setting T=1 on a non Thumb-aware processor. That is, the next instruction executed causes entry to the Undefined Instruction exception. Entry to the exception handler causes the processor to re-enter ARM state, and the handler can detect that this was the cause of the exception because J and T are both set in SPSR_und.
> - MSR cannot be used to change the J bit in the CPSR.
> - The placement of the J bit avoids any usage of the status or extension bytes in code run on ARMv5TE or earlier processors. This ensures that OS code written using the older CPSR, SPSR, CPSR_all, or SPSR_all syntax for the destination of an MSR instruction continues to work.

2.8.4   The control bits

The bottom eight bits of a PSR are known collectively as the *control bits*. They are the:
- *Interrupt disable bits*
- *T bit*
- *Mode bits* on page 2-18.

The control bits change when an exception occurs. When the processor is operating in a privileged mode, software can manipulate these bits.

Interrupt disable bits

The I and F bits are the interrupt disable bits:
- when the I bit is set, IRQ interrupts are disabled
- when the F bit is set, FIQ interrupts are disabled.

T bit

> ──── **Caution** ────
>
> Never use an MSR instruction to force a change to the state of the T bit in the CPSR. If you do this, the processor enters an unpredictable state.

The T bit reflects the operating state:

- when the T bit is set, the processor is executing in Thumb state
- when the T bit is clear, the processor is executing in ARM state, or Jazelle state depending on the J bit.

The operating state is reflected by the **CPTBIT** and **CPJBIT** external delayed signals.

Mode bits

M[4:0] are the mode bits.

——— **Caution** ———

An illegal value programmed into M[4:0] causes the processor to enter an unrecoverable state. If this occurs, apply reset. Not all combinations of the mode bits define a valid processor mode, so take care to use only those bit combinations shown.

These bits determine the processor operating mode as shown in Table 2-2.

Table 2-2 PSR mode bit values

M[4:0]	Mode	Visible state registers	
		Thumb	**ARM**
b10000	User	r0–r7, r8-r12[a], SP, LR, PC, CPSR	r0–r14, PC, CPSR
b10001	FIQ	r0–r7, r8_fiq-r12_fiq[a], SP_fiq, LR_fiq PC, CPSR, SPSR_fiq	r0–r7, r8_fiq–r14_fiq, PC, CPSR, SPSR_fiq
b10010	IRQ	r0–r7, r8-r12[a], SP_irq, LR_irq, PC, CPSR, SPSR_irq	r0–r12, r13_irq, r14_irq, PC, CPSR, SPSR_irq
b10011	Supervisor	r0–r7, r8-r12[a], SP_svc, LR_svc, PC, CPSR, SPSR_svc	r0–r12, r13_svc, r14_svc, PC, CPSR, SPSR_svc

Table 2-2 PSR mode bit values (continued)

M[4:0]	Mode	Visible state registers	
		Thumb	**ARM**
b10111	Abort	r0–r7, r8-r12[a], SP_abt, LR_abt, PC, CPSR, SPSR_abt	r0–r12, r13_abt, r14_abt, PC, CPSR, SPSR_abt
b11011	Undefined	r0–r7, r8-r12[a], SP_und, LR_und, PC, CPSR, SPSR_und	r0–r12, r13_und, r14_und, PC, CPSR, SPSR_und
b11111	System	r0–r7, r8-r12[a], SP, LR, PC, CPSR	r0–r14, PC, CPSR

a. Access to these registers is limited in Thumb state.

2.8.5   Reserved bits

The remaining bits in the PSRs are unused, but are reserved. When changing a PSR flag or control bits, make sure that these reserved bits are not altered. You must ensure that your program does not rely on reserved bits containing specific values because future processors might use some or all of the reserved bits.

2.9 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 attempting to handle an exception, the ARM7EJ-S processor preserves the current processor state so that the original program can resume when the handler routine has finished.

If two or more exceptions arise simultaneously, the exceptions are dealt with in the fixed order given in *Exception priorities* on page 2-27.

This section provides details of the ARM7EJ-S processor exception handling:

- *Exception entry and exit summary*
- *Entering an ARM exception* on page 2-21
- *Leaving an ARM exception* on page 2-22.

2.9.1 Exception entry and exit summary

Table 2-3 shows 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 or entry	Return instruction	Previous state			Notes
		ARM r14_x	Thumb r14_x	Jazelle r14_x	
SWI	`MOVS PC, R14_svc`	PC + 4	PC+2	-	Where the PC is the address of the SWI or undefined instruction. Not used in Jazelle state.
UNDEF	`MOVS PC, R14_und`	PC + 4	PC+2	-	
PABT	`SUBS PC, R14_abt, #4`	PC + 4	PC+4	PC+4	Where the PC is the address of the instruction that had the Prefetch Abort.
FIQ	`SUBS PC, R14_fiq, #4`	PC + 4	PC+4	PC+4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority.
IRQ	`SUBS PC, R14_irq, #4`	PC + 4	PC+4	PC+4	

Table 2-3 Exception entry and exit (continued)

Exception or entry	Return instruction	Previous state			Notes
		ARM r14_x	**Thumb r14_x**	**Jazelle r14_x**	
DABT	`SUBS PC, R14_abt, #8`	PC + 8	PC+8	PC+8	Where the PC is the address of the Load or Store instruction that generated the Data Abort.
RESET	`NA`	-	-	-	The value saved in r14_svc upon reset is `UNPREDICTABLE`.
BKPT	`SUBS PC, R14_abt, #4`	PC + 4	PC+4	PC+4	Software breakpoint.

2.9.2 Entering an ARM exception

When handling an ARM exception the ARM7EJ-S processor:

1. Preserves the address of the next instruction in the appropriate LR. When the exception entry is from:
    *   ARM and Jazelle states, the processor copies the address of the next instruction into the LR (current PC + 4 or PC + 8 depending on the exception)
    *   Thumb state, the processor writes the value of the PC into the LR, offset by a value (current PC + 4 or PC + 8 depending on the exception) that causes the program to resume from the correct place on return.

    The exception handler does not have to determine the state when entering an exception. For example, in the case of a SWI, `MOVS PC, r14_svc` always returns to the next instruction regardless of whether the SWI was executed in ARM or Thumb state.

2. Copies the CPSR into the appropriate SPSR.

3. Forces the CPSR mode bits to a value which depends on the exception.

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

The ARM7EJ-S processor can also set the interrupt disable flags to prevent otherwise unmanageable nesting of exceptions.

—— **Note** ——

Exceptions are always entered, handled, and exited in ARM state. When the processor is in Thumb state or Jazelle state and an exception occurs, the switch to ARM state takes place automatically when the exception vector address is loaded into the PC.

—————————

2.9.3   Leaving an ARM exception

When an exception has completed, the exception handler must move the LR, minus an offset to the PC. The offset varies according to the type of exception, as shown in Table 2-3 on page 2-20.

If the S bit is set and rd is r15, the core copies the SPSR back to the CPSR and clears the interrupt disable flags that were set on entry.

—— **Note** ——

The action of restoring the CPSR from the SPSR automatically resets the T bit and J bit to the values held immediately prior to the exception. The I and F bits are automatically restored to the value they held immediately prior to the exception.

—————————

2.9.4   Reset

When the **nRESET** signal is driven LOW a reset occurs, and the processor abandons the executing instruction.

When **nRESET** is driven HIGH again the processor:

1.   Forces **CPSR[4:0]** to b10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR T bit and J bit. Other bits in the CPSR are indeterminate.

2.   Forces the PC to fetch the next instruction from the reset vector address.

3.   Reverts to ARM state, and resumes execution.

After reset, all register values except the PC and CPSR are indeterminate.

Refer to Chapter 8 *Device Reset* for more details of the reset behavior.

Copyright © 2001 ARM Limited. All rights reserved.

2.9.5   Fast interrupt request

The *Fast Interrupt Request* (FIQ) exception supports fast interrupts. In ARM state, FIQ mode has eight banked registers to reduce, or even remove the requirement for storing the previous state of the registers on entry to the interrupt routine (minimizing the overhead of context switching).

An FIQ is externally generated by taking the **nFIQ** signal input LOW.

Irrespective of whether exception entry is from ARM state, Thumb state, or Jazelle state, an FIQ handler returns from the interrupt by executing:

```
SUBS PC,R14_fiq,#4
```

You can disable FIQ exceptions within a privileged mode by setting the CPSR F flag. When the F flag is clear, the processor checks for a LOW level on the **nFIQ** signal at the end of each instruction.

FIQs and IRQs are disabled automatically when an FIQ occurs. Nested interrupts are allowed but it is up to the programmer to save any corruptible registers and re-enable FIQs and IRQs explicitly.

2.9.6   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 on entry to an FIQ sequence. You can disable IRQ at any time, by setting the I bit in the CPSR from a privileged mode.

Irrespective of whether exception entry is from ARM state, Thumb state, or Jazelle state, an IRQ handler returns from the interrupt by executing:

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

You can disable IRQ exceptions within a privileged mode by setting the CPSR I flag. When the I flag is clear, the processor checks for a LOW level on the **nIRQ** signal at the end of each instruction.

IRQs are disabled automatically when an IRQ occurs. Nested interrupts are allowed but it is up to you to save any corruptible registers and to re-enable IRQs.

2.9.7   Aborts

An abort indicates that the current memory access cannot be completed. An abort is signaled by the external abort input pin, **ABORT**.

There are two types of abort:
*   *Prefetch Abort* on page 2-24

•   *Data Abort*.

IRQs are disabled automatically when an abort occurs.

Prefetch Abort

This is signaled by an assertion on the **ABORT** input pin and checked at the end of each instruction fetch.

When a Prefetch Abort occurs, the processor marks the prefetched instruction as invalid, but does not take the exception until the instruction reaches the Execute stage 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.

After dealing with the cause of the abort, the handler must execute the following instruction irrespective of the processor operating state:

```
SUBS PC,R14_abt,#4
```

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

Data Abort

This is signaled by an assertion on the **ABORT** input pin and checked at the end of each data access.

The ARM7EJ-S processor implements the *base restored Data Abort model*, which differs from the *base updated Data Abort model* implemented by the ARM7TDMI-S processor.

The difference in the Data Abort model affects only a very small section of operating system code, in the Data Abort handler. It does not affect user code.

With the *base restored Data Abort model*, when a Data Abort exception occurs during the execution of a memory access instruction, the base register is always restored by the processor hardware to the value it contained *before* the instruction was executed. This removes the requirement for the Data Abort handler to *unwind* any base register update that might have been specified by the aborted instruction. This greatly simplifies the software Data Abort handler.

The abort mechanism enables you to use a demand-paged virtual memory system. In such a system, the processor is allowed to generate arbitrary addresses. When the data at an address is unavailable, the *Memory Management Unit* (MMU) signals an abort. The abort handler must then work out the cause of the abort, make the requested data

available, and retry the aborted instruction. The application program requires no knowledge of the amount of memory available to it, and its state is not affected by the abort.

After dealing with the cause of the abort, the handler must execute the following return instruction irrespective of the processor operating state at the point of entry:

```
SUBS PC,R14_abt,#8
```

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

2.9.8   Software interrupt instruction

You can use the *Software Interrupt Instruction* (SWI) to enter Supervisor mode, usually to request a particular supervisor function. An SWI handler returns by executing the following instruction, irrespective of the processor operating state:

```
MOVS PC, R14_svc
```

This action restores the PC and CPSR, and returns to the instruction following the SWI. The SWI handler reads the opcode to extract the SWI function number.

IRQs are automatically disabled when a software interrupt occurs and are automatically re-enabled on return.

2.9.9   Undefined instruction

When an instruction is encountered that neither the processor, nor any coprocessor in the system can handle, the ARM7EJ-S processor takes the undefined instruction trap. Software can use this mechanism to extend the ARM instruction set by emulating undefined coprocessor instructions.

After emulating the failed instruction, the trap handler must execute the following instruction, irrespective of the processor operating state:

```
MOVS PC,R14_und
```

This action restores the CPSR and returns to the next instruction after the undefined instruction.

IRQs are automatically disabled when an undefined instruction trap occurs and are automatically re-enabled on return. For more information about undefined instructions, refer to the *ARM Architecture Reference Manual*.

2.9.10 Breakpoint instruction (BKPT)

A breakpoint (BKPT) instruction operates as though the instruction caused a Prefetch Abort.

A breakpoint instruction does not cause the ARM7EJ-S processor to take the Prefetch Abort exception until the instruction reaches the Execute stage of the pipeline. If the instruction is not executed, for example because a branch occurs while it is in the pipeline, the breakpoint does not take place.

After dealing with the breakpoint, the handler must execute the following instruction irrespective of the processor operating state:

```
SUBS PC,R14_abt,#4
```

This action restores both the PC and the CPSR, and retries the breakpointed instruction.

— **Note** —

If the EmbeddedICE-RT logic is in halt mode, a breakpoint instruction causes the processor to enter debug state. See *Debug control register* on page B-31.

2.9.11 Exception vectors

You can configure the location of the exception vector addresses using the input **CFGHIVECS**, as shown in Table 2-4.

Table 2-4 Configuration of exception vector base address locations

Value of CFGHIVECS	Exception vector base location
0	0x0000 0000
1	0xFFFF 0000

Table 2-5 shows the exception vector addresses and entry conditions for the different exception types.

Table 2-5 Exception vectors

Exception	Offset from vector base	Mode on entry	I bit on entry	F bit on entry
Reset	0x00	Supervisor	Disabled	Disabled
Undefined instruction	0x04	Undefined	Disabled	Unchanged
Software interrupt	0x08	Supervisor	Disabled	Unchanged

Table 2-5 Exception vectors (continued)

Exception	Offset from vector base	Mode on entry	I bit on entry	F bit on entry
Abort (prefetch)	`0x0C`	Abort	Disabled	Unchanged
Abort (data)	`0x10`	Abort	Disabled	Unchanged
Reserved	`0x14`	Reserved	-	-
IRQ	`0x18`	IRQ	Disabled	Unchanged
FIQ	`0x1C`	FIQ	Disabled	Disabled

2.9.12   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. BKPT, undefined instruction, and SWI (lowest priority).

Some exceptions cannot occur together:

* The BKPT, or undefined instruction, and SWI exceptions are mutually exclusive. Each corresponds to a particular (non-overlapping) decoding of the current instruction.

* When FIQs are enabled, and a Data Abort occurs at the same time as an FIQ, the processor enters the Data Abort handler, and proceeds immediately to the FIQ vector.

  A normal return from the FIQ causes the Data Abort handler to resume execution.

  Data Aborts must have higher priority than FIQs to ensure that the transfer error does not escape detection. You must add the time for this exception entry to the worst-case FIQ latency calculations in a system that uses aborts to support virtual memory.

  The FIQ handler must not access any memory that can generate a Data Abort, because the initial Data Abort exception condition will be lost.

ARM DDI 0214B

Chapter 3
Memory Interface

This chapter describes the ARM7EJ-S processor memory interface. It contains the following sections:

- *About the memory interface* on page 3-2
- *Bus interface signals* on page 3-3
- *Bus cycle types* on page 3-4
- *Addressing signals* on page 3-9
- *Data timed signals* on page 3-11
- *Using CLKEN to control bus cycles* on page 3-15.

3.1   About the memory interface

The ARM7EJ-S processor has a von Neumann architecture, with a single 32-bit data bus carrying both instructions and data. Only load, store, and swap instructions can access data from memory.

The ARM7EJ-S processor supports four basic types of memory cycle:

- nonsequential
- sequential
- internal
- coprocessor register transfer.

                                     ARM DDI 0214B

3.2   Bus interface signals

The signals in the ARM7EJ-S processor bus interface can be grouped into four categories:

• clocking and clock control
• address class signals
• memory request signals
• data timed signals.

The clocking and clock control signals are:

• **CLK**
• **CLKEN**
• **nRESET**.

The address class signals are:

• **ADDR[31:0]**
• **WRITE**
• **SIZE[1:0]**
• **PROT[1:0]**
• **LOCK**.

The memory request signals are:

• **TRANS[1:0]**.

The data timed signals are:

• **WDATA[31:0]**
• **RDATA[31:0]**
• **ABORT**.

Each of these signal groups shares a common timing relationship to the bus interface cycle. All signals in the ARM7EJ-S processor bus interface are generated on or sampled at the rising edge of **CLK**.

Bus cycles can be extended using the **CLKEN** signal (see *Using CLKEN to control bus cycles* on page 3-15). All other sections of this chapter describe a simple system in which **CLKEN** is permanently HIGH.

3.3   Bus cycle types

The ARM7EJ-S processor bus interface is pipelined, and 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.
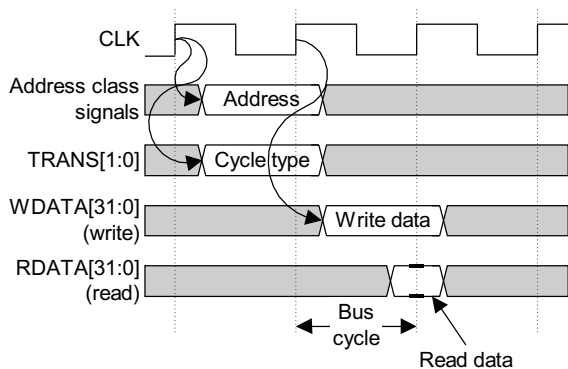
Figure 3-1 Simple memory cycle

The ARM7EJ-S processor bus interface can perform four different types of memory cycle. These are indicated by the state of the **TRANS[1:0]** signals. Memory cycle types are encoded on the **TRANS[1:0]** signals as shown in Table 3-1.

Table 3-1 Cycle types

TRANS[1:0]	Cycle type	Description
b00	I cycle	Internal cycle.
b01	C cycle	Coprocessor 15 MCR operation (see *MCR operation* on page 5-7).
b10	N cycle	Nonsequential cycle.
b11	S cycle	Sequential cycle.

A memory controller for the ARM7EJ-S processor commits to a memory access only on an N cycle or an S cycle.

The ARM7EJ-S processor has three basic types of bus cycle:

Nonsequential cycle

During this cycle, the ARM7EJ-S processor requests a transfer to, or from an address that is unrelated to the address used in the preceding cycle.

Sequential cycle   During this cycle, the ARM7EJ-S processor requests a transfer to or from an address that is either one word or one halfword greater than the address used in the preceding cycle.

Internal cycle   During this cycle, the ARM7EJ-S processor does not require a transfer because it is performing an internal function and no useful prefetching can be performed at the same time.

3.3.1   Nonsequential cycles

A nonsequential cycle is the simplest form of bus cycle, and occurs when the ARM7EJ-S processor requests a transfer to, or from, an address that is unrelated to the address used in the preceding cycle. The memory controller must initiate a memory access to satisfy this request.

The address class signals and **TRANS[1:0]** is N cycle are broadcast on the bus. At the end of the next bus cycle the data is transferred between the processor and the memory. This is illustrated in Figure 3-2.
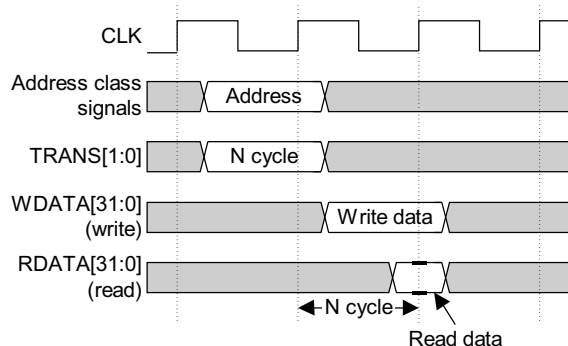
Figure 3-2 Nonsequential memory cycle

The processor can perform back-to-back nonsequential memory cycles. This happens, for example, when an STR instruction is executed, as shown in Figure 3-3 on page 3-6. If you are designing a memory controller for the processor, and your memory system is

---

unable to cope with this case, you must use the **CLKEN** signal to extend the bus cycle to allow sufficient cycles for the memory system. For more information, see *Using CLKEN to control bus cycles* on page 3-15.
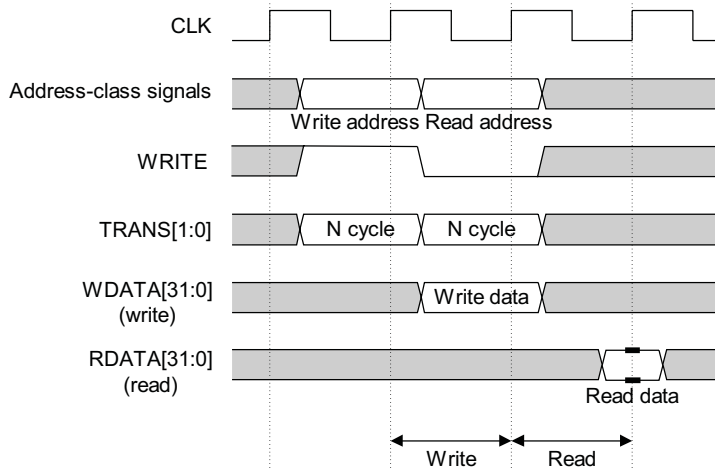
Figure 3-3 Back to back memory cycles

3.3.2 Sequential cycles

Sequential cycles perform burst transfers on the bus. You can use this information to optimize the design of a memory controller interfacing to a burst memory device, such as a DRAM.

During a sequential cycle, the processor requests a memory location that is part of a sequential burst. If this is the first cycle in the burst, the address can be the same as the previous internal cycle. Otherwise the address is incremented from the previous cycle:

- for a burst of word accesses, the address is incremented by 4 bytes
- for a burst of halfword accesses, the address is incremented by 2 bytes.

Bursts of byte accesses are not possible.

A burst always starts with an N cycle or a merged I-S cycle (see *Simple memory cycle* on page 3-4), and continues with S cycles. A burst comprises transfers of the same type. The **ADDR[31:0]** signal increments during the burst. The other address class signals remain the same throughout the burst.

The types of bursts are shown in Table 3-2.

Table 3-2 Burst types

Burst type	Address increment	Cause
Word read	4 bytes	Code fetches, or LDM instruction
Word write	4 bytes	STM instruction
Halfword read	2 bytes	Thumb code fetches

All accesses in a burst are of the same width, direction, and protection type. For more details, see *Addressing signals* on page 3-9.

An example of a burst access is shown in Figure 3-4.
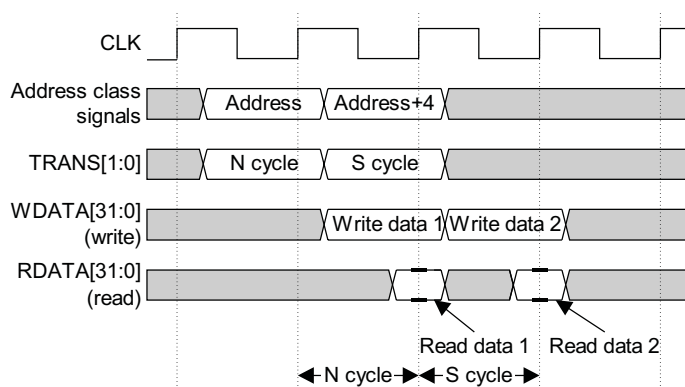
Figure 3-4 Sequential access cycles

3.3.3   Internal cycles

During an internal cycle, the processor does not require a memory access, because an internal function is being performed and no useful prefetching can be performed at the same time.

Where possible the processor broadcasts the address for the next access, so that decode can start, but the memory controller must not commit to a memory access. This is described in *Sequential access cycles*.

3.3.4 Merged I-S cycles

Where possible, the processor performs an optimization on the bus to allow extra time for memory decode. When this happens, the address of the next memory cycle is broadcast during an internal cycle on this bus. This allows the memory controller to decode the address, but it must not initiate a memory access during this cycle. In a merged I-S cycle, the next cycle is a sequential cycle to the same memory location. This commits to the access, and the memory controller must initiate the memory access. This is shown in Figure 3-5.
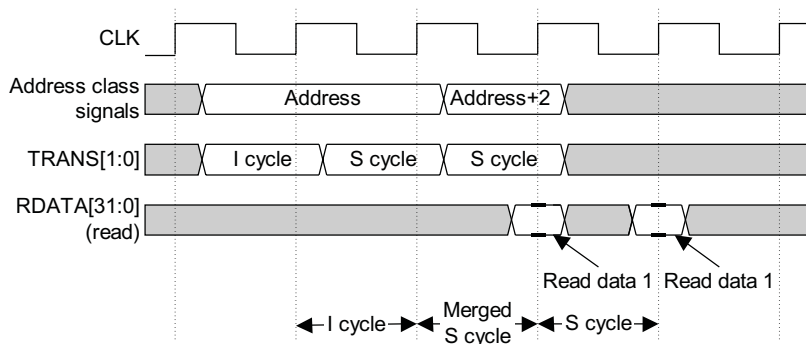
Figure 3-5 Merged I-S cycle

——— **Note** ———

When designing a memory controller, make sure that the design also works when an I cycle is followed by an N cycle to a different address. This sequence might occur during exceptions, or during writes to the PC. It is essential that the memory controller does not commit to the memory cycle during an I cycle.

3.4   Addressing signals

The address class signals are:

*   *ADDR[31:0]*
*   *WRITE*
*   *SIZE[1:0]*
*   *PROT[1:0]* on page 3-10
*   *LOCK* on page 3-10.

3.4.1   ADDR[31:0]

ADDR[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. When a word access is signalled, the memory system must ignore the bottom two bits, **ADDR[1:0]**, and when a halfword access is signalled the memory system must ignore the bottom bit, **ADDR[0]**.

3.4.2   WRITE

WRITE indicates a processor write cycle when HIGH, and a processor read cycle when LOW. 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.

3.4.3   SIZE[1:0]

The **SIZE[1:0]** bus encodes the size of the transfer. The processor can transfer word, halfword, and byte quantities. This is encoded on **SIZE[1:0]** as shown in Table 3-3.

Table 3-3 Transfer widths

SIZE[1:0]	Transfer width
b00	Byte
b01	Halfword
b10	Word
b11	Reserved

The size of transfer does not change during a burst of S cycles.

─── **Note** ───

A writable memory system for the ARM7EJ-S processor must have individual byte write enables. Both the C Compiler and the ARM debug tool chain (for example, Multi-ICE) assume that arbitrary bytes in the memory can be written. If individual byte write capability is not provided, it might not be possible to use either of these tools.

3.4.4   PROT[1:0]

The **PROT[1:0]** bus encodes information about the transfer. A memory management unit uses this signal to determine whether an access is from a privileged mode, and whether it is an opcode or a data fetch. This can therefore be used to implement an access permission scheme. The encoding of **PROT[1:0]** is shown in Table 3-4.

Table 3-4 PROT[1:0] encoding

PROT[1:0]	Mode	Opcode or data
b00	User	Opcode
b01	User	Data
b10	Privileged	Opcode
b11	Privileged	Data

3.4.5   LOCK

LOCK indicates to an arbiter that an atomic operation is being performed on the bus. **LOCK** is normally LOW, but is set HIGH to indicate that a SWP or SWPB instruction is being performed. These instructions perform an atomic read or write operation and can be used to implement semaphores.

3.5   Data timed signals

The data timed signals are:

- *RDATA[31:0]*
- *ABORT*
- *WDATA[31:0].*

3.5.1   RDATA[31:0]

RDATA[31:0] is the read data bus, and is used by the ARM7EJ-S processor to fetch both opcodes and data. The **RDATA[31:0]** signal is sampled on the rising edge of **CLK** at the end of the bus cycle.

3.5.2   ABORT

ABORT indicates that a memory transaction failed to complete successfully. **ABORT** is sampled at the end of the bus cycle during active memory cycles (S cycles and N cycles).

If **ABORT** is asserted on a data access, it causes the ARM7EJ-S processor to take the Data Abort trap. If it is asserted on an opcode fetch, the abort is tracked down the pipeline, and the Prefetch Abort trap is taken if the instruction is executed.

ABORT can be used by a memory management system to implement, for example, a basic memory protection scheme or a demand-paged virtual memory system.

For more details about aborts, see *Aborts* on page 2-23.

3.5.3   WDATA[31:0]

WDATA[31:0] is the write data output bus, used to transfer data from the processor to the memory system.

3.6 Byte and halfword accesses

The ARM7EJ-S processor indicates the size of a transfer using the **SIZE[1:0]** signals. These are encoded as shown in Table 3-3 on page 3-9.

All writable memory in an ARM7EJ-S processor-based system supports the writing of individual bytes to enable the use of the C Compiler and the ARM debug tool chain (for example, Multi-ICE).

The address produced by the ARM7EJ-S processor is always a byte address. However, the memory system ignores the insignificant bits of the address. The significant address bits are shown in Table 3-5.

Table 3-5 Significant address bits

SIZE[1:0]	Width	Significant address bits
b00	Byte	**ADDR[31:0]**
b01	Halfword	**ADDR[31:1]**
b10	Word	**ADDR[31:2]**

When a halfword or byte read is performed, a 32-bit memory system can return the complete 32-bit word, and the ARM7EJ-S processor extracts the valid halfword or byte field from it. The fields extracted depend on the state of the **CFGBIGEND** signal, which determines the endianness of the system (see *Memory formats* on page 2-4).

The fields extracted by the ARM7EJ-S processor are shown in Table 3-6.

Table 3-6 Word accesses

SIZE[1:0]	ADDR[1:0]	Little-endian CFGBIGEND=0	Big-endian CFGBIGEND=1
b10	bXX	RDATA[31:0]	RDATA[31:0]

 ARM DDI 0214B

When connecting 8-bit to 16-bit memory systems to the ARM7EJ-S processor, make sure that the data is presented to the correct byte lanes on the processor as shown in Table 3-7 and Table 3-8.

Table 3-7 Halfword accesses

SIZE[1:0]	ADDR[1:0]	Little-endian CFGBIGEND=0	Big-endian CFGBIGEND=1
b01	b0X	RDATA[15:0]	RDATA[31:16]
b01	b1X	RDATA[31:16]	RDATA[15:0]

Table 3-8 Byte accesses

SIZE[1:0]	ADDR[1:0]	Little-endian	Big-endian
b00	b00	RDATA[7:0]	RDATA[31:24]
b00	b01	RDATA[15:8]	RDATA[23:16]
b00	b10	RDATA[23:16]	RDATA[15:8]
b00	b11	RDATA[31:24]	RDATA[7:0]

3.6.1 Writes

When the processor performs a byte or halfword write, the data being written is replicated across the bus, as illustrated in Figure 3-6 on page 3-14. The memory system can use the most convenient copy of the data. A writable memory system must be capable of performing a write to any single byte in the memory system. This capability is required by the ARM C Compiler and the Debug tool chain.
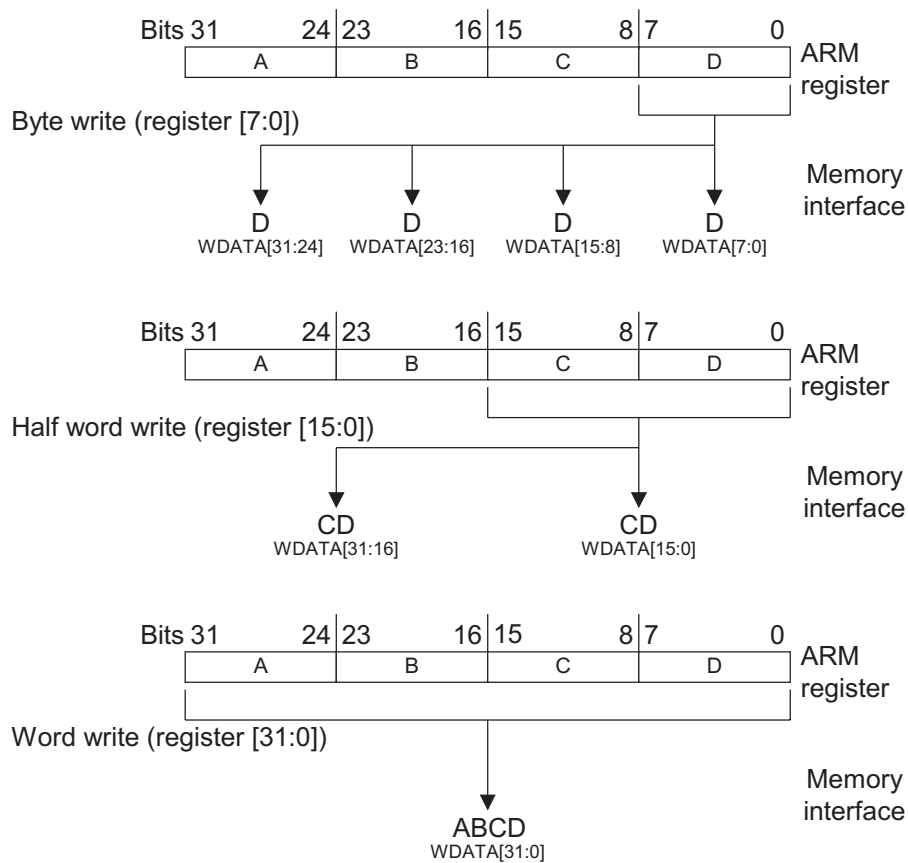
Figure 3-6 Data replication

3.7 Using CLKEN to control bus cycles

The pipelined nature of the ARM7EJ-S processor bus interface means that there is a distinction between *clock* cycles and *bus* cycles. **CLKEN** can be used to stretch a *bus* cycle, so that it lasts for many *clock* cycles. The **CLKEN** input extends the timing of bus cycles in increments of complete **CLK** cycles:

- when **CLKEN** is HIGH on the rising edge of **CLK**, a bus cycle completes
- when **CLKEN** is sampled LOW, the bus cycle is extended.

In the pipeline, the address class signals and the memory request signals are ahead of the data transfer by one *bus* cycle. In a system using **CLKEN** this can be more than one **CLK** cycle. This is illustrated in Figure 3-7, which shows **CLKEN** being used to extend a nonsequential cycle. In the example, the first N cycle is followed by another N cycle to an unrelated address, and the address for the second access is broadcast before the first access completes.
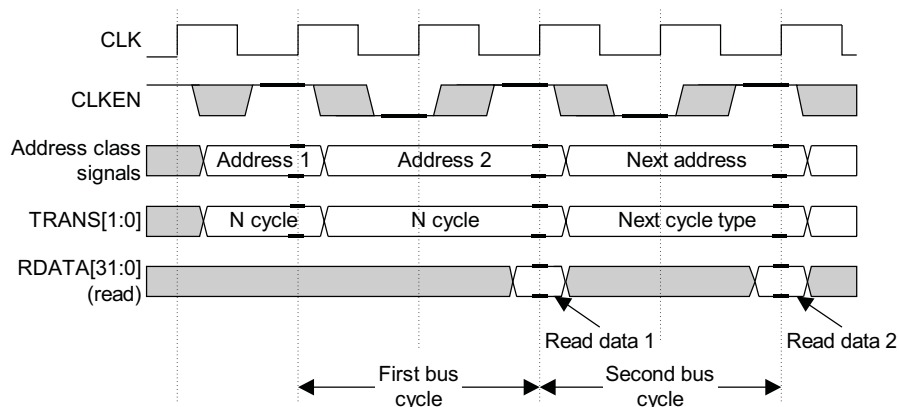
Figure 3-7 Use of CLKEN

——— **Note** ———

When designing a memory controller, you are strongly advised to sample the values of **TRANS[1:0]** and the address class signals only when **CLKEN** is HIGH. This ensures that the state of the memory controller is not accidentally updated during a bus cycle.

Chapter 4
Interrupts

This chapter describes the ARM7EJ-S processor interrupt behavior. It contains the following sections:

- *About interrupts* on page 4-2
- *Hardware interface* on page 4-3
- *Maximum interrupt latency* on page 4-5
- *Minimum interrupt latency* on page 4-6.

4.1 About interrupts

The ARM7EJ-S processor provides a two-level, fixed-priority hardware interrupt scheme.

The *Fast Interrupt Request* (FIQ) exception provides support for fast interrupts. The *Interrupt Request* (IRQ) exception provides support for normal priority interrupts. Refer to *Exceptions* on page 2-20 for more details about the programmer's model for interrupts and to Chapter 10 *AC Parameters* for details on interrupt signal timing.This chapter describes:

- issues concerning the hardware interface to the processor interrupt mechanism that a system designer must be aware of when integrating an ARM7EJ-S processor system

- issues that a programmer must be aware of when writing interrupt handler routines

- the worst case and best case interrupt latency.

4.2   Hardware interface

The hardware interrupts are described in the following sections:

- *Generating an interrupt*
- *Synchronization*
- *Re-enabling interrupts after an interrupt exception*.

4.2.1   Generating an interrupt

You can make the processor take the FIQ or IRQ exceptions (provided that interrupts
are enabled) by asserting the **nFIQ** or **nIRQ** inputs, respectively.It is essential that, once
asserted, the interrupt input remains asserted until the processor acknowledges to the
source of the interrupt that the interrupt has been taken. This acknowledgement usually
occurs when the interrupt service routine accesses the peripheral causing the interrupt,
for example:

- by reading an interrupt status register in the system interrupt controller
- by writing to a clear interrupt control bit
- by writing data to, or by reading data from, the interrupting peripheral.

4.2.2   Synchronization

The **nFIQ** and **nIRQ** inputs are synchronous inputs to the processor, and must be setup
and held about the rising edge of the processor clock, **CLK**. If interrupt events that are
asynchronous to **CLK** are present in a system, external synchronization logic is
required.

4.2.3   Re-enabling interrupts after an interrupt exception

You must take care when re-enabling interrupts (for example at the end of an interrupt
routine or with a re-entrant interrupt handler). You must ensure that the original source
of the interrupt has been removed before interrupts are enabled again on the processor.
If you cannot guarantee this, the processor might retake the interrupt exception
prematurely.When considering the timing relation of removing the source of interrupt
and re-enabling interrupts on the processor, you must take into account the pipelined
nature of the processor and the memory system to which it is connected. For example,
the instruction that clears the interrupt request (that is, deassertion of **nFIQ** or **nIRQ**)
typically does not take effect until after the instruction reaches the Memory pipeline
stage. The instruction that re-enables interrupts on the processor can cause the processor
to be sensitive to interrupts as early as when the instruction reaches the Execute pipeline
stage.For example, consider the following instruction sequence:

```
STR r0, [r1] ;Write to interrupt controller, clearing interruptSUBS pc, r14, #4
;Return from interrupt routine
```

The execution of this code sequence is illustrated in Figure 4-1.

| Cycle | 1 | 2 | 3 | 4 | 5 |

CLK

nFIQ

FIQDIS

| STR r0, [r1] | Decode | Execute | Memory | Write | |
| SUBS pc, r14, #4 | | Decode | Execute | Memory | Write |
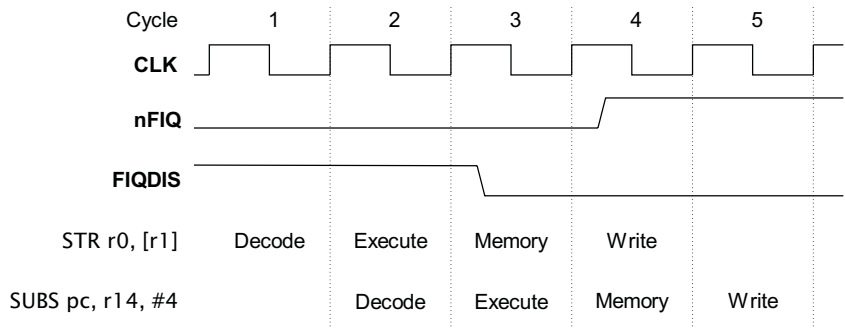
Figure 4-1 Retaking the FIQ exception

In Figure 4-1, the STR to the interrupt controller does not cause the deassertion of the **nFIQ** input until cycle 4. The SUBS instruction causes the processor to be sensitive to interrupts during cycle 3. Because of this timing relationship, the processor retakes the FIQ exception in this example.

The **FIQDIS** (and similarly **IRQDIS**) output from the processor indicates when the processor is sensitive to the state of **nFIQ** (**nIRQ**) (LOW for sensitive, HIGH for insensitive). If **nFIQ** is asserted in the same cycle that **FIQDIS** is LOW, the processor takes the FIQ exception in a later cycle, even if the **nFIQ** input is subsequently deasserted.

There are several approaches that you can adopt to ensure that interrupts are not enabled too early on the processor. The best approach to use is highly dependent on the overall system, and can be a combination of hardware and software. Example approaches are to:

*   analyze the system and ensure that enough instructions separate the instruction that removes the interrupt and the instruction that re-enables interrupts on the processor

*   use polling to read back a status bit from the system interrupt controller until it indicates that the interrupt has been removed before re-enabling interrupts.

4.3   Maximum interrupt latency

The processor samples the interrupt input pins on the rising-edge of the system clock, **CLK**, with **CLKEN** HIGH. The sampled signal is examined and can cause an interrupt in the following cases:

•   When a new instruction is scheduled to enter the Execute stage of the pipeline.

•   When a new instruction is in the Execute stage for the first *cycle* of its execution. Here *cycle* refers to **CLK** cycles with **CLKEN** HIGH.

•   When a coprocessor instruction is being busy-waited in the Execute stage.

•   When a new instruction that interlocked in the Execute stage has just progressed to its first active Execute cycle.

If the sampled signal is asserted at the same time as a multi-cycle instruction has started its second or subsequent cycle of execution, the interrupt exception entry does not start until the instruction has completed. The worst-case interrupt latency occurs when the longest possible LDM instruction incurs a Data Abort. The processor must enter the Data Abort mode before taking the interrupt so that the interrupt exception exit can occur correctly. This causes a worst-case latency of 24 cycles:

•   The longest LDM instruction is one that loads all of the registers, including the PC. Counting the first Execute cycle as 1, the LDM takes 16 cycles.

•   The last word to be transferred by the LDM is transferred in cycle 17, and the abort status for the transfer is returned in this cycle.

•   If a Data Abort happens, the processor detects this in cycle 18 and prepares for the Data Abort exception entry in cycle 19.

•   Cycles 20 and 21 are the Fetch and Decode stages of the Data Abort entry respectively.

•   During cycle 22, the processor prepares for FIQ entry, issuing Fetch and Decode cycles in cycles 23 and 24.

•   Therefore, the first instruction in the FIQ routine enters the Execute stage of the pipeline in stage 25, giving a worst-case latency of 24 cycles.

———— **Note** ————

The worst-case latency for IRQs can be longer, at least as long as the worst-case FIQ service routine.

4.4   Minimum interrupt latency

The minimum latency for **FIQ** or **IRQ** is the shortest time the request can be sampled by the input register (one cycle), plus the exception entry time (three cycles). The first interrupt instruction enters the Execute pipeline stage four cycles after the interrupt is asserted.

Chapter 5
Coprocessor Interface

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

5.1 About the coprocessor interface

The ARM7EJ-S processor supports the connection of coprocessors through the coprocessor interface and supports all classes of coprocessor instructions.

Coprocessors determine the instructions they must execute using a *pipeline follower* in the coprocessor. As each instruction arrives from memory, it enters both the ARM pipeline and the coprocessor pipeline. The coprocessor determines when an instruction is being fetched by the ARM7EJ-S processor, so that the instruction can be loaded into the coprocessor, and the pipeline follower advanced.

To ease integration of a coprocessor, the interface from the ARM7EJ-S processor to the coprocessor has been pipelined by a single clock cycle. This provides an optimized interface that significantly eases the implementation task for a coprocessor.

5.2   Synchronizing the coprocessor pipeline

A coprocessor connected to the ARM7EJ-S processor determines which instructions it requires to execute by implementing a pipeline follower. The instruction arrives from the memory and enters the ARM7EJ-S processor pipeline. Because of the pipelined interface between the ARM7EJ-S processor and the coprocessor, the coprocessor pipeline follower operates one cycle behind the ARM7EJ-S processor, sampling the **CPINSTR[31:0]** output bus from the ARM7EJ-S coprocessor interface.

To hide the pipeline delay, a mechanism inside the ARM7EJ-S coprocessor interface block stalls the ARM7EJ-S processor for a cycle whenever an external coprocessor instruction is decoded. This enables the external coprocessor to resynchronize with the ARM7EJ-S processor.

After this initial stall cycle, the two pipelines can be considered synchronized. The ARM7EJ-S processor then informs the coprocessor when instructions move from Decode into Execute, and whether the instruction has passed its condition codes and is to be executed.

5.3 Handshake signals CHSD and CHSE

The handshake signals **CHSD** (decode stage of the pipeline) and **CHSE** (execute stage) are used by the coprocessor to indicate if it can handle the current instruction.

Table 5-1 describes how the handshake signals **CHSD[1:0]** and **CHSE[1:0]** are encoded.

Table 5-1 Handshake signals

State	CHSD/CHSE	Description
ABSENT	b10	If there is no coprocessor attached that can execute the coprocessor instruction, the handshake signals indicate the ABSENT state. In this case, the ARM7EJ-S processor takes the undefined instruction trap.
WAIT	b00	If there is a coprocessor attached that can handle the instruction, but not immediately, the coprocessor handshake signals are driven to indicate that the ARM7EJ-S processor must stall until the coprocessor can catch up. This is known as the busy-wait condition. In this case, the ARM7EJ-S processor loops in an idle state waiting for **CHSE[1:0]** to be driven to another state, or for an interrupt to occur. If **CHSE[1:0]** changes to ABSENT, the undefined instruction trap is taken. If **CHSE[1:0]** changes to GO or LAST, the instruction proceeds as follows. If an interrupt occurs, the ARM7EJ-S processor is forced out of the busy-wait state. This is indicated to the coprocessor by the **CPPASS** signal going LOW. The instruction is restarted later and so the coprocessor must not commit to the instruction (it must not change the coprocessor state) until it has seen **CPPASS** HIGH, when the handshake signals indicate the GO or LAST condition.
GO	b01	The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires another cycle of execution. Both the ARM7EJ-S processor and the coprocessor must also consider the state of the **CPPASS** signal before actually committing to the instruction. For an LDC or STC instruction, the coprocessor instruction drives the handshake signals with GO when two or more words still have to be transferred.
		When only one further word is to be transferred, the coprocessor drives the handshake signals with LAST. During the Execute stage, the ARM7EJ-S processor outputs the address for the LDC or STC. At this stage, the ARM7EJ-S processor also has to perform a data request corresponding to the LDC or STC to be performed. Figure 5-1 on page 5-5 and Figure 5-2 on page 5-6 show examples of these operations.
LAST	b11	An LDC or STC can be used for more than one item of data. If this is the case, possibly after busy waiting, the coprocessor drives the coprocessor handshake signals with a number of GO states, and in the penultimate cycle drives LAST (indicating that the next transfer is the final one). If there is only one transfer, the sequence is [WAIT,[WAIT,...]],LAST.

5.4 LDC operation

The number of words transferred is determined by how the coprocessor drives the **CHSD[1:0]** and **CHSE[1:0]** buses. In the example LDC cycle timing shown in Figure 5-1, one word of data is transferred.
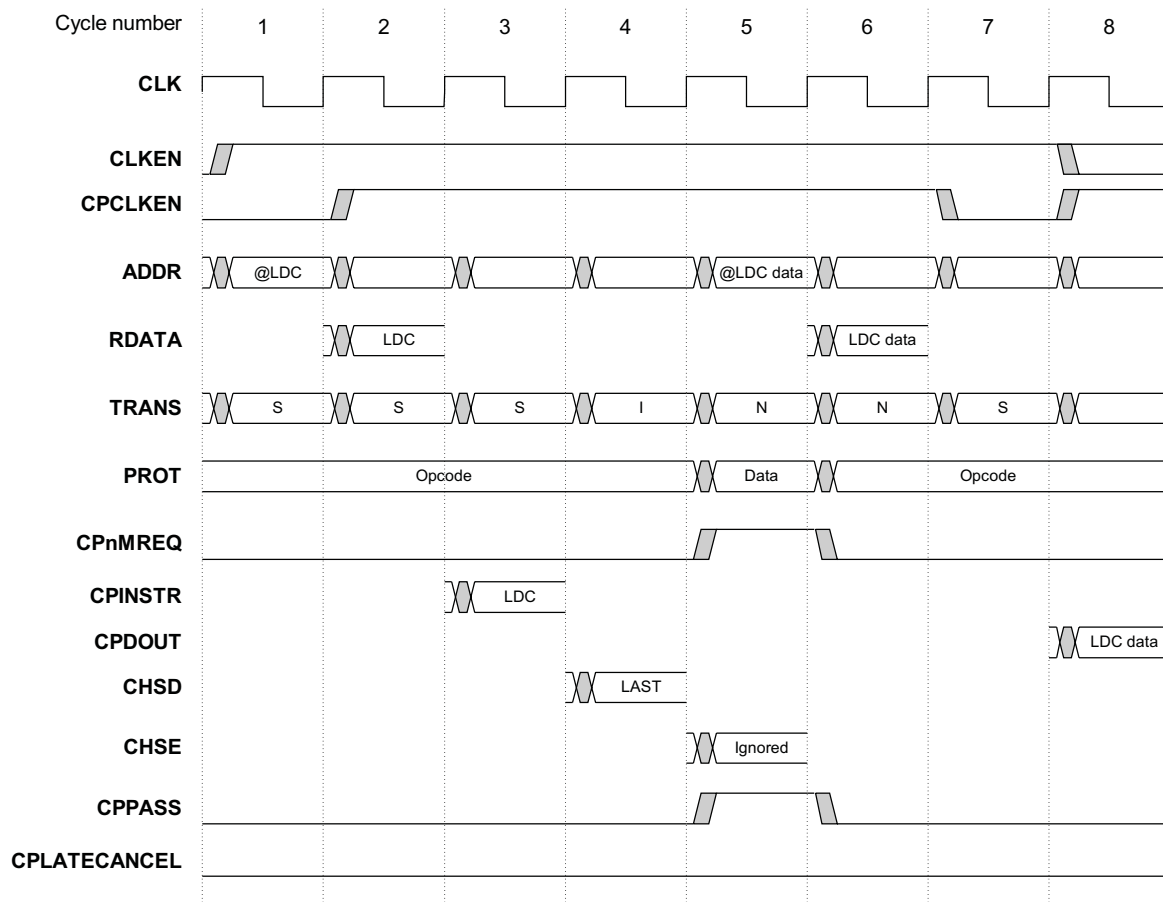
Figure 5-1 LDC cycle timing

――― **Note** ―――

The coprocessor must sample the **CPDOUT** bus at the end of the coprocessor write stage.

5.5 STC operation

The STC operation is similar to LDC, except that one additional WAIT cycle is inserted before the data transfer request. This WAIT cycle ensures that no combinational path exists between the external coprocessor and the memory. Without the extra WAIT state, a combinational path is possible between **CPDIN** and **WDATA**.

The STC operation is shown in Figure 5-2.
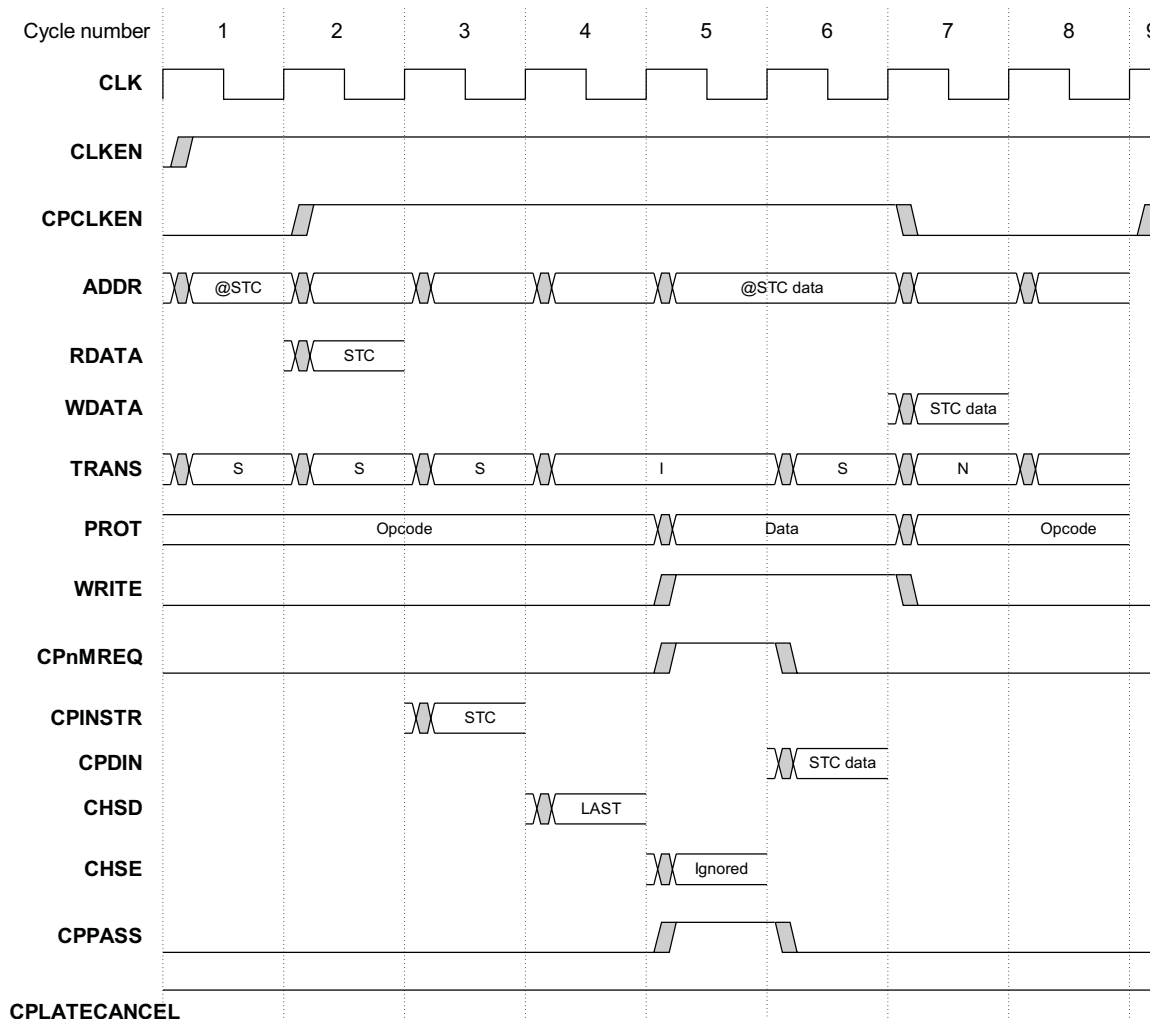
Figure 5-2 STC cycle timing

5.6 MCR operation

An MCR operation is similar to an LDC operation, except that the ARM7EJ-S processor does not have to perform a memory access. This saves one cycle compared with the LDC operation. A timing diagram is shown in Figure 5-3.

The case of an MCR operation to coprocessor 15 is described in *Coprocessor 15 MCR operation* on page 5-9.
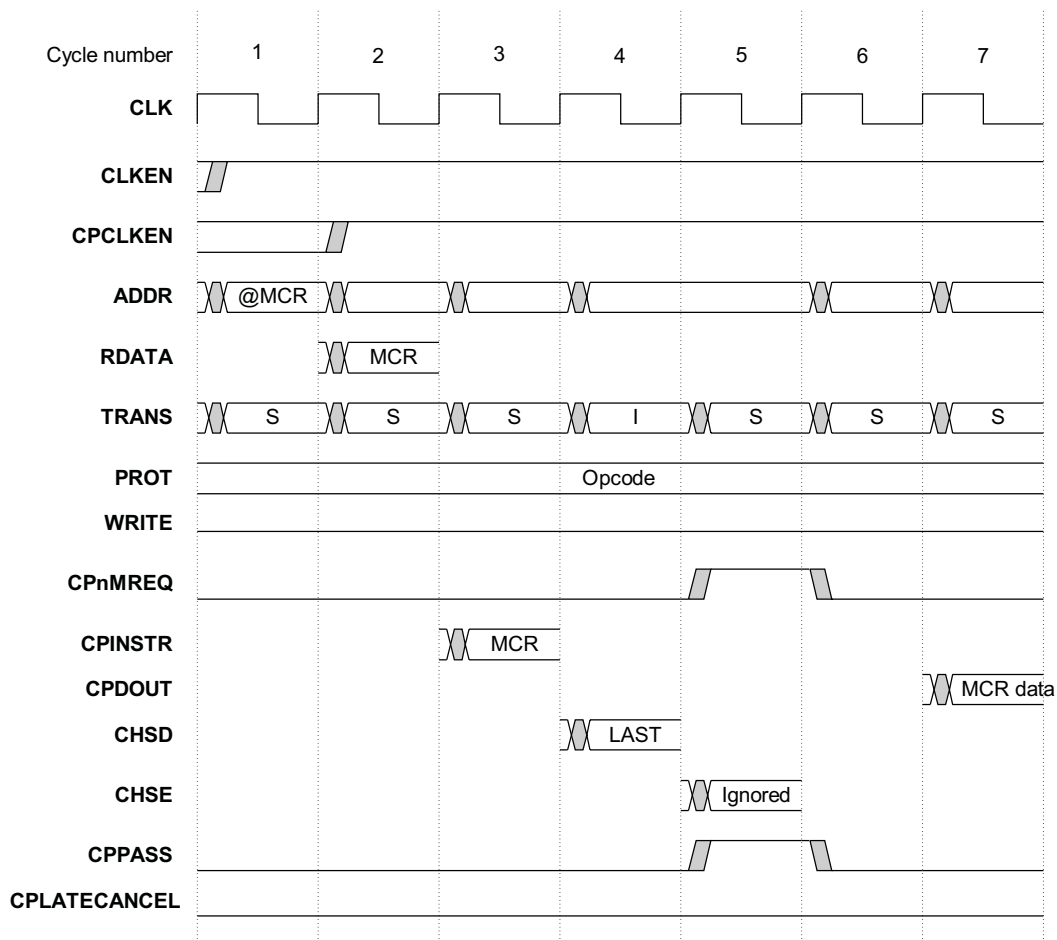
Figure 5-3 MCR cycle timing

Copyright © 2001 ARM Limited. All rights reserved.

——— **Note** ———

In Figure 5-3 on page 5-7, as for an `LDC` operation, the coprocessor still has to sample the **CPDOUT** bus at the end of the write pipeline stage.

————————

5.7   Coprocessor 15 MCR operation

Coprocessor 15 is typically reserved for use as a system control coprocessor. For an MCR
to coprocessor 15, it is possible to transfer the coprocessor data to the coprocessor on
the **ADDR** and **WDATA** busses. To do this the coprocessor must drive GO on the
coprocessor handshake signals for a number of cycles. For each cycle that the
coprocessor responds with GO on the handshake signals, the coprocessor data is driven
onto **ADDR** and **WDATA** as shown in Figure 5-4.
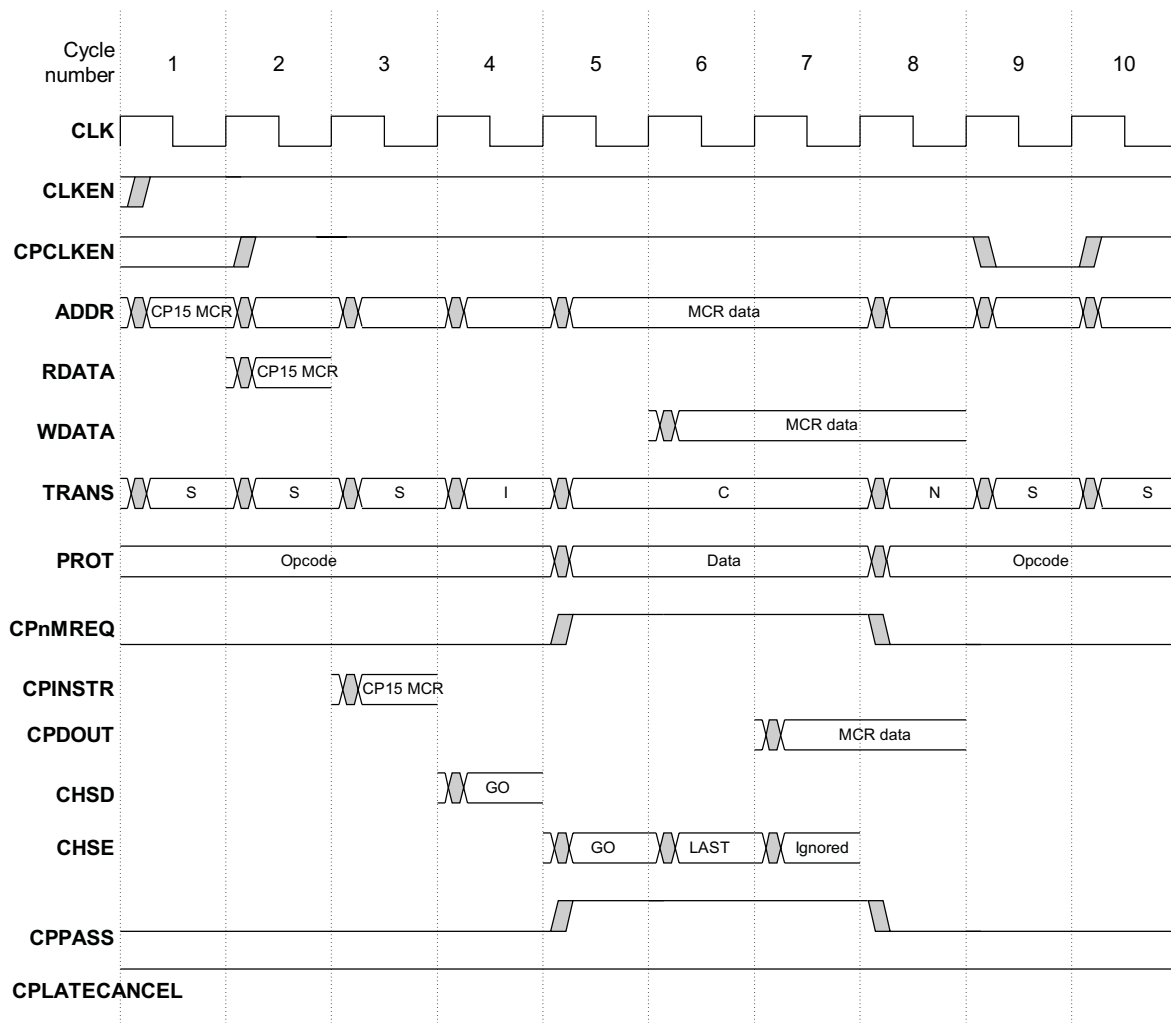
Figure 5-4 Coprocessor 15 MCR cycle timing

Copyright © 2001 ARM Limited. All rights reserved.

5.8 MRC operation

An MRC operation is similar to an MCR operation, except that data is available for the core in cycle 6. The MRC operation is shown in Figure 5-5.
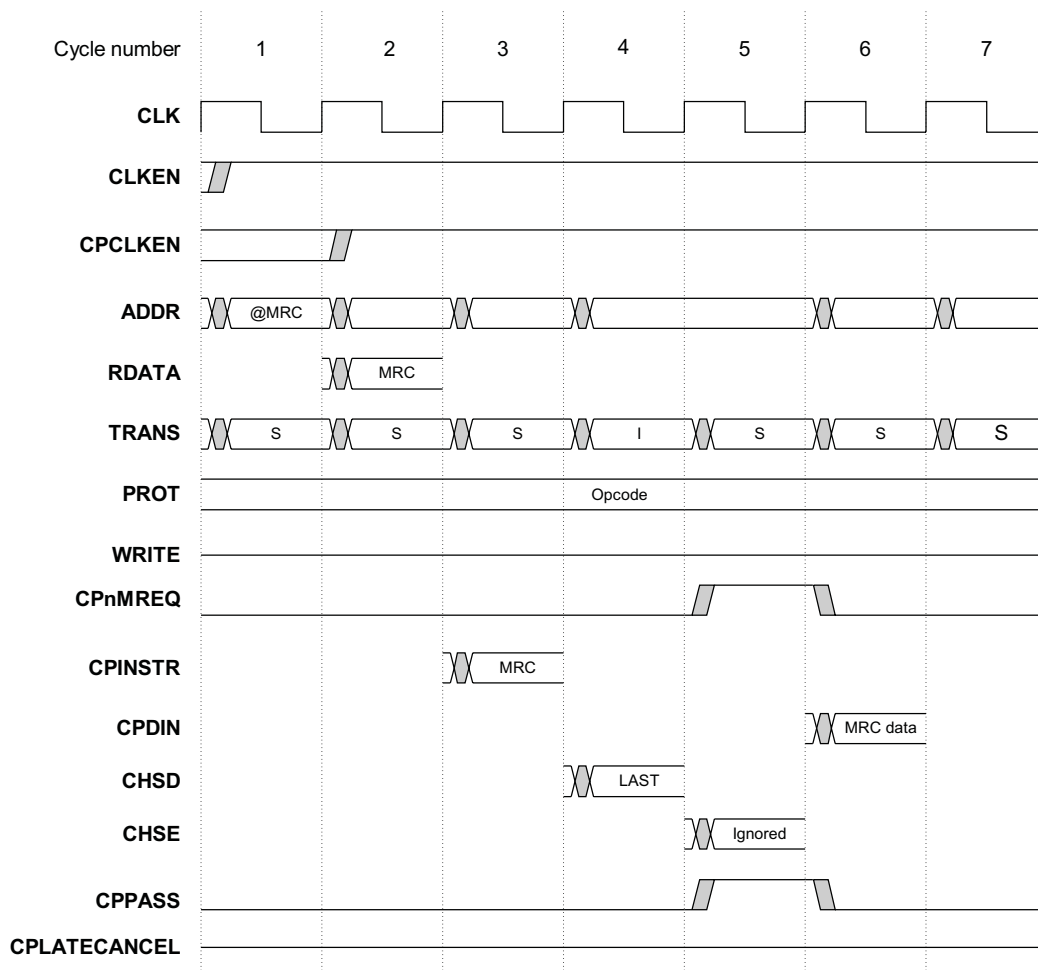
Figure 5-5 MRC cycle timing

5.9 MCRR operation

MCRR operations are very similar to MCR operations, as shown in Figure 5-6. The only difference is that the coprocessor has to remain in the Execute stage for at least two cycles, one for each of the registers to be transferred.

The coprocessor sends a GO onto **CHSD** during the Decode stage of the coprocessor pipeline (corresponding to the first register to be transferred, cycle 4 on the figure), then a LAST onto **CHSE** for the second register (cycle 5).
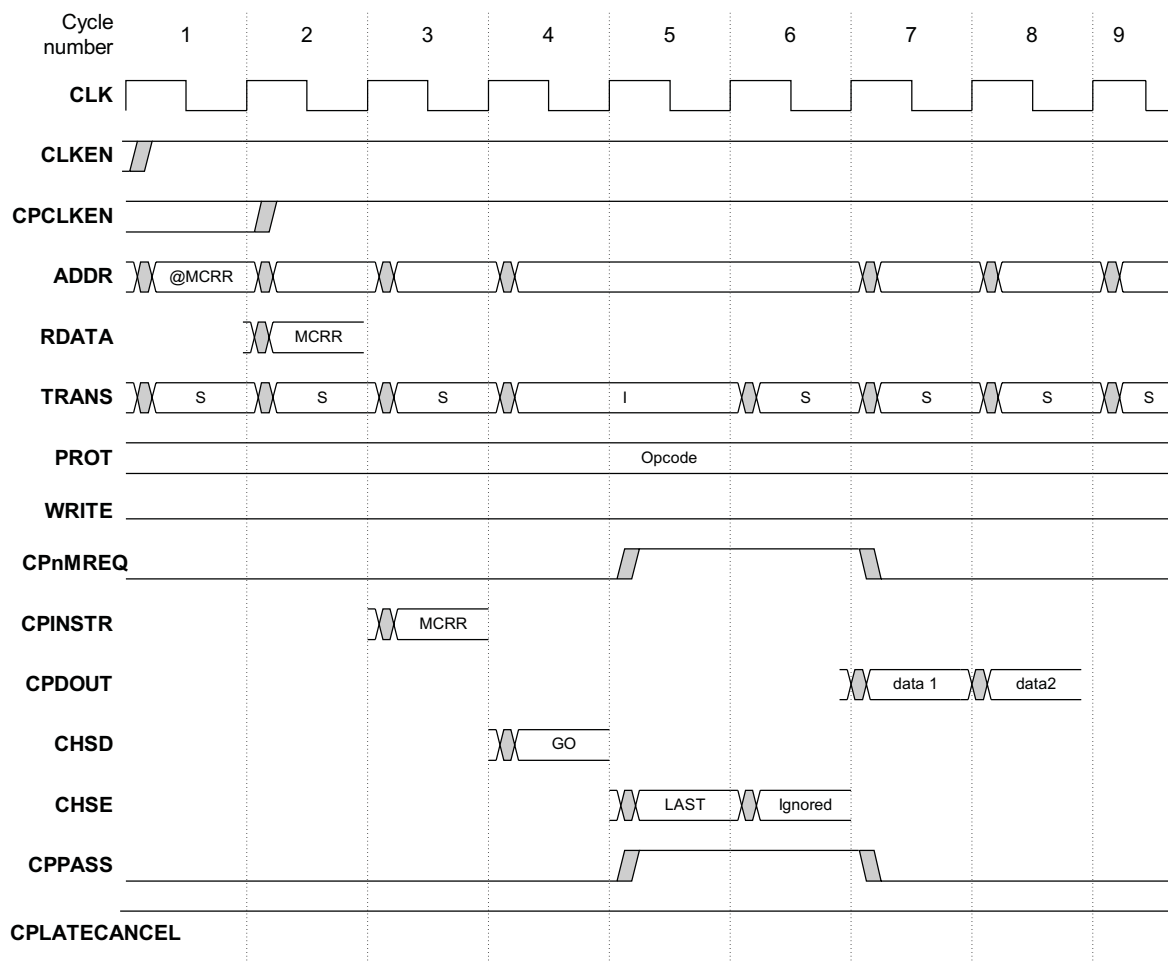
Figure 5-6 MCRR cycle timing

5.10   MRRC operation

MRRC operations are very similar to MRC operations, except that the coprocessor has to remain in its Execute stage for the second register to be transferred. This is the same as the difference between the MRRC and the MRC described in *MCRR operation* on page 5-11. Figure 5-7 shows an example of MRRC operation.
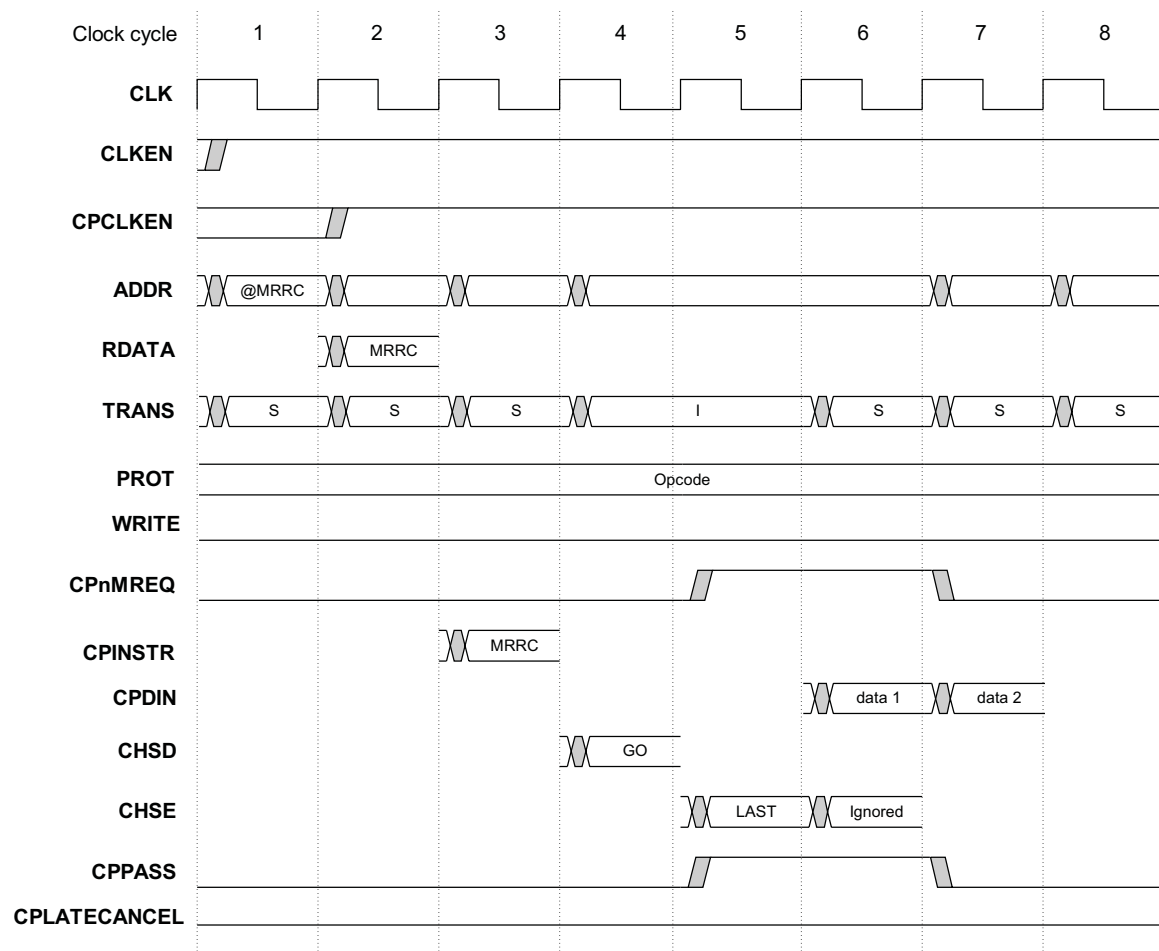
Figure 5-7 MRRC cycle timing

5.11 CDP operation

CDP instructions usually execute in a single cycle:

- if the coprocessor can accept the instruction for execution then the **CPPASS** signal is driven HIGH during the Execute cycle

- if the coprocessor can execute the instruction immediately then the coprocessor drives **CHSD[1:0]** with LAST

- if the instruction requires a busy-wait cycle then the coprocessor drives **CHSD[1:0]** with WAIT and then **CHSE[1:0]** with LAST.

Figure 5-8 on page 5-14 shows a CDP operation that is canceled due to an interrupt detected on **nIRQ**.

In Figure 5-8 on page 5-14, the CDP instruction enters the Execute stage of the pipeline and is signaled to execute by **CPPASS**. In the following cycle **CPLATECANCEL** is asserted. This causes the coprocessor to terminate execution of the CDP instruction and prevents the CDP instruction from causing state changes to the coprocessor.
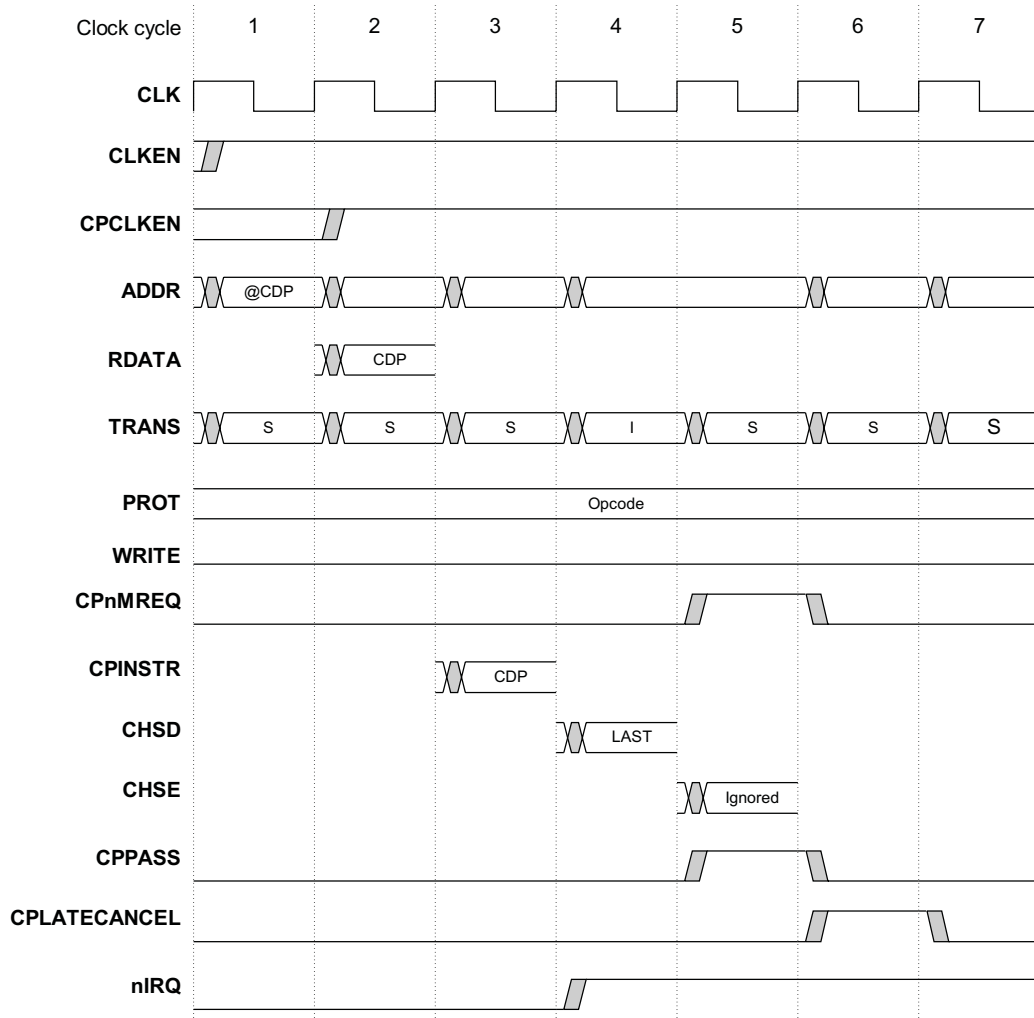
Figure 5-8 CDP cycle timing

5.12   Privileged instructions

The coprocessor might restrict certain instructions for use in privileged modes only. To do this, the coprocessor has to track the **CPnTRANS** output. The **CPnTRANS** information is valid for the coprocessor instruction moving from Decode to Execute stage in the coprocessor pipeline:

- if **CPnTRANS** is LOW then the system is in User mode
- if **CPnTRANS** is HIGH then the system is in a privileged mode.

5.13   Busy-waiting and interrupts

The coprocessor is permitted to stall or busy-wait the ARM7EJ-S processor during the execution of a coprocessor instruction if, for example, it is still busy with an earlier coprocessor instruction. To do this, the coprocessor associated with the Decode stage instruction drives WAIT onto **CHSD[1:0]**. When the instruction concerned enters the Execute stage of the pipeline the coprocessor can drive WAIT onto **CHSE[1:0]** for as many cycles as necessary to keep the instruction in the busy-wait loop.For interrupt latency reasons, the coprocessor can be interrupted while busy-waiting, and cause the current instruction to be abandoned. Abandoning execution is done through **CPPASS**. The coprocessor must monitor the state of **CPPASS** during every busy-wait cycle:

*   if **CPPASS** is HIGH then the instruction must still be executed
*   if **CPPASS** is LOW, the instruction must be abandoned.

5.14   Operating states

CPTBIT and **CPJBIT** are delayed signal outputs that indicate the operating state of the ARM7EJ-S processor to coprocessors:

- in ARM state, the **CPTBIT** and **CPJBIT** signals are LOW
- in Thumb state, the **CPTBIT** signal is HIGH
- in Jazelle state, the **CPJBIT** signal is HIGH.

——— **Note** ———

There are no coprocessor instructions in the Thumb or Jazelle instruction space, so instructions fetched in these states can be ignored to save power in the coprocessor.

5.15 Connecting coprocessors

This section describes how to connect coprocessors to the ARM7EJ-S processor:

- *Connecting a single coprocessor*
- *Connecting multiple coprocessors*.

5.15.1 Connecting a single coprocessor

A coprocessor in an ARM7EJ-S processor system only requires connection to the coprocessor interface, without any connection on the system memory buses.

The list of the signals that must be connected between the ARM7EJ-S processor and the coprocessor is available in *Coprocessor interface signals* on page A-6.

5.15.2 Connecting multiple coprocessors

If you have multiple coprocessors in your system, you must connect the following coprocessor outputs to all coprocessors present in your system:

- **CPCLKEN**
- **CPINSTR**
- **CPDOUT**
- **CPTBIT**
- **CPJBIT**
- **CPnMREQ**
- **CPnTRANS**
- **CPPASS**
- **CPLATECANCEL**.

You must connect the coprocessor inputs as shown in Table 5-2.

Table 5-2 Coprocessor input signal connections

Name	Connection information
CPEN	Static signal that must be tied HIGH.
CHSD and **CHSE**	Combine the individual bit 1 of **CHSD** and **CHSE** by ANDing logically before connecting onto the ARM7EJ-S processor corresponding inputs. Combine the individual bit 0 of **CHSD** and **CHSE** by ORing logically before connecting onto the ARM7EJ-S processor corresponding inputs.
CPDIN	Multiplex the **CPDIN** data output of each coprocessor, or build an OR-tree between all **CPDIN** coprocessor outputs if **CPDIN** is driven LOW when the coprocessor is inactive, so that the ARM7EJ-S processor **CPDIN** input only receives the **CPDIN** value from the active coprocessor.

5.16   No external coprocessors

If you are implementing a system that does not include any external coprocessors, you must tie both **CHSD** and **CHSE** to 10 (ABSENT). This indicates that no external coprocessors are present in the system. If any coprocessor instructions are received, they cause the processor to take the undefined instruction trap, allowing the coprocessor instructions to be emulated in software if required.

The coprocessor enable signal, **CPEN**, must be tied LOW to save power.

The coprocessor-specific outputs from the ARM7EJ-S processor must be left unconnected.

5.17   Undefined instructions

The ARM7EJ-S processor implements full ARMv5TE architecture undefined instruction handling. This means that any instruction described in the *ARM Architecture Reference Manual* as UNDEFINED, automatically causes the ARM7EJ-S processor to take the undefined instruction trap.

Any coprocessor instruction that is not accepted by a coprocessor also results in the ARM7EJ-S processor taking the undefined instruction trap.

Chapter 6
Debug Interface and EmbeddedICE-RT

This chapter describes:

- The ARM7EJ-S processor debug interface in the following sections:
    - *About the debug interface* on page 6-2
    - *Debug systems* on page 6-3
    - *Debug interface signals* on page 6-9
    - *Core clock domains* on page 6-12
    - *Determining the core and system state* on page 6-14
    - *Using Watchpoints and breakpoints in Jazelle state* on page 6-22.

- The ARM7EJ-S processor EmbeddedICE-RT logic in the following sections:
    - *About EmbeddedICE-RT* on page 6-6
    - *Disabling EmbeddedICE-RT* on page 6-8
    - *Debug Communications Channel* on page 6-15
    - *Monitor mode debug* on page 6-20.

6.1 About the debug interface

The ARM7EJ-S 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.

The ARM7EJ-S processor contains hardware extensions for advanced debugging features. These make it easier to develop application software, operating systems, and the hardware itself. The processor supports two modes of debug operation:

- *Halt mode*
- *Monitor mode*.

6.1.1 Halt mode

In halt mode debug, the debug extensions enable the core to be forced into *debug state*. In debug state, the core is stopped and isolated from the rest of the system. This enables the internal state of the core, and the external state of the system, to be examined while all other system activity continues as normal. When debug has been completed, the core and system state can be restored, and program execution resumed.

6.1.2 Monitor mode

On a breakpoint or watchpoint, an Instruction Abort or Data Abort is generated instead of entering debug state in halt mode. When used in conjunction with a debug monitor program activated by the abort exception entry, it is possible to debug the processor while enabling the execution of critical interrupt service routines. The debug monitor program typically communicates with the debug host over the debug communication channel. Monitor mode debug is described in *Monitor mode debug* on page 6-20.

6.2 Debug systems

The ARM7EJ-S processor forms one component of a debug system that interfaces from the high-level debugging performed by the user to the low-level interface supported by the ARM7EJ-S processor. Figure 6-1 shows a typical debug system.
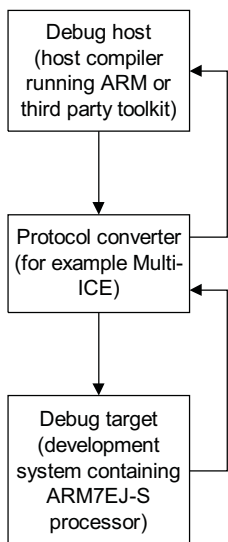
Figure 6-1 Typical debug system

A debug system typically has three parts:

- *The debug host*
- *The protocol converter* on page 6-4
- *The ARM7EJ-S processor* on page 6-4 (the debug target).

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

6.2.1 The debug host

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

6.2.2   The protocol converter

An interface, such as an RS232 or parallel connection, connects the debug host to the ARM7EJ-S processor development system. The messages broadcast over this connection must be converted to the interface signals of the processor. The protocol converter performs this conversion.

6.2.3   The ARM7EJ-S processor

The ARM7EJ-S processor has hardware extensions that ease debugging at the lowest level. The debug extensions:

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

The major blocks of the ARM7EJ-S processor are:

ARM7EJ-S core

This is the processor core, with hardware support for debug.

EmbeddedICE-RT logic

This is a set of registers and comparators used to generate debug exceptions (such as breakpoints). This unit is described in *About EmbeddedICE-RT* on page 6-6.

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

ETM interface   This interface facilitates connection of an *Embedded Trace Macrocell* (ETM) to the core. The ETM is described in the *ETM9 (Rev 2a) Technical Reference Manual*.

These blocks are shown in Figure 6-2 on page 6-5.
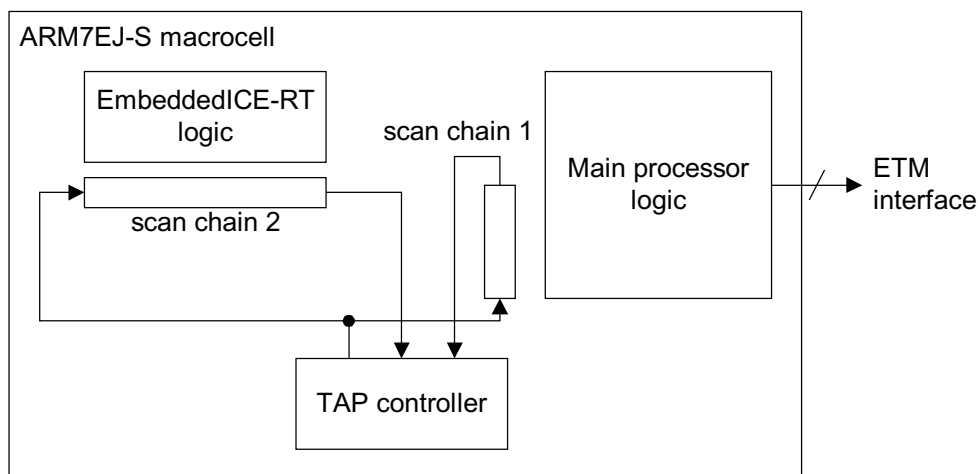
Figure 6-2 Debug block diagram

In halt mode, a request on one of the external debug interface signals, or on an internal functional unit known as the *EmbeddedICE-RT logic*, forces the processor into debug state. The events that activate debug are:

- a breakpoint (a given instruction fetch)
- a watchpoint (a data access)
- an external debug request
- scanned debug request (a debug request scanned into the EmbeddedICE-RT delay control register).

The internal state of the processor is examined using the JTAG serial interface, that enables instructions to be serially inserted into the core pipeline without using the external data bus. So, for example, when in debug state, a *store multiple* (STM) instruction can be inserted into the instruction pipeline, and this exports the contents of the core registers. This data can be serially shifted out without affecting the rest of the system.

6.3   About EmbeddedICE-RT

The EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM7EJ-S processor.

EmbeddedICE-RT is programmed serially using the TAP controller. Figure 6-3 shows the relationship between the core, EmbeddedICE-RT, and the TAP controller. It only shows the signals that are pertinent to EmbeddedICE-RT.
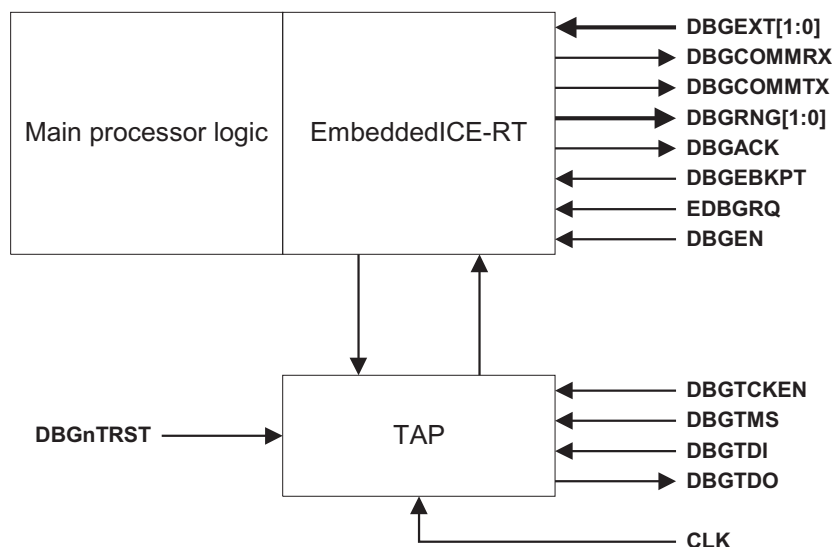
Figure 6-3 Major debug components

The EmbeddedICE-RT logic comprises:

* two real-time watchpoint units
* two independent registers, the debug control register and the debug status register
* *Debug Communications Channel* (DCC).

The debug control register and the debug status register provide overall control of EmbeddedICE-RT operation.

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

———— **Note** ————

You can mask 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 in halt mode when in debug state.

The EmbeddedICE-RT logic can be configured into a mode of operation where watchpoints or breakpoints generate Data or Prefetch Aborts respectively. This enables a real-time debug monitor system to debug the processor while still allowing critical fast interrupt requests to be serviced.

6.4   Disabling EmbeddedICE-RT

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

———— **Caution** ————

Hard-wiring the **DBGEN** input LOW *permanently* disables all debug functionality.

———————————

When **DBGEN** is LOW:
*   **DBGEBKPT** and **EDBGRQ** are inhibited
*   **DBGACK** is always LOW.

6.5 Debug interface signals

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

- **DBGEBKPT** and **EDBGRQ** are system requests for the core to enter debug state
- **DBGEN** is used to enable or disable debug state
- **DBGACK** is used by the core to flag back to the system that it is in debug state.

6.5.1 Entry into debug state on breakpoint

An instruction being fetched from memory is sampled at the end of a cycle. To apply a breakpoint to that instruction, the breakpoint signal must be asserted by the end of the same cycle. This is shown in Figure 6-4.

Pipeline stages shown in italics occur in debug state
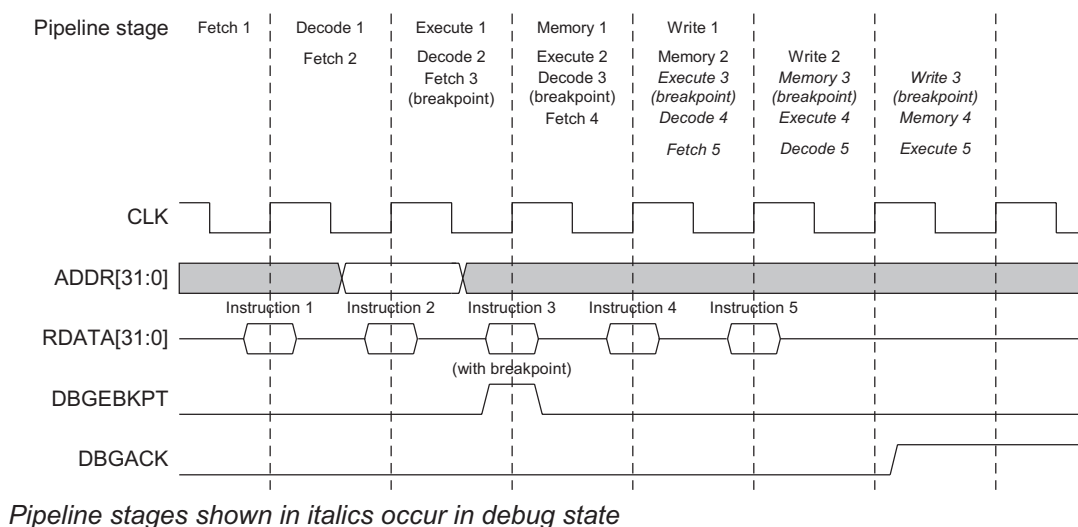
Figure 6-4 Breakpoint timing

You can build external logic, such as additional breakpoint comparators, to extend the breakpoint functionality of the EmbeddedICE-RT logic. You must apply their output to the **DBGEBKPT** input.

——— **Note** ———

The timing of the **DBGEBKPT** input makes it unlikely that data-dependent external breakpoints are possible. **DBGEBKPT** is not supported in Jazelle state, and must not be asserted while the core is in Jazelle state.

———

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

——— **Note** ———

If a breakpointed instruction does not reach the Execute stage, for instance, if an earlier instruction is a branch, then both the breakpointed instruction and breakpoint status are discarded and the ARM does not enter debug state.

In Figure 6-4 on page 6-9 the third instruction breakpointed. The debug entry sequence is initiated when this instruction enters the Execute stage. The processor completes the debug entry sequence and asserts **DBGACK** two cycles later.

6.5.2 Breakpoints and exceptions

A breakpointed instruction can have a Prefetch Abort associated with it. If so, the Prefetch Abort takes priority and the breakpoint is ignored. If there is a Prefetch Abort with invalid instruction data, and the breakpoint is data-dependent then the breakpoint is triggered incorrectly.

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

On an instruction boundary, if there is a breakpointed instruction and an interrupt occurs, the interrupt is taken and the breakpointed instruction is discarded. When the interrupt has been serviced, the execution flow is returned to the original program. This means that the instruction which was previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters debug state when it reaches the execute stage of the pipeline.

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

6.5.3 Watchpoints

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

You can build external logic, such as external watchpoint comparators, to extend the functionality of the EmbeddedICE-RT logic. You must apply their output to the **DBGEBKPT** input.

After a watchpointed access, the next instruction in the processor pipeline is always allowed to complete execution. Where this instruction is a single-cycle data-processing instruction, entry into debug state is delayed for one cycle while the instruction completes.

6.5.4 Watchpoints and exceptions

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

6.5.5 Debug request

A debug request can take place through the EmbeddedICE-RT logic or by asserting the **EDBGRQ** signal. The request is registered and passed to the processor. Debug request takes priority over any pending interrupt. Following registering, the core enters debug state when:

• the breakpointed instruction at the Execute stage of the pipeline has completed

• the Memory and Write stages of the pipeline have completed for instructions issued prior to the breakpointed instruction.

While waiting for the instruction to finish executing, no more instructions are issued to the Execute stage of the pipeline.

When a hardware debug request occurs, the processor enters debug state even if the EmbeddedICE-RT logic is configured for monitor mode debug.

6.5.6 Actions of the processor in debug state

When the processor is in debug state the memory interface indicates internal cycles. This enables the rest of the memory system to ignore the processor and function as normal. Because the rest of the system continues operation, the processor ignores aborts and interrupts.

The **CFGBIGEND** signal must not be changed by the system while in debug state. If it changes, not only is there a synchronization problem, but the view of the processor seen by the programmer changes without the knowledge of the debugger. The **nRESET** signal must also be held stable during debug. If the system applies reset to the processor (**nRESET** is driven LOW), the state of the processor changes without the knowledge of the debugger.

6.6   Core clock domains

The ARM7EJ-S processor has a single clock, **CLK**, that is qualified by two clock enables:

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

During normal operation, **CLKEN** qualifies **CLK** to clock the core. When the ARM7EJ-S processor is in debug state, **DBGTCKEN** qualifies **CLK** to clock the core.

6.6.1   Clocks and synchronization

If the system and test clocks are asynchronous, they must be synchronized externally to the ARM7EJ-S processor. The ARM Multi-ICE debug agent directly supports one or more cores within an ASIC design. Synchronized off-chip debug clocking with the ARM7EJ-S processor 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 be returned. Synchronization is maintained because the off-chip device does not progress to the next **TCK** until after **RTCK** is received. Figure 6-5 on page 6-13 shows this synchronization.
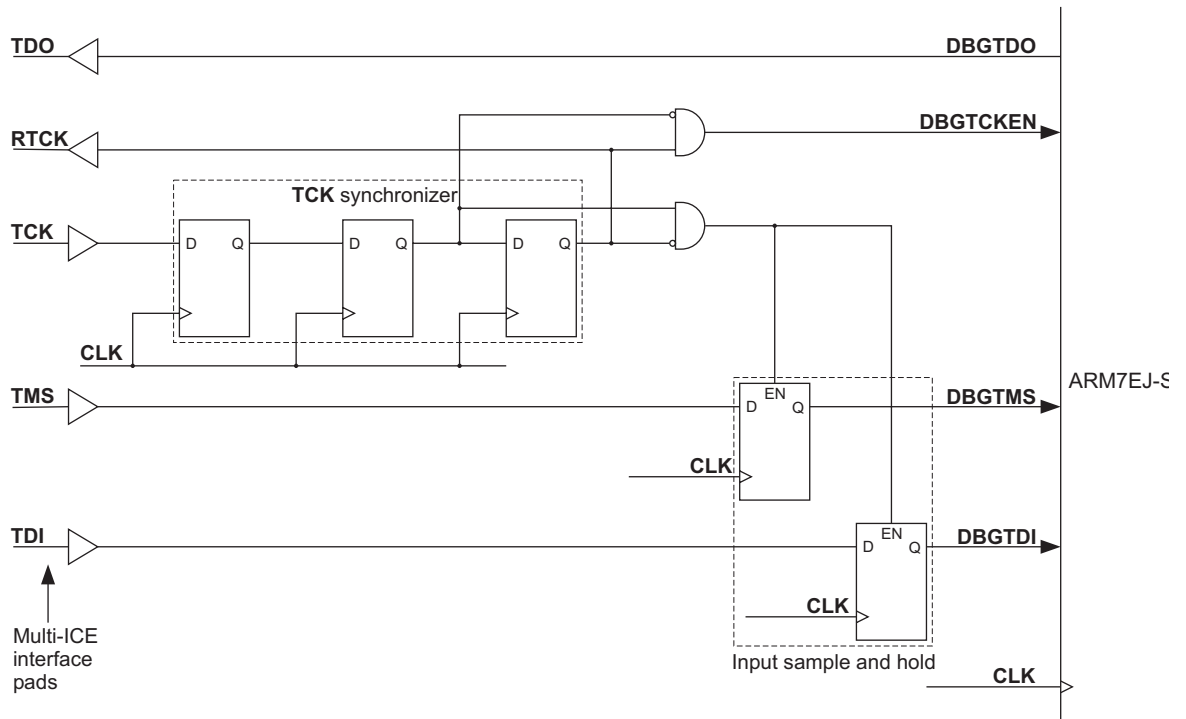
Figure 6-5 Clock synchronization

6.7   Determining the core and system state

When the core is in debug state, you can examine the core and system state by forcing the load and store multiple instructions into the pipeline.

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

For more details about determining the core state, see *Determining the core and system state* on page B-17.

6.8   Debug Communications Channel

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

The DCC comprises:

*   a 32-bit wide communications data read register
*   a 32-bit wide communications data write register
*   a 32-bit wide (only 6 bits are used) communications control register for synchronized handshaking between the processor and the asynchronous debugger.

These registers are located in fixed locations in the EmbeddedICE-RT logic register map (as described in *EmbeddedICE-RT logic* on page B-27) and are accessed from the processor using `MCR` and `MRC` instructions to coprocessor 14.

In addition to the communications channel registers, the processor can access a bit 0 of the 32-bit debug status register for use in the monitor mode debug configuration.

6.8.1   DCC registers

Coprocessor 14 contains 4 registers, allocated as shown in Table 6-1.

Table 6-1 Coprocessor 14 register map

Register name	Register number	Notes
Control	C0	Read only[a]
Data read	C1	For reads
Data write	C1	For writes
Monitor mode debug status	C2	Read/write

a. You can clear bit 0 of the communications channel control register by writing to it from the debugger (JTAG) side.

The registers are accessed by the debugger using the scan chain in the usual way. They are accessed by the processor using coprocessor register transfer instructions.

6.8.2 DCC control register

The DCC control register is read-only.

─── **Note** ───

The control register should be viewed as read-only. However, the debugger can clear the R bit by performing a write to the DCC control register. This feature must not be used under normal circumstances.

───────────

The register controls synchronized handshaking between the processor and the debugger. The DCC control register is shown in Figure 6-6.
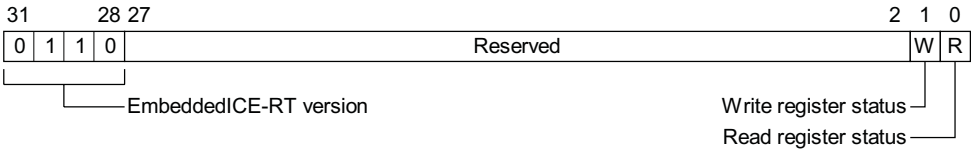
Figure 6-6 DCC control register

The function of each register bit is described below:

Bits 31:28   Contain a fixed pattern that denotes the EmbeddedICE version number (b0110 in this case).

Bits 27:2   Are reserved.

Bit 1   Denotes if the communications data write register is available (from the viewpoint of the processor). Seen from the processor, if the communications data write register is free (bit 1 is clear), new data can be written. If the register is not free (bit 1 is set), the processor must poll until bit 1 is clear.Seen from the debugger, when W=1, some new data has been written that can then be scanned out.

Bit 0   Denotes if there is new data in the communications data read register. Seen from the processor, if bit 0 is clear, there is some new data that can be read using an MRC instruction.

Seen from the debugger, if bit 0 is set, the communications data read register is free, and new data may be placed there through the scan chain. If R=1, this denotes that data previously placed there through the scan chain has not been collected by the processor, and so the debugger must wait.

You can use the following instructions to access these registers:

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

The above instruction returns the DCC control register into Rd.

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

The above instruction writes the value in Rn to the communications data write register.

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

The above instruction returns the debug data read register into Rd.

―――― **Note** ――――

The Thumb instruction set does not support coprocessor instructions. Therefore, the processor must be in ARM state before you can access the DCC.

6.8.3 DCC monitor mode debug status register

The coprocessor 14 monitor mode debug status register is provided for use by a debug monitor when the processor is configured into the monitor mode debug mode.

The coprocessor 14 monitor mode debug status register is a 32-bit wide register with one read/write bit with the format shown in Figure 6-7.
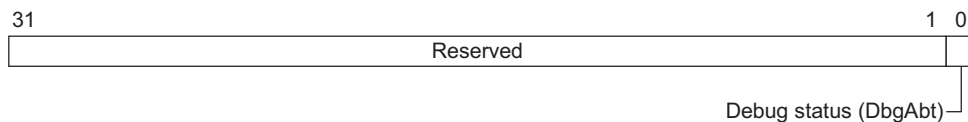
Figure 6-7 Monitor mode debug status register

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

A typical use of this bit is by a monitor mode debug aware abort handler. This examines the DbgAbt bit to determine whether the abort was externally or internally generated. If the DbgAbt bit is set, the abort handler initiates communication with the debugger over the communications channel.

6.8.4 Using the debug communications channel

You can send and receive messages using the DCC. These are described in:

* *Sending a message to the debugger*
* *Receiving a message from the debugger*.

Sending a message to the debugger

Before the processor can send a message to the debugger, it must check that the communications data write register is free for use by finding out if the W bit of the DCC control register is clear. The processor reads the DCC control register to check the status of the W bit:

* If the W bit is clear, the communications data write register is clear.

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

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

The debugger has two options available for reading data from the communications data write register:

* Poll the DCC control register before reading the communications data written. If the W bit is set, there is valid data present in the DCC data write register. The debugger can then read this data and scan the data out. The action of reading the data clears the DCC control register W bit. Then the communications process can begin again.

* Poll the DCC data write register, obtaining data and valid status. The data scanned out consists of the contents of the communications data write register (which might or might not be valid), and a flag that indicates whether the data read is valid or not. The status flag is present in the Addr[0] bit position of scan chain 2 when the data is scanned out. See *Test data registers* on page B-10 for details of scan chain 2.

Receiving a message from the debugger

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

* If the R bit is LOW, the DCC data read register is free, and data can be placed there for the processor to read.

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

When the DCC 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.

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

6.8.5 Debug communications channel reset

The communications channel has the following behavior during reset:

- During the assertion of **DBGnTRST** (LOW):
  - DCC CP14 accesses are not bounced
  - reading the DCC control register returns b10 in the bottom two bits of the register (write register full, read register empty)
  - **DBGCOMMRX** is reset asynchronously to LOW (read buffer empty)
  - **DBGCOMMTX** is reset asynchronously to LOW (transmit buffer full).

- On removal of reset on **DBGnTRST** (LOW to HIGH), after the next rising edge of **CLK**:
  - reading the DCC control register returns b00 in the bottom two bits of the register (write register empty, read register empty)
  - **DBGCOMMTX** is HIGH (transmit buffer empty).

6.9   Monitor mode debug

The ARM7EJ-S processor contains logic that enables the debugging of a system without stopping the core entirely. This enables the continued servicing of critical interrupt routines while the core is being interrogated by the debugger. Setting bit 4 of the DCC control register enables the monitor mode debug features of the core. When this bit is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the core to enter abort mode, taking the Prefetch or Data Abort vectors respectively.

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

*   Breakpoints and watchpoints cannot be data-dependent. No support is provided for use of the range functionality. Breakpoints and watchpoints can only be based on:
    —   instruction or data addresses
    —   external watchpoint conditioner
    —   User or Privileged mode access
    —   read/write data access (watchpoints)
    —   access size
    —   chained comparisons.

*   The single-step hardware must not be enabled.

*   External breakpoints or watchpoints are not supported.

*   The vector catching hardware can be used but must not be configured to catch the Prefetch or Data Abort exceptions.

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

The fact that an abort has been generated by monitor mode is recorded in the monitor mode debug status register in CP14 (see *DCC monitor mode debug status register* on page 6-17).

Because the monitor mode debug bit does not put the core into debug state, it now becomes necessary to change the contents of the watchpoint registers while external memory accesses are taking place, rather than changing them when in debug state. If the watchpoint registers are altered during an access, all matches from the affected watchpoint unit using the register being updated are disabled for the cycle of the update.

                       ARM DDI 0214B

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

1.   Disable the watchpoint unit using the control register for that watchpoint unit.

2.   Alter the values in the other registers.

3.   Re-enable the watchpoint unit by rewriting the control register.

6.10 Using Watchpoints and breakpoints in Jazelle state

The following information is detailed in this section:

• *Watchpoints*

• *Breakpoints*

• *Monitor mode*.

This section describes how to use watchpoints and breakpoints in Jazelle state.

——— **Note** ———

Breakpoints and watchpoints can have exceptions occurring at the same time. The behavior of these are described in:

• *Breakpoints and exceptions* on page 6-10

• *Watchpoints and exceptions* on page 6-11.

6.10.1 Watchpoints

Watchpoint comparisons are on data addresses and data values. The comparison mechanism is unaltered for Jazelle state. The watchpoint is taken in the same fashion as in ARM state. This means that after the load or store instruction that caused the watchpoint has executed a further execute cycle is completed.

6.10.2 Breakpoints

Breakpoint comparisons are made on the instruction address. There is also a dedicated breakpoint instruction available in Jazelle state.

——— **Note** ———

Instruction data comparisons are not supported in Jazelle state.

6.10.3 Monitor mode

In monitor mode debug:

Watchpoints Execute the LDR/STR instruction and then take a Data Abort. The link register is the PC value of the last instruction not to execute + 4. This enables the following return command to be used, as in ARM state:

SUBS pc, lr, #4

Breakpoints Cause Prefetch Aborts. The link register is calculated in the same way as normal Prefetch Aborts. The return instruction remains:

```
SUBS pc, lr, #4
```

Chapter 7
Embedded Trace Macrocell Interface

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

- *About the ETM interface* on page 7-2
- *Enabling and disabling the ETM interface* on page 7-3.

7.1   About the ETM interface

The ARM7EJ-S processor supports the connection of an external ETM to provide real-time code tracing of the ARM7EJ-S processor in an embedded system.The ETM interface is primarily one way. To provide code tracing, the ETM block must be able to monitor various ARM7EJ-S processor signals, which are collected and driven out from the ARM7EJ-S processor as the ETM interface.The ETM interface outputs are pipelined by two clock cycles to provide early output timing and to isolate any ETM input load from the critical ARM7EJ-S processor signals. The latency of the pipelined outputs does not effect ETM trace behavior, because all outputs are delayed by the same amount.

7.2   Enabling and disabling the ETM interface

The ARM7EJ-S processor ETM interface has one input, **ETMPWRDOWN**, which is used by the ETM to switch the ARM7EJ-S processor ETM interface on and off.

When **ETMPWRDOWN** is LOW, the ETM interface is enabled and the outputs are driven so that an external ETM can begin code tracing.

When **ETMPWRDOWN** is driven HIGH, the ETM interface outputs are held at their last value before the interface was disabled.

———— **Note** ————

If an ETM is not used in an embedded ARM7EJ-S processor design then the **ETMPWRDOWN** input must be tied HIGH to save power.

Copyright © 2001 ARM Limited. All rights reserved.

Chapter 8
Device Reset

This chapter describes the reset behavior. It contains the following sections:

- *About device reset* on page 8-2
- *Reset modes* on page 8-3.

8.1   About device reset

This section describes the ARM7EJ-S processor reset signals and how you must use them for correct operation of the device.

The ARM7EJ-S processor has two reset inputs:

nRESET   This is the main processor reset that initializes the majority of the core logic.

DBGnTRST This the debug logic reset that you can use to reset the TAP controller and the EmbeddedICE-RT unit.

Both **nRESET** and **DBGnTRST** are active LOW signals that asynchronously reset logic in the core. You must take care, when designing the logic, to drive these reset signals correctly.

8.2   Reset modes

Two reset signals are present in the ARM7EJ-S processor design to enable you to reset different parts of the design independently. A description of the reset signaling combinations and possible applications is shown in Table 8-1.

Table 8-1 Reset modes

Mode	nRESET	DBGnTRST	Application
Full system reset	LOW	LOW	Reset at power up, full system reset.
Core reset	LOW	HIGH	Reset of processor core only.
EmbeddedICE-RT reset	HIGH	LOW	Reset of EmbeddedICE-RT circuitry.
Normal	HIGH	HIGH	No reset. Normal run mode.

8.2.1   Full system reset

You must apply a full system reset to the ARM7EJ-S processor when power is first applied to the system. In this case, the leading (falling) edge of **nRESET** and **DBGnTRST** do not have to be synchronous to **CLK**. The trailing (rising) edge of the reset signals must be set up and held around the rising edge of the clock. You must do this to ensure that the entire system leaves reset in a predictable manner. This is particularly important in multi-processor systems. Figure 8-1 shows the application of system reset.
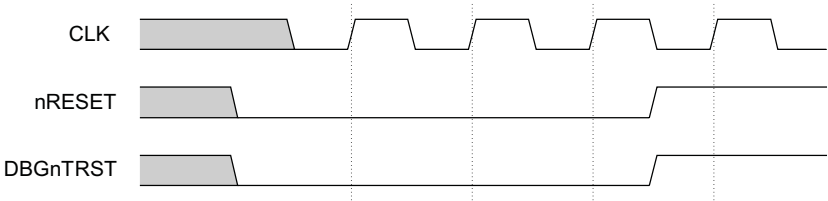
Figure 8-1 System reset

It is recommended that you assert the reset signals for at least three **CLK** cycles to ensure correct reset behavior. Adopting a three-cycle reset eases the integration of other ARM-supplied components into the system.

8.2.2   Core reset

A core reset initializes the majority of the ARM7EJ-S processor, excluding the TAP controller and the EmbeddedICE-RT unit. Core reset is typically used for resetting a system that has been operating for some time (for example, watchdog reset).

Sometimes you might not want to reset the EmbeddedICE-RT unit when resetting the rest of the core, for example, if EmbeddedICE-RT has been configured to breakpoint (or capture) fetches from the reset vector.

For core reset, both the leading and trailing edges of **nRESET** must be set up and held around the rising edge of **CLK**. This ensures that there are no metastability issues between the ARM7EJ-S processor and the EmbeddedICE-RT unit.

8.2.3   EmbeddedICE-RT reset

EmbeddedICE-RT reset initializes the state of the TAP controller and the EmbeddedICE-RT unit. EmbeddedICE-RT reset is typically used by the Multi-ICE module for connecting a debugger without powering down the system.

EmbeddedICE-RT reset enables initialization of the EmbeddedICE-RT unit without affecting the normal operation of the ARM7EJ-S processor.

For EmbeddedICE-RT reset, both the leading and trailing edges of **DBGnTRST** must be set up and held around the rising edge of **CLK**. This ensures that there are no metastability issues between the ARM7EJ-S processor and the EmbeddedICE-RT unit.

Refer to *Clocks and synchronization* on page 6-12 for more details of synchronization between the Multi-ICE and ARM7EJ-S processor.

8.2.4   Normal operation

During normal operation, neither core reset nor EmbeddedICE-RT reset is asserted.

Chapter 9
Instruction Cycle Times

This chapter describes the instruction cycle timings and illustrates the interlock conditions present in the ARM7EJ-S processor design. It contains the following sections:

- *Store double register* on page 9-31
- *Data swap* on page 9-32
- *Software interrupt, undefined instruction, and exception entry* on page 9-33
- *Coprocessor data processing operation* on page 9-35
- *Load coprocessor register (from memory)* on page 9-36
- *Store coprocessor register (to memory)* on page 9-39
- *Coprocessor register transfer (to ARM)* on page 9-42
- *Coprocessor register transfer (from ARM)* on page 9-43
- *Double coprocessor register transfer (to ARM)* on page 9-44
- *Double coprocessor register transfer (from ARM)* on page 9-45
- *Coprocessor absent* on page 9-46
- *Unexecuted instructions* on page 9-48.

9.1 About instruction cycle timings

The pipelined architecture of the ARM7EJ-S processor overlaps the execution of several instructions in different pipeline stages. The tables in this chapter show the number of cycles required by an instruction when it has reached the Execute stage of the pipeline.

The *instruction cycle count* is the number of cycles for which an instruction occupies the Execute stage of the pipeline. The other pipeline stages (Fetch, Decode, Memory, and Writeback) are only occupied for one cycle by any instruction (in this model, interlock cycles are grouped in with the instruction generating the data that creates the interlock condition, not the instruction dependent on the data).

The request, address, and control signals on the memory interface are pipelined so that they are generated in the cycle before the one to which they apply. They are shown this way in the tables in this chapter.

The instruction address, **ADDR[31:0]**, is incremented for prefetching instructions in most cases. The increment varies with the instruction length:

- 4 bytes in ARM state
- 2 bytes in Thumb state.

The letter i is used to indicate the instruction length.

———— **Note** ————

All cycle counts in this chapter assume zero wait-state memory access. In a system where **CLKEN** is used to add wait states, the cycle counts must be adjusted accordingly.

9.1.1 Key to tables in this chapter

Table 9-1 shows the key to the other tables in this chapter.

Table 9-1 Key to tables in this chapter

Symbol	Meaning
b	The number of busy-wait states during coprocessor accesses.
n	The number of words transferred in any of the following instructions: `LDMSTMLDCSTC`
C	Coprocessor register transfer cycle (C-cycle).
I	Internal cycle (I-cycle).

Table 9-1 Key to tables in this chapter (continued)

Symbol	Meaning
N	Nonsequential cycle (N-cycle).
S	Sequential cycle (S-cycle).
pc	The address of the branch instruction.
pc'	The branch target address.
(pc')	The memory contents of the branch target address.
i	4 when in ARM state, or 2 when in Thumb state.
-	Indicates that the signal is not active, and is therefore not valid in this cycle.
	A blank entry in a table indicates that the status of the signal is not determined by the instruction in that cycle. The status of the signal is determined either by the preceding or following instruction.

9.1.2 Instruction cycle counts

Table 9-2 shows the ARM7EJ-S processor instruction cycle counts and bus activity during execution of the ARM instruction set.

Table 9-2 ARM instruction cycle counts and bus activity

Instruction	Cycles	Memory bus	Comment
CLZ	1	1S	All cases.
Data operation	1	1S	Normal case, PC not destination.
Data operation	2	1S+1I	With register controlled shift, PC not destination.
Data operation	3	2S+1N	PC destination register, arithmetic data output (ADD, SUB, RSB, ADC, SBC).
Data operation	4	2S+1N+1I	PC destination register, logical data output (RSC, ORR, EOR, MOV, BIC).
Data operation	4	2S+1N+1I	With operand shift, PC destination register.
LDR	2	2N	Normal case, not loading PC.
LDR	2	2N	Not loading PC and following instruction uses loaded word.

Table 9-2 ARM instruction cycle counts and bus activity (continued)

Instruction	Cycles	Memory bus	Comment
LDR	3	1I+2N	Not loading PC and shifted offset.
LDR	3	1I+2N	Not loading PC and shifted offset and following instruction uses loaded word.
LDR	3	2N+1I	Loaded byte, halfword, or unaligned word used by following instruction (1-cycle load-use interlock).
LDR	5	2N+1I+2S	PC is destination register.
LDR	6	2N+2I+2S	PC is destination register, scaled register offset.
LDRD	3	1S+2N	Normal case.
LDRD	4	1S+2N	Last loaded word used by following instruction.
STR	2	2N	Normal case.
STR	3	1I+2N	Scaled offset.
STRD	3	1S+2N	All cases.
LDM	2	2N	Loading 1 register, not the PC.
LDM	n+1	2N+(n-1)S	Loading n registers, n > 1, not loading the PC.
LDM	n+1	2N+(n-1)S	Loading n registers, n > 1, not loading the PC, last word loaded used by following instruction.
LDM	n+4	(n+1)S+2N+1I	Loading n registers including the PC, n > 0.
LDM	5	2S+1I+2N	Load PC.
STM	2	2N	Storing 1 register.
STM	n+1	(n-1)S+2N	Storing n registers, n > 1.
SWP	3	3N	Normal case.
SWP	4	2N+1I+1S	Single-cycle interlock.
B, BL, BX, BLX, BXJ	3	2S+1N	All cases.
SWI, Undefined	3	2S+1N	All cases.
Coprocessor absent	b+4	2S+1N+1I+bI	All cases.
CDP	b+2	1S+(b+1)I	All cases.

Table 9-2 ARM instruction cycle counts and bus activity (continued)

Instruction	Cycles	Memory bus	Comment
LDC	b+3	(b+1)I+2N	Loading 1 register.
STC	b+4	(b+2)I+1S+1N	Storing 1 register.
LDC	b+n+2	(b+1)I+(n-1)S+2N	Loading m registers, m>1.
STC	b+n+3	1N+nS+(b+2)I	Storing m registers, m>1.
MCR	b+2	1S+(b+1)I	All cases.
MCRR	b+3	1S+(b+2)I	All cases.
MRC	b+2	1S+(b+1)I	Normal case.
MRC	b+3	1S+(b+2)I	Following instruction uses transferred data.
MRRC	b+3	1S+(b+2)I	Normal case.
MRRC	b+4	1S+(b+3)I	Following instruction uses last transferred data.
MRS	2	1S+1I	All cases.
MSR	1	1S	If only flags are updated (mask_f).
MSR	3	1S+2I	If any bits other than just the flags are updated (all masks other than mask_f).
MUL, MLA	2	1S+1I	Normal case.
MUL, MLA	3	1S+2I	Following instruction uses the result in its first Execute cycle or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.
MULS, MLAS	4	1S+3I	All cases, sets flags.
QADD, QDADD,QSUB, QDSUB	1	1S	Normal case.
QADD, QDADD,QSUB, QDSUB	2	1S+1I	Following instruction uses the result in its first Execute cycle.
SMULL, UMULL,SMLAL, UMLAL	3	1S+2I	Normal case.
SMULL, UMULL,SMLAL, UMLAL	4	1S+3I	Following instruction uses RdHi result in its first Execute cycle or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.

Table 9-2 ARM instruction cycle counts and bus activity (continued)

Instruction	Cycles	Memory bus	Comment
SMULLS, UMULLS, SMLALS, UMLALS	5	1S+4I	All cases, sets flags.
SMULxy, SMLAxy	1	1S	Normal case.
SMULxy, SMLAxy	2	1S+1I	Following instruction uses the result in its first Execute or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.
SMULWx, SMLAWx	1	1S	Normal case.
SMULWx, SMLAWx	2	1S+1I	Following instruction uses the result in its first Execute or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.
SMLALxy	2	1S+1I	Normal case.
SMLALxy	3	1S+2I	Following instruction uses RdHi result in its first Execute cycle or its first Memory cycle. Does not apply to a multiply accumulate using result for accumulate operand.

9.2   Branch and ARM branch with link

Any ARM or Thumb branch, and an ARM branch with link operation takes three cycles as follows:

1.   During the first cycle, a branch instruction calculates the branch destination while performing a prefetch from the current PC. This prefetch is performed in all case, because by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch. If the previous instruction requested a data memory access, the data is transferred in this cycle.

2.   During the second cycle, the ARM7EJ-S processor performs a fetch from the branch destination. If the link bit is set, the return address to be stored in r14 is calculated.

3.   During the third cycle, the ARM7EJ-S processor performs a fetch from the destination + i, refilling the instruction pipeline.

Table 9-3 shows the cycle timings for branches and ARM branch with link operations.

Table 9-3 Cycle timings for branch and ARM branch with link

Cycle	ADDR	RDATA	TRANS
1	pc'	(pc+2i)	N cycle
2	pc'+i	(pc')	S cycle
3	pc'+2i	(pc'+i)	S cycle
		(pc'+2i)	

9.3 Thumb branch with link

A Thumb Branch with Link (BL) operation comprises two consecutive Thumb instructions, and takes four cycles as follows:

1. The first instruction acts as a simple data operation. The ARM7EJ-S processor takes a single cycle to add the PC to the upper part of the offset and store the result in r14. If the previous instruction requested a data memory access, the data is transferred in this cycle.

2. The second instruction acts similarly to the ARM BL instruction over three cycles:

    a. During the first cycle, the ARM7EJ-S processor calculates the final branch target address while performing a prefetch from the current PC.

    b. During the second cycle, the ARM7EJ-S processor performs a fetch from the branch destination, while calculating the return address to be stored in r14.

    c. During the third cycle, the ARM7EJ-S processor performs a fetch from the destination + 2, refilling the instruction pipeline.

Table 9-4 shows the cycle timings of the complete operation.

Table 9-4 Cycle timings for Thumb branch with link

Cycle	ADDR	RDATA	TRANS
1	pc+3i	(pc+i)	S cycle
2	pc'	(pc+3i)	N cycle
3	pc'+i	(pc')	S cycle
4	pc'+i	(pc'+i)	S cycle
		(pc'+i)	

9.4   Branch and Exchange

Branch and Exchange operations comprise the following:

*   Branch and Exchange (`BX`)

*   Branch with Link and Exchange to the value of a specified register (`BLX <Rm>`)

*   ARM Branch with Link and Exchange to an immediate operand value (ARM `BLX <immediate>`)

*   Branch and Exchange to Jazelle (`BXJ`).

Execution of these instructions takes three cycles, and is similar to a branch, as follows:

1.   During the first cycle, the ARM7EJ-S processor extracts the branch destination and the new core state while performing a prefetch from the current PC. This prefetch is performed in all cases, because by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch. In the case of `BX` and `BLX <Rm>`, the branch destination new state comes from the register. For `BLX <immediate>` the destination is calculated as a PC offset. The state is always changed.

     If the previous instruction requested a memory access (and there is no interlock in the case of `BX` or `BLX <register>`), the data is transferred in this cycle.

2.   During the second cycle, the ARM7EJ-S processor performs a fetch from the branch destination, using the new instruction width, dependent on the state that has been selected. If the link bit is set, the return address to be stored in r14 is calculated.

3.   During the third cycle, the ARM7EJ-S processor performs a fetch from the destination +2 or +4 dependent on the new specified state, refilling the instruction pipeline.

Table 9-5 on page 9-11 shows the cycle timings, where:

i   Is the instruction width before the `BX` or `BLX` instruction.

i'   Is the instruction width after the `BX` or `BLX` instruction.

tj'   Is the state of the **CPTBIT** and **CPJBIT** signals after the BX or BLX instruction.

Table 9-5 Cycle timings for Branch and Exchange

Cycle	ADDR	RDATA	TRANS	CPTBIT, CPJBIT
1	pc'	(pc+2i)	N cycle	tj'
2	pc'+i'	(pc')	S cycle	tj'
3	pc'+2i'	(pc'+i')	S cycle	tj'
		(pc'+2i')		

9.5 Thumb Branch, Link, and Exchange immediate

A Thumb Branch, Link, and Exchange immediate (`BLX <immediate>`) operation is similar to a Thumb `BL` operation. It comprises two consecutive Thumb instructions, and takes four cycles:

1. The first instruction acts as a simple data operation. It takes a single cycle to add the PC to the upper part of the offset and store the result in r14. If the previous instruction requested a data memory access, the data is transferred in this cycle.

2. The second instruction acts similarly to the ARM `BLX` instruction:

   a. During the first cycle, the ARM7EJ-S processor calculates the final branch target address while performing a prefetch from the current PC.

   b. During the second cycle, the ARM7EJ-S processor performs a fetch from the branch destination, using the new instruction width, dependent on the state that has been selected. The return address to be stored in r14 is calculated.

   c. During the third cycle, the ARM7EJ-S processor performs a fetch from the destination + 4, refilling the instruction pipeline.

Table 9-6 shows the cycle timings of the complete operation.

Table 9-6 Cycle timings for Thumb Branch, Link, and Exchange

Cycle	ADDR	RDATA	TRANS	CPTBIT
1	pc+3i	(pc+2i)	S cycle	t
2	pc'	(pc+3i)	N cycle	t'
3	pc'+i	(pc')	S cycle	t'
4	pc'+2i	(pc'+i)	S cycle	t'
		(pc'+2i)		

9.6   Data operations

A normal data operation executes in a single execute cycle except where the shift is determined by the contents of a register. A normal data operation requires up to two operands, that are read from the register file onto the A and B buses.

The ALU combines the A bus operand with the (shifted) B bus operand according to the operation specified in the instruction. The ARM7EJ-S processor pipelines this result and writes it into the destination register, when required. Compare and test operations do not write a result as they only affect the status flags.

An instruction prefetch occurs at the same time as the data operation, and the PC is incremented.

When a register-specified shift is used, an additional execute cycle is needed to read the shifting register operand. The instruction prefetch occurs during this first cycle.

The PC can be one or more of the register operands. When the PC is the destination, the external bus activity is affected. When the ARM7EJ-S processor writes the result to the PC, the contents of the instruction pipeline are invalidated, and the ARM7EJ-S processor takes the address for the next instruction prefetch from the ALU rather than the incremented address. The ARM7EJ-S processor refills the instruction pipeline before any further instruction execution takes place. Exceptions are locked out while the pipeline is refilling.

——— **Note** ———

Shifted register with destination equal to PC is not possible in Thumb state.

Table 9-7 shows the data operation cycle timings.

Table 9-7 Cycle timings for data operations

Cycle		ADDR	RDATA	TRANS
Normal	1	pc+3i	(pc+2i)	S cycle
			(pc+3i)	
ADD, SUB, RSB, ADC, SBC, RSC, MOV operation, dest=pc	1	pc'	(pc+2i)	N cycle
	2	pc'+ i	(pc')	S cycle
	3	pc'+2i	(pc'+i)	S cycle
			(pc'+ 2i)	

Table 9-7 Cycle timings for data operations (continued)

Cycle		ADDR	RDATA	TRANS
shift(Rs)	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	S cycle
			(pc+3i)	
shift or AND, ORR, EOR, MVN operation, dest=pc	1	pc+3i	(pc+2i)	I cycle
	2	pc'	-	N cycle
	3	pc'+i	(pc')	S cycle
	4	pc'+2i	(pc'+i)	S cycle
			(pc'+2i)	

9.7   MRS

An `MRS` operation always takes two cycles, as follows:

1.   The first cycle enables any pending state changes to the PSR to be made.

2.   The second cycle passes the PSR register through the ALU so that it can be written to the destination register.

———— **Note** ————

The `MRS` instruction can only be executed when in ARM state.

Table 9-8 shows the `MRS` cycle timings.

Table 9-8 Cycle timings for MRS

Cycle	ADDR	RDATA	TRANS
1	pc+3i	(pc+2i)	I cycle
2	pc+3i	-	S cycle
		(pc+3i)	

9.8   MSR

An MSR operation takes one cycle to execute if it only updates the status flags of the CPSR, and three cycles if it updates other parts of the PSR.

——— **Note** ———
MSR instructions can only be executed in ARM state.

Table 9-9 shows the cycle timings for MSR operations.

Table 9-9 Cycle timings for MSR

Cycle		ADDR	RDATA	TRANS
MSR flags	1	pc+3i	(pc+2i)	S cycle
			(pc+3i)	
MSR other	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	I cycle
	3	pc+3i	-	S cycle
			(pc+3i)	

9.9   Multiply and multiply accumulate

The multiply instructions use special hardware that implements integer multiplication. All cycles except the last are internal.

During the first (Execute) stage of a multiply instruction, the multiplier and multiplicand operands are read onto the A and B buses, to which the multiplier unit is connected. The first stage of the multiplier performs Booth recoding and partial product summation, using 16 bits of the multiplier operand each cycle.

During the second (Memory) stage of a multiply instruction, the partial product result from the Execute stage is added with an optional accumulate term (read onto the C bus) and a possible feedback term from a previous multiply step for multiplications which require additional cycles.

———— **Note** ————

In Thumb state, only the `MULS` and `MLAS` operations are possible.

9.9.1   Interlocks

The multiply unit in the ARM7EJ-S processor operates in both the Execute and Memory stages of the pipeline. For this reason, the multiplier result is not available until the end of the Memory stage of the pipeline. If the following instruction requires the use of the multiplier result, then it must be interlocked so that the correct value is available. This applies to all instructions that require the multiply result for the first Execute cycle or first Memory cycle of the instruction, except for multiply accumulate instructions using the previous multiply result as the accumulator operand.

For example, the following sequence incurs a single-cycle interlock:

```
MULr0, r1, r2SUBr4, r0, r3
```

The following cycle also incurs a single-cycle interlock:

```
MLAr0, r1, r2, r3STRr0, [r8]
```

The following example does not incur an interlock:

```
MLAr0, r1, r2, r0MLAr0, r3, r4, r0
```

Table 9-10 shows the cycle timing for `MUL` and `MLA` instructions with and without interlocks.

Table 9-10 Cycle timing for MUL and MLA

Cycle		ADDR	RDATA	TRANS
Normal	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	S cycle
			(pc+3i)	
Interlock	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	I cycle
	3	pc+3i	-	S cycle
			(pc+3i)	

The `MULS` and `MLAS` instructions always take four cycles to execute, and cannot generate interlocks in following instructions.

Table 9-11 shows the cycle timing for `MULS` and `MLAS` instructions.

Table 9-11 Cycle timings for MULS and MLAS

Cycle	ADDR	RDATA	TRANS
1	pc+3i	(pc+2i)	I cycle
2	pc+3i	-	I cycle
3	pc+3i	-	I cycle
4	pc+3i	-	S cycle
		(pc+3i)	

Table 9-12 on page 9-19 shows the cycle timing for `SMULL`, `UMULL`, `SMLAL`, and `UMLAL` instructions with and without interlocks.

Table 9-12 Cycle timing for SMULL, UMULL, SMLAL, and UMLAL

Cycle		ADDR	RDATA	TRANS
Normal	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	I cycle
	3	pc+3i	-	S cycle
			(pc+3i)	
Interlock	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	I cycle
	3	pc+3i	-	I cycle
	4	pc+3i	-	S cycle
			(pc+3i)	

The SMULLS, UMULLS, SMLALS, and UMLALS instructions always take five cycles to execute, and cannot generate interlocks in following instructions.

Table 9-13 shows the cycle timing for the SMULLS, UMULLS, SMLALS, and UMLALS instructions.

Table 9-13 Cycle timings for SMULLS, UMULLS, SMLALS, and UMLALS

Cycle	ADDR	RDATA	TRANS
1	pc+3i	(pc+2i)	I cycle
2	pc+3i	-	I cycle
3	pc+3i	-	I cycle
4	pc+3i	-	I cycle
5	pc+3i	-	S cycle
		(pc+3i)	

Table 9-14 shows the cycle timings for SMULxy, SMLAxy, SMULWy, and SMLAWy instructions with and without interlocks.

Table 9-14 Cycle timings for SMULxy, SMLAxy, SMULWy, and SMLAWy

Cycle		ADDR	RDATA	TRANS
Normal	1	pc+3i	(pc+2i)	S cycle
		b	(pc+3i)	b
Interlock	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	S cycle
			(pc+3i)	

Table 9-15 shows the cycle timing for SMLALxy instructions with and without interlocks.

Table 9-15 Cycle timings for SMLALxy

Cycle		ADDR	RDATA	TRANS
Normal	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	S cycle
			(pc+3i)	
Interlock	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	I cycle
	3	pc+3i	-	S cycle
			(pc+3i)	

9.10   QADD, QDADD, QSUB, and QDSUB

The instructions in this class usually take one cycle to execute, and are only available in ARM state.

9.10.1   Interlocks

The instructions in this class use both the Execute and Memory stages of the pipeline. For this reason, the result of an instruction in this class is not available until the end of the Memory stage of the pipeline. If a following instruction requires the use of the result, then it must be interlocked so that the correct value is available. This applies to all instructions that require the result for the first Execute cycle. Instructions that require the result of a QADD or similar instruction for the first Memory cycle do not incur an interlock.

As an example, the following sequence incurs a single-cycle interlock:

QADD r0, r1, r2SUB r4, r0, r3

The following cycle does not incur a single-cycle interlock:

QDSUB r0, r1, r2STR r0, [r8]

The following example does not incur an interlock:

QADD r0, r4, r5MLA r0, r3, r4, r0

Table 9-16 shows the cycle timings for QADD, QDADD, QSUB, and QDSUB instructions with and without interlocks.

Table 9-16 Cycle timings for QADD, QDADD, QSUB, and QDSUB

Cycle		ADDR	RDATA	TRANS
Normal	1	pc+3i	(pc+2i)	S cycle
			(pc+3i)	b
Interlock	1	pc+3i	(pc+2i)	I cycle
	2	pc+3i	-	S cycle
			(pc+3i)	

9.11   Load register

A load register operation can take a variable number of cycles. There might be a number of cycles before the loaded value is available for later instructions.

───── **Note** ─────

Destination equal to PC is not possible in Thumb state.

9.11.1   Interlocks

Unaligned word loads, load byte (LDRB), and load halfword (LDRH) instructions use the byte rotate unit in the Write stage of the pipeline. This introduces a single-cycle load-use interlock, that can affect the two instructions immediately following the load instruction.

The following example incurs a single-cycle interlock:

```
LDRB r0, [r1, #1]ADD r2, r0, r3ORR r4, r4, r5
```

The following example incurs a single-cycle interlock:

```
LDRB r0, [r1, #1]ORR r4, r4, r5ADD r2, r0, r3
```

When an interlock has been incurred for one instruction it does not have to be incurred for a later instruction. For example, the following sequence incurs a single-cycle interlock on the first ADD instruction, but the second ADD does not incur any interlocks:

```
LDRB r0, [r1, #1]ADD r2, r0, r3ADD r4, r0, r5
```

A single-cycle interlock refers to the number of unwaited clock cycles to which the interlock applies. If a multi-cycle instruction separates a load instruction and the instruction using the result of the load, then no interlock can apply. The following example does not incur an interlock:

```
LDRB r0, [r1]MUL r6, r7, r8ADD r4, r0, r5
```

Table 9-17 on page 9-23 shows the cycle timing for basic load register operations.

Table 9-17 Cycle timings for basic load register operations

Cycle		ADDR	RDATA	TRANS
Normal case	1	da	(pc+2i)	N cycle
	2	pc+3i	(da)	N cycle
			(pc+3i)	
Scaled offset	1	pc+3i	(pc+2i)	I cycle
	2	da	-	N cycle
	3	pc+3i	(da)	N cycle
			(pc+3i)	
dest=pc	1	da	(pc+2i)	N cycle
	2	pc+3i	(da)	I cycle
	3	pc'	-	N cycle
	4	pc'+i	(pc')	S cycle
	5	pc'+2i	(pc'+i)	S cycle
			(pc'+2i)	
Scaled offset dest=pc	1	pc+3i	(pc+2i)	I cycle
	2	da	-	N cycle
	3	pc+3i	(da)	I cycle
	4	pc'	-	N cycle
	5	pc'+i	(pc')	S cycle
	6	pc'+2i	(pc'+i)	S cycle
			(pc'+2i)	

Table 9-18 on page 9-24 shows the cycle timing for load operations resulting in simple interlocks.

Table 9-18 Cycle timings for load operations resulting in simple interlocks

Cycle		ADDR	RDATA	TRANS
Single-cycle interlock	1	da	(pc+2i)	N cycle
	2	pc+3i	(da)	I cycle
	3	pc+3i	-	N cycle
			(pc+3i)	

With more complicated interlock cases you cannot consider the load instruction in isolation. This is because in these cases the load instruction has vacated the Execute stage of the pipeline and a later instruction has occupied it.

Table 9-19 shows the one-cycle interlock incurred for the following sequence of instructions:

```
LDRB r0, [r1]ADD r6, r6, r3 ADD r2, r0, r1
```

Table 9-19 Cycle timings for an example LDRB, ADD and ADD sequence

Cycle		ADDR	RDATA	TRANS
LDRB r0, [r1]	1	da	(pc+2i)	N cycle
	2	pc+3i	(da)	N cycle
ADD r6, r6, r3	3	pc+4i	(pc+3i)	I cycle
	4	pc+4i	-	S cycle
ADD r2, r0, r1	5	pc+5i	(pc+4i)	S cycle
			(pc+5i)	

Table 9-20 on page 9-25 shows thecycle timing for the following code sequence:

```
LDRB r0, [r2]STMIA r3, {r0-r1}
```

Table 9-20 Cycle timings for an example LDRB and STMIA sequence

Cycle		ADDR	RDATA	TRANS	WDATA
LDRB r0, [r2]	1	da	(pc+2i)	N cycle	
	2	pc+3i	(da)	N cycle	
STMIA r3, {r0-r1}	3	pc+4i	(pc+3i)	I cycle	
	4	r3	-	N cycle	
	5	r3+4	-	S cycle	r0
	6	pc+4i	-	N cycle	r1
			(pc+4i)		

9.12   Store register

A store register operation takes two or more cycles. Duringthe last Execute cycle, the store address is calculated, and the data to be stored is read onto the C bus.

Table 9-21 shows the cycle timing for a store register operation.

Table 9-21 Cycle timings for a store register operation

Cycle		ADDR	RDATA	TRANS	WDATA
Normal case	1	da	(pc+2i)	N cycle	
	2	pc+3i	-	N cycle	Rd
			(pc+3i)		
Scaled offset	1	pc+3i	(pc+2i)	I cycle	
	2	da	-	N cycle	
	3	pc+3i	-	N cycle	Rd
			pc+3i		

 ARM DDI 0214B

9.13   Load multiple registers

A load multiple (LDM) instruction takes several cycles to execute, depending on the number of registers transferred and whether the PC is in the list of registers transferred:

1.   During the first cycle, the ARM7EJ-S processor calculates the address of the first word to be transferred, while performing an instruction prefetch.

2.   During the second and subsequent cycles, the ARM7EJ-S processor reads the data requested in the previous cycle and calculates the address of the next word to be transferred. The new value for the base register is calculated.

When a Data Abort occurs, the instruction continues to completion. The ARM7EJ-S processor prevents all register writing after the abort. The ARM7EJ-S processor restores the modified base pointer (which the load activity before the abort occurred might have overwritten).

When the PC is in the list of registers to be loaded, the ARM7EJ-S processor invalidates the current contents of the instruction pipeline. The PC is always the last register to be loaded, so an abort at any point prevents the PC from being overwritten.

——— **Note** ———

LDM with destination = PC cannot be executed in Thumb state. However, POP{Rlist, PC} equates to an LDM with destination = PC.

———————

Table 9-22 shows the LDM cycle timings.

Table 9-22 Cycle timings for LDM

Cycle		ADDR	RDATA	TRANS
1 register (not PC)				
	1	da	(pc+2i)	N cycle
	2	pc+3i	(da)	N cycle
			(pc+3i)	
n registers (n > 1) (not PC)				
	1	da	(pc+2i)	N cycle
	2	da++	(da)	S cycle
	.	da++	(da++)	S cycle
	n	da++	(da++)	S cycle

Table 9-22 Cycle timings for LDM (continued)

Cycle		ADDR	RDATA	TRANS
	n+1	pc+3i	(da++)	N cycle
			(pc+3i)	
1 register, dest=pc				
	1	da	(pc+2i)	N cycle
	2	pc+3i	(da)	I cycle
	3	pc'	-	N cycle
	4	pc'+i	(pc')	S cycle
	5	pc'+2i	(pc'+i)	S cycle
			(pc'+2i)	
n registers (n >1) (incl pc)				
	1	da	(pc+2i)	N cycle
	2	da++	(da)	S cycle
	.	da++	(da++)	S cycle
	n	da++	(da++)	S cycle
	n + 1	pc+3i	(da++)	I cycle
	n + 2	pc'	-	N cycle
	n + 3	pc'+i	(pc')	S cycle
	n + 4	pc'+2i	(pc'+i)	S cycle
			(pc'+2i)	

9.14   Store multiple registers

Store multiple (STM) instructions proceed in a similar way to load multiple instructions:

1.   During the first cycle, the ARM7EJ-S processor calculates the address of the first word to be transferred, while performing an instruction prefetch and also calculating the new value for the base register.

2.   During the second and subsequent cycles, ARM7EJ-S processor stores the data requested in the previous cycle and calculates the address of the next word to be transferred.

When a Data Abort occurs, the instruction continues to completion. The ARM7EJ-S processor restores the modified base pointer (which the load activity before the abort occurred might have overwritten).

Table 9-23 shows the STM cycle timings.

Table 9-23 Cycle timings for STM

Cycle		ADDR	RDATA	TRANS	WDATA
1 register	1	da	(pc+2i)	N cycle	
	2	pc+3i	-	N cycle	R
			(pc+3i)		-
n registers (n > 1)	1	da	(pc+2i)	N cycle	
	2	da++	-	S cycle	R
	.	da++	-	S cycle	R'
	n	da++		S cycle	R"
	n+1	pc+3i	-	N cycle	R'''
			(pc+3i)		

9.15   Load double register

The LDRD instruction behaves in the same way as an LDM of two registers. For more details, see *Load multiple registers* on page 9-27.

9.16 Store double register

The STRD instruction behaves in the same way as an STM of two registers. For more details, see *Store multiple registers* on page 9-29 and the appropriate entries in Table 9-23 on page 9-29.

9.17   Data swap

A data swap is similar to a back-to-back load and store instruction. The data is read from external memory in the second cycle and the contents of the register are written to the external memory in the third cycle (which is merged with the first Execute cycle of the next instruction).

The data swapped can be a byte or word quantity.

The swap operation might be aborted in either the read or the write cycle. An aborted swap operation does not affect the destination register.

—— **Note** ——

Data swap instructions are not available in Thumb state.

The **LOCK** output of the ARM7EJ-S processor is driven HIGH for both read and write cycles, to indicate to the memory system that it is an atomic operation.

9.17.1   Interlocks

A swap operation can cause single-cycle interlocks in a similar way to a load register instruction.

Table 9-24 shows the cycle timing for the basic data swap operation.

Table 9-24 Cycle timings for a basic data swap operation

Cycle		ADDR	RDATA	TRANS	LOCK	WDATA
Normal	1	da	(pc+2i)	N cycle	1	
	2	da	(da)	N	1	
	3	pc+3i	-	N cycle	0	Rd
			(pc+3i)			
1-cycle interlock	1	da	(pc+2i)	N cycle	1	
	2	da	(da)	N cycle	1	-
	3	pc+3i	-	I cycle	0	Rd
	4	pc+3i	-	S cycle	0	-
			(pc+3i)			-

Copyright © 2001 ARM Limited. All rights reserved.

9.18   Software interrupt, undefined instruction, and exception entry

Exceptions, software interrupts (SWIs), and undefined instructions force the PC to a specific value and refill the instruction pipeline from that address, as follows:

1.   During the first cycle, the ARM7EJ-S processor constructs the forced address, and a mode change might take place.

2.   During the second cycle:

   a.   The ARM7EJ-S processor performs a fetch from the exception address.

   b.   The return address to be stored in r14 is calculated.

   c.   The state of the CPSR is saved in the relevant SPSR.

3.   During the third cycle, the ARM7EJ-S processor performs a fetch from the exception address + 4, refilling the instruction pipeline.

The exception entry cycle timings are show in Table 9-25, where:

pc   Is one of:

   •   the address of the SWI instruction for SWIs

   •   the address of the instruction following the last one to be executed before entering the exception for interrupts

   •   the address of the aborted instruction for Prefetch Aborts

   •   the address of the instruction following the one that attempted the aborted data transfer for Data Aborts.

Xn   Is the appropriate exception address.

Table 9-25 Exception entry cycle timings

Cycle	ADDR	RDATA	TRANS	CPTBIT
1	Xn		N cycle	0
2	Xn+4	(Xn)	S cycle	0
3	Xn+8	(Xn+4)	S cycle	0
		(Xn+8)		

——— **Note** ———

The value on the **RDATA** bus can be unpredictable in the case of Prefetch Abort or Data Abort entry.

9.19 Coprocessor data processing operation

A coprocessor data (CDP) operation is a request from the ARM7EJ-S processor for the coprocessor to initiate some action. The coprocessor does not have to complete the action immediately, but the coprocessor must commit to completion before driving **CHSD** or **CHSE** to LAST.

If the coprocessor cannot perform the requested task, it leaves **CHSD** at ABSENT.

When the coprocessor is able to perform the task, but cannot commit immediately, it drives **CHSD** to WAIT, and in subsequent cycles drives **CHSE** to WAIT until able to commit. When it is able to commit, it drives **CHSE** to LAST.

An interrupt can cause the ARM7EJ-S processor to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 5-16).

——— **Note** ———

Coprocessor operations are only available in ARM state.

Table 9-26 shows the cycle timings for coprocessor data operations. In Table 9-26, **LC** represents the signal **CPLATECANCEL**.

Table 9-26 Cycle timings for coprocessor data operations

Cycle		ADDR	RDATA	TRANS	CPINSTR	CPPPASS	LC	CHSD	CHSE
Ready					(pc)				
	1	pc+3i	(pc+2i)	I cycle	(pc+i)	0	0	LAST	
	2	pc+3i	-	S cycle	(pc+2i)	1	0		-
			(pc+3i)		(pc+2i)				
Not ready					(pc)				
	1	pc+3i	(pc+2i)	I cycle	(pc+i)	0	0	WAIT	
		pc+3i	-	I cycle	(pc+2i)	1	0		WAIT
	n+1	pc+3i	-	I cycle	(pc+2i)	1	0		LAST
	n+2	pc+3i	-	S cycle	(pc+2i)	1	0		-
			(pc+3i)		(pc+2i)				

9.20   Load coprocessor register (from memory)

The load coprocessor (LDC) operation transfers one or more words of data from memory to a coprocessor.

The coprocessor commits to the transfer only when it is ready to accept the data. The coprocessor indicates that it is ready for the transfer to commence by driving **CHSD** or **CHSE** to GO. The ARM7EJ-S processor produces addresses and requests data memory reads on behalf of the coprocessor, which is expected to accept the data at sequential rates.

The coprocessor is responsible for determining the number of words to be transferred. It indicates this by setting **CHSD** or **CHSE** to LAST in the cycle before it is ready to initiate the transfer of the last data word.

An interrupt can cause the ARM7EJ-S processor to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 5-16).

——— **Note** ———
Coprocessor operations are only available in ARM state.

The load coprocessor register cycle timings are shown in Table 9-27, where:

INSTR   Indicates the signal **CPINSTR**.

OUT   Indicates the signal **CPDOUT**.

P   Indicates the signal **CPPASS**.

LC   Indicates the signal **CPLATECANCEL**.

D   Indicates the signal **CHSD**.

E   Indicates the signal **CHSE**.

N   Indicates the signal **CLKEN**.

Table 9-27 Cycle timings for load coprocessor register operations

Cycle		ADDR	RDATA	TRANS	INSTR	OUT	P	LC	D	E	N
1 register, ready					(pc)						1
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	LAST		1
	2	da	-	N cycle	(pc+2i)		1	0		-	1

Copyright © 2001 ARM Limited. All rights reserved.

Table 9-27 Cycle timings for load coprocessor register operations (continued)

Cycle	ADDR	RDATA	TRANS	INSTR	OUT	P	LC	D	E	N
3	pc+3i	(da)	N cycle	(pc+2i)		0	0		-	1
		(pc+3i)		(pc+2i)						0
				(pc+3i)	(da)					1
1 register, not ready				(pc)						
1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	WAIT		1
	pc+3i	-	I cycle	(pc+2i)		1	0		WAIT	1
n+1	pc+3i	-	I cycle	(pc+2i)		1	0		LAST	1
n+2	da	-	N cycle	(pc+2i)		1	0		-	1
n+3	pc+3i	(da)	N cycle	(pc+2i)		0	0		-	1
		(pc+3i)		(pc+2i)						0
				(pc+3i)	(da)					1
m registers (m > 1) ready				(pc)						
1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	GO		1
2	da	-	N cycle	(pc+2i)		1	0		GO	1
3	da++	(da)	S cycle	(pc+2i)		1	0		GO	1
	da++	(da++)	S cycle	(pc+2i)	(da)	1	0		GO	1
m	da++	(da++)	S cycle	(pc+2i)	(da++)	1	0		LAST	1
m+1	da++	(da++)	S cycle	(pc+2i)	(da++)	1	0		-	1
m+2	pc+3i	(da++)	N cycle	(pc+2i)	(da++)	0	0		-	1
		(pc+3i)		(pc+2i)						0
				(pc+3i)	(da++)					1
m registers (m > 1) not ready				(pc)						
1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	WAIT		1
	pc+3i	-	I cycle	(pc+2i)		1	0		WAIT	1
n+1	pc+3i	-	I cycle	(pc+2i)		1	0		GO	1

Table 9-27 Cycle timings for load coprocessor register operations (continued)

Cycle		ADDR	RDATA	TRANS	INSTR	OUT	P	LC	D	E	N
	n+2	da	-	N cycle	(pc+2i)		1	0		GO	1
	n+3	da++	(da)	S cycle	(pc+2i)		1	0		GO	1
		da++	(da++)	S cycle	(pc+2i)	(da)	1	0		GO	1
	n+m	da++	(da++)	S cycle	(pc+2i)	(da++)	1	0		LAST	1
	n+m+1	da++	(da++)	N cycle	(pc+2i)	(da++)	1	0		-	1
	n+m+2	pc+3i	(da++)	N cycle	(pc+2i)	(da++)	0	0		-	1
			(pc+3i)		(pc+2i)						0
					(pc+3i)	(da++)					1

Copyright © 2001 ARM Limited. All rights reserved.   ARM DDI 0214B

9.21   Store coprocessor register (to memory)

The store coprocessor (STC) operation transfers one or more words of data from a coprocessor to memory.

The coprocessor commits to the transfer only when it is ready to write the data. The coprocessor indicates that it is ready for the transfer to commence by driving **CHSD** or **CHSE** to GO. The ARM7EJ-S processor produces addresses and requests data memory writes on behalf of the coprocessor, which is expected to produce the data at sequential rates.

The coprocessor is responsible for determining the number of words to be transferred. It indicates this by setting **CHSD** or **CHSE** to LAST in the cycle before it is ready to initiate the transfer of the last data word.

An interrupt can cause the ARM7EJ-S processor to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 5-16).

——— **Note** ———
Coprocessor operations are only available in ARM state.

The store coprocessor register cycle timings are shown in Table 9-28, where:

INSTR	Indicates the signal **CPINSTR**.
IN	Indicates the signal **CPDIN**.
P	Indicates the signal **CPPASS**.
LC	Indicates the signal **CPLATECANCEL**.
D	Indicates the signal **CHSD**.
E	Indicates the signal **CHSE**.

Table 9-28 Cycle timings for STC

Cycle	ADDR	RDATA	TRANS	WDATA	INSTR	IN	P	LC	D	E
1 register, ready					(pc)					
1	pc+3i	(pc+2i)	I cycle		(pc+i)		0	0	LAST	-
2	da	-	I cycle		(pc+2i)		1	0		-
3	da	-	S cycle		(pc+2i)	CPdata1	0	0		-

Copyright © 2001 ARM Limited. All rights reserved.

Table 9-28 Cycle timings for STC (continued)

Cycle	ADDR	RDATA	TRANS	WDATA	INSTR	IN	P	LC	D	E
4	pc+3i	-	N cycle	CPdata1	(pc+2i)		0	0		-
		(pc+3i)			(pc+2i)					
1 register, not ready					(pc)					
1	pc+3i	(pc+2i)	I cycle		(pc+i)		0	0	WAIT	
.	pc+3i	-	I cycle		(pc+2i)		1	0		WAIT
n+1	da	-	I cycle		(pc+2i)		1	0		LAST
n+2	da	-	I cycle		(pc+2i)		1	0		-
n+3	da	-	S cycle		(pc+2i)	CPdata1	0	0		-
n+4	pc+3i	-	N cycle	CPdata1	(pc+2i)		0	0		-
		(pc+3i)			(pc+2i)					
m registers (m > 1), ready					(pc)					
1	pc+3i	(pc+2i)	I cycle		(pc+i)		0	0	GO	
2	da	-	I cycle		(pc+2i)		1	0		GO
3	da	-	S cycle		(pc+2i)	data(1)	1	0		GO
4	da++	-	S cycle	data(1)	(pc+2i)	data(2)	1	0		GO
.	da++	-	S cycle	data(m-4)	(pc+2i)	data(m-3)	1	0		GO
m	da++	-	S cycle	data(m-3)	(pc+2i)	data(m-2)	1	0		LAST
m+1	da++	-	S cycle	data(m-2)	(pc+2i)	data(m-1)	1	0		-
m+2	da++	-	S cycle	data(m-1)	(pc+2i)	data(m)	0	0		-
m+3	pc+3i	-	N cycle	data(m)	(pc+2i)		0	0		-
		(pc+3i)			(pc+2i)					
m registers, (m > 1), not ready					(pc)					
1	pc+3i	(pc+2i)	I cycle		(pc+i)		0	0	WAIT	
.	pc+3i	-	I cycle		(pc+2i)		1	0		WAIT
n+1	da	-	I cycle		(pc+2i)		1	0		GO

Table 9-28 Cycle timings for STC (continued)

Cycle	ADDR	RDATA	TRANS	WDATA	INSTR	IN	P	LC	D	E
n+2	da	-	I cycle		(pc+2i)		1	0		GO
n+3	da	-	S cycle		(pc+2i)	data(1)	1	0		GO
n+4	da++	-	S cycle	data(1)	(pc+2i)	data(2)	1	0		GO
.	da++	-	S cycle	data(2)	(pc+2i)	data(m-3)	1	0		GO
n+m	da++	-	S cycle	data(m-3)	(pc+2i)	data(m-2)	1	0		LAST
n+m+1	da++	-	S cycle	data(m-2)	(pc+2i)	data(m-1)	1	0		-
n+m+2	da++	-	S cycle	data(m-1)	(pc+2i)	data(m)	0	0		-
n+m+3	pc+3i	-	N cycle	data(m)	(pc+2i)		0	0		-
		(pc+3i)			(pc+2i)					

9.22   Coprocessor register transfer (to ARM)

The move from coprocessor (MRC) operation transfers a single coprocessor register into the specified ARM register.

An interrupt can cause the ARM7EJ-S processor to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 5-16).

───── **Note** ─────

Coprocessor operations are only available in ARM state.

The MRC instruction cycle timings are shown in Table 9-29, where:

P   Indicates the signal **CPPASS**.

LC   Indicates the signal **CPLATECANCEL**.

Table 9-29 Cycle timings for MRC

Cycle		ADDR	RDATA	TRANS	CPINSTR	CPDIN	P	LC	CHSD	CHSE
Ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	LAST	
	2	pc+3i	-	S cycle	(pc+2i)		1	0		-
		(pc+3i)			(pc+2i)	CPdata				
Not ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	WAIT	
	.	pc+3i	-	I cycle	(pc+2i)		1	0		WAIT
	n+1	pc+3i	-	I cycle	(pc+2i)		1	0		LAST
	n+2	pc+3i	-	S cycle	(pc+2i)		1	0		-
		(pc+3i)			(pc+2i)	CPdata				

9.23   Coprocessor register transfer (from ARM)

The move to coprocessor (MCR) operation transfers a specified ARM register to a coprocessor register.

An interrupt can cause the ARM7EJ-S processor to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 5-16).

───── **Note** ─────

Coprocessor operations are only available in ARM state.

The MCR instruction cycle timings are shown in Table 9-30, where:

P   Indicates the signal **CPPASS**.

LC   Indicates the signal **CPLATECANCEL**.

Table 9-30 Cycle timings for MCR

Cycle		ADDR	RDATA	TRANS	CPINSTR	CPDOUT	P	LC	CHSD	CHSE
Ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	LAST	
	2	pc+3i	-	S cycle	(pc+2i)		1	0		-
			(pc+3i)		(pc+2i)					
					(pc+3i)	Rd				
Not ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0		WAIT
	.	pc+3i	-	I cycle	(pc+2i)		1	0		WAIT
	n+1	pc+3i	-	I cycle	(pc+2i)		1	0		LAST
	n+2	pc+3i	-	S cycle	(pc+2i)		1	0		-
			(pc+3i)		(pc+2i)					
					(pc+3i)	Rd				

9.24   Double coprocessor register transfer (to ARM)

The move double from coprocessor (MRRC) operation transfers two coprocessor registers into the specified ARM registers.

An interrupt can cause the ARM7EJ-S processor to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 5-16).

—— **Note** ——

Coprocessor operations are only available in ARM state.

The MRRC instruction cycle timings are shown in Table 9-31, where:

P   Indicates the signal **CPPASS**.

LC   Indicates the signal **CPLATECANCEL**.

Table 9-31 Cycle timings for MRRC

Cycle		ADDR	RDATA	TRANS	CPINSTR	CPDIN	P	LC	CHSD	CHSE
Ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	GO	
	2	pc+3i	-	I cycle	(pc+2i)		1	0		LAST
	3	pc+3i	-	S cycle	(pc+2i)	CPdata1	1	0		-
		(pc+3i)			(pc+2i)	CPdata2				
Not ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	WAIT	
	.	pc+3i	-	I cycle	(pc+2i)		1	0		WAIT
	n+1	pc+3i	-	I cycle	(pc+2i)		1	0		GO
	n+2	pc+3i	-	I cycle	(pc+2i)		1	0		LAST
	n+3	pc+3i	-	S cycle	(pc+2i)	Cpdata1	1	0		-
		(pc+3i)			(pc+2i)	CPdata2				

9.25 Double coprocessor register transfer (from ARM)

The move double to coprocessor (MCRR) operation transfers two specified ARM registers to a coprocessor.

An interrupt can cause the ARM7EJ-S processor to abandon a busy-waiting coprocessor instruction (see *Busy-waiting and interrupts* on page 5-16).

—— **Note** ——

Coprocessor operations are only available in ARM state.

The MCRR instruction cycle timings are shown in Table 9-32, where:

P   Indicates the signal **CPPASS**.

LC   Indicates the signal **CPLATECANCEL**.

Table 9-32 Cycle timings for MCRR

Cycle		ADDR	RDATA	TRANS	CPINSTR	CPDOUT	P	LC	CHSD	CHSE
ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	GO	
	2	pc+3i	-	I cycle	(pc+2i)		1	0		LAST
	3	pc+3i	-	S cycle	(pc+2i)		1	0		-
			(pc+3i)		(pc+2i)	Rd				
					(pc+3i)	Rn				
not ready					(pc)					
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	0	WAIT	
	.	pc+3i	-	I cycle	(pc+2i)		1	0		WAIT
	n+1	pc+3i	-	I cycle	(pc+2i)		1	0		GO
	n+2	pc+3i	-	I cycle	(pc+2i)		1	0		LAST
	n+3	pc+3i	-	S cycle	(pc+2i)		1	0		-
			(pc+3i)		(pc+2i)	Rd				
					(pc+3i)	Rn				

9.26   Coprocessor absent

If no coprocessor is able to process a coprocessor instruction, the instruction is treated as an undefined instruction. This enables software to emulate coprocessor instructions when no hardware coprocessor is present.

—— **Note** ——

By default, **CHSD** and **CHSE** must be driven to ABSENT unless the coprocessor instruction is being handled by a coprocessor. Coprocessor operations are only available in ARM state.

The cycle timings for coprocessor absent instructions are shown in Table 9-33, where:

P   Indicates the signal **CPPASS**.

LC   Indicates the signal **CPLATECANCEL**.

Table 9-33 Cycle timings for coprocessor absent

Cycle		ADDR	RDATA	TRANS	CPINSTR	P	LC	CHSD	CHSE
Coprocessor absent in decode					(pc)				
	1	pc+3i	(pc+2i)	I cycle	(pc+i)		0	ABSENT	
	2	pc+3i	-	I cycle	(pc+2i)	1	0		-
	3	0x4	-	N cycle	(pc+2i)	1	0		-
	4	0x8	(0x4)	S cycle	(pc+2i)	0	1		-
	5	0xC	(0x8)	S cycle	(0x4)	0	0		-
			(0xC)		(0x8)				
Coprocessor absent in execute					(pc)				
	1	pc+3i	(pc+2i)	I cycle	(pc+i)	0	0	WAIT	
	.	pc+3i	-	I cycle	(pc+2i)	1	0		WAIT
	n+1	pc+3i	-	I cycle	(pc+2i)	1	0		ABSENT
	n+2	pc+3i	-	I cycle	(pc+2i)	1	0		-
	n+3	0x4	-	N cycle	(pc+2i)	1	0		-

                           ARM DDI 0214B

Table 9-33 Cycle timings for coprocessor absent (continued)

Cycle		ADDR	RDATA	TRANS	CPINSTR	P	LC	CHSD	CHSE
	n+4	0x8	(0x4)	S cycle	(pc+2i)	0	1		-
	n+5	0xc	(0x8)	S cycle	(0x4)	0	0		-
			(0xC)		(0x8)				

9.27   Unexecuted instructions

When the condition code of any instruction is not met, the instruction is not executed. An unexecuted instruction takes one cycle.

Table 9-34 shows the instruction cycle timing for unexecuted instructions.

Table 9-34 Cycle timing for unexecuted instructions

Cycle	ADDR	RDATA	TRANS
1	pc + 3i	(pc + 2i)	S cycle
		(pc + 3i)	

Chapter 10
AC Parameters

This chapter gives the AC timing parameters of the ARM7EJ-S processor. It contains the following sections:

- *About the AC parameters* on page 10-2
- *Memory interface* on page 10-3
- *Coprocessor interface* on page 10-5
- *Exception and configuration* on page 10-7
- *Debug interface* on page 10-8
- *Interrupt sensitivity* on page 10-10
- *JTAG interface* on page 10-11
- *Boundary scan and debug logic output data* on page 10-13
- *ETM interface* on page 10-14
- *AC timing parameter definitions* on page 10-15.

10.1 About the AC parameters

This section provides information about the AC parameters.

Each group of parameters is provided with a figure showing the parameter relationships and a table listing the parameters in alphabetic order.

All of the parameters used in this chapter are shown in Table 10-9 on page 10-15 with relevant cross references.

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

Prefixes to signal groups and names are used as part of the timing parameter symbols. They are:

T_{ih} The percentage of one clock cycle from the rising clock edge that an input must be held for.

T_{is} The percentage of one clock cycle to the rising clock edge that an input must be setup for.

T_{oh} The percentage of one clock cycle from the rising clock edge that an output is held for.

T_{ov} The percentage of one clock cycle from the rising clock edge before an output becomes valid.

10.2   Memory interface

Memory interface timing parameters are shown in Figure 10-1. The timing parameters used in Figure 10-1 are shown in Table 10-1.
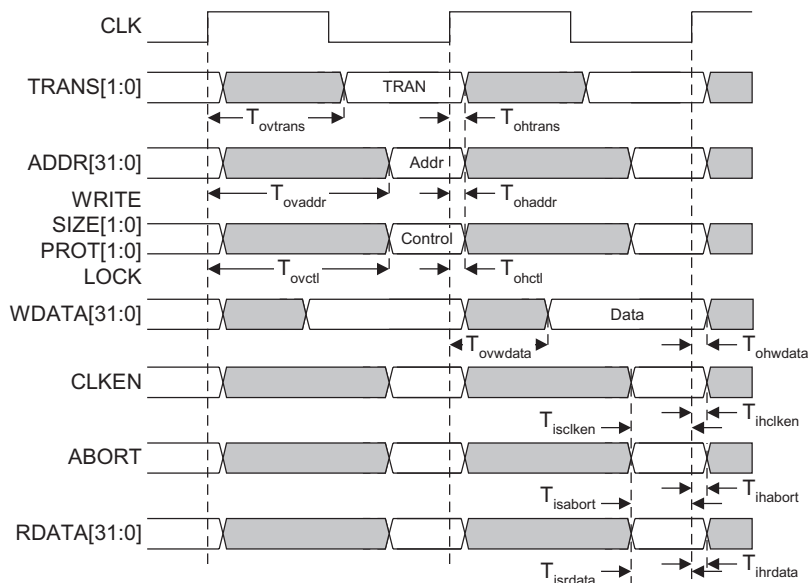

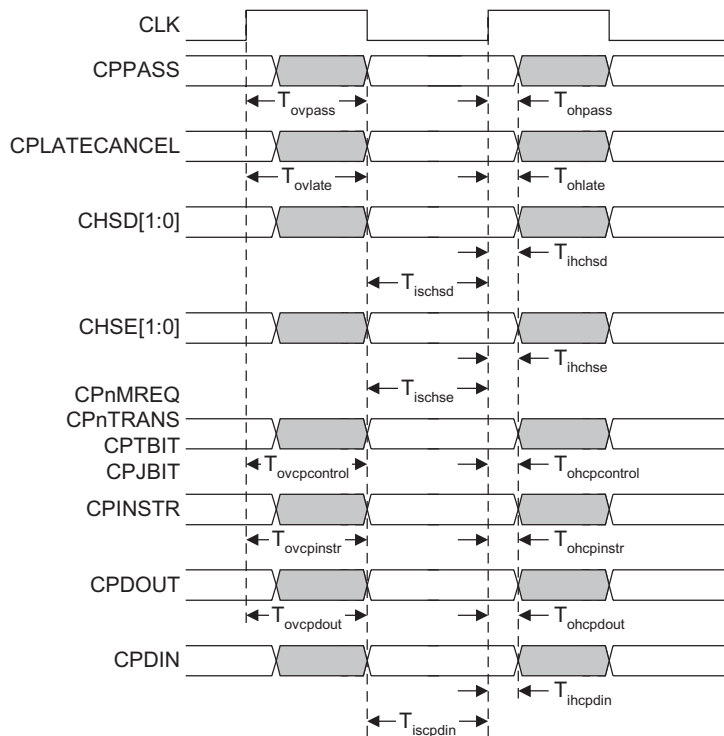**Figure 10-1 Memory interface timing**

Table 10-1 Memory interface timing parameters

Symbol	Parameter	Min	Max
$T_{ihabort}$	**ABORT** input hold from rising **CLK**	15%	-
$T_{ihclken}$	**CLKEN** input hold from rising **CLK**	15%	-
$T_{ihrdata}$	**RDATA** input hold from rising **CLK**	15%	-
$T_{isabort}$	**ABORT** input setup to rising **CLK**	25%	-
$T_{isclken}$	**CLKEN** input setup to rising **CLK**	40%	-
$T_{isrdata}$	**RDATA** input setup to rising **CLK**	25%	-

Copyright © 2001 ARM Limited. All rights reserved.

Table 10-1 Memory interface timing parameters (continued)

Symbol	Parameter	Min	Max
T_{ohaddr}	**ADDR** hold time from rising **CLK**	0%	-
T_{ohctl}	Control hold time from rising **CLK**	0%	-
$T_{ohtrans}$	Transaction type hold time from rising **CLK**	0%	-
$T_{ohwdata}$	**WDATA** hold time from rising **CLK**	0%	-
T_{ovaddr}	Rising **CLK** to **ADDR** valid	-	65%
T_{ovctl}	Rising **CLK** to control valid	-	65%
$T_{ovtrans}$	Rising **CLK** to transaction type valid	-	65%
$T_{ovwdata}$	Rising **CLK** to **WDATA** valid	-	60%

 ARM DDI 0214B

10.3   Coprocessor interface

Coprocessor interface timing parameters are shown in Figure 10-2. The timing parameters used in Figure 10-2 are shown in Table 10-2.

Figure 10-2 Coprocessor interface timing

Table 10-2 Coprocessor interface timing parameters

Symbol	Parameter	Min	Max
T_{ihchsd}	**CHSD** input hold from rising **CLK**	15%	-
T_{ihchse}	**CHSE** input hold from rising **CLK**	15%	-
$T_{ihcpdin}$	**CPDIN** input hold from rising **CLK**	15%	-
T_{ischsd}	**CHSD** input setup to rising **CLK**	35%	-

Table 10-2 Coprocessor interface timing parameters (continued)

Symbol	Parameter	Min	Max
T_{ischse}	**CHSE** input setup to rising **CLK**	35%	-
$T_{iscpdin}$	**CPDIN** input setup to rising **CLK**	35%	-
$T_{ohcpcontrol}$	**CPnMREQ**, **CPnTRANS**, **CPTBIT**, and **CPJBIT** hold from **rising CLK**	0%	-
$T_{ohcpdout}$	**CPDOUT** hold time from **CLK** rising	0%	-
$T_{ohcpinstr}$	**CPINSTR** hold time from **CLK** rising	0%	-
T_{ohlate}	**CPLATECANCEL** hold from **CLK** rising	0%	-
T_{ohpass}	**CPPASS** hold time from **CLK** rising	0%	-
$T_{ovcpcontrol}$	Rising **CLK** to **CPnMREQ**, **CPnTRANS**, **CPTBIT**, and **CPJBIT** valid	-	35%
$T_{ovcpdout}$	Rising **CLK** to **CPDOUT** valid	-	35%
$T_{ovcpinstr}$	Rising **CLK** to **CPINSTR** valid	-	35%
T_{ovlate}	Rising **CLK** to **CPLATECANCEL** valid	-	35%
T_{ovpass}	Rising **CLK** to **CPPASS** valid	-	35%

 ARM DDI 0214B

10.4 Exception and configuration

Exception and configuration timing parameters are shown in Figure 10-3. The timing parameters used in Figure 10-3 are shown in Table 10-3.
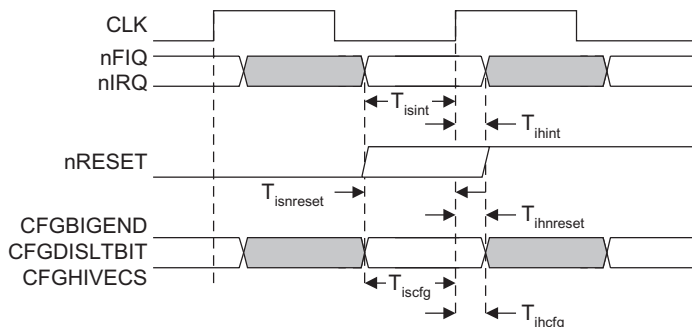
Figure 10-3 Exception and configuration timing

Table 10-3 Exception and configuration timing parameters

Symbol	Parameter	Min	Max
T_{ihcfg}	Configuration input hold from rising **CLK**	15%	-
T_{ihint}	Interrupt input hold from rising **CLK**	15%	-
$T_{ihnreset}$	**nRESET** input hold from rising **CLK**	15%	-
T_{iscfg}	Configuration input setup to rising **CLK**	30%	-
T_{isint}	Interrupt input setup to rising **CLK**	30%	-
$T_{isnreset}$	**nRESET** input setup to rising **CLK**	35%	-

10.5   Debug interface

Debug interface timing parameters are shown in Figure 10-4. The timing parameters used in Figure 10-4 are shown in Table 10-4.
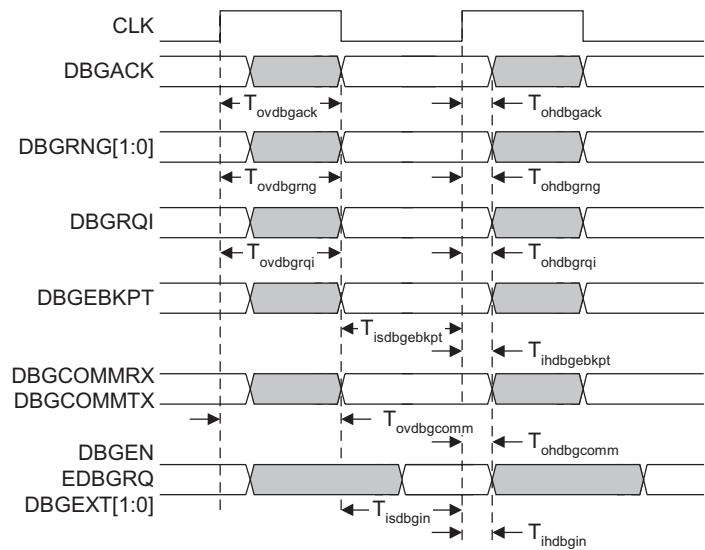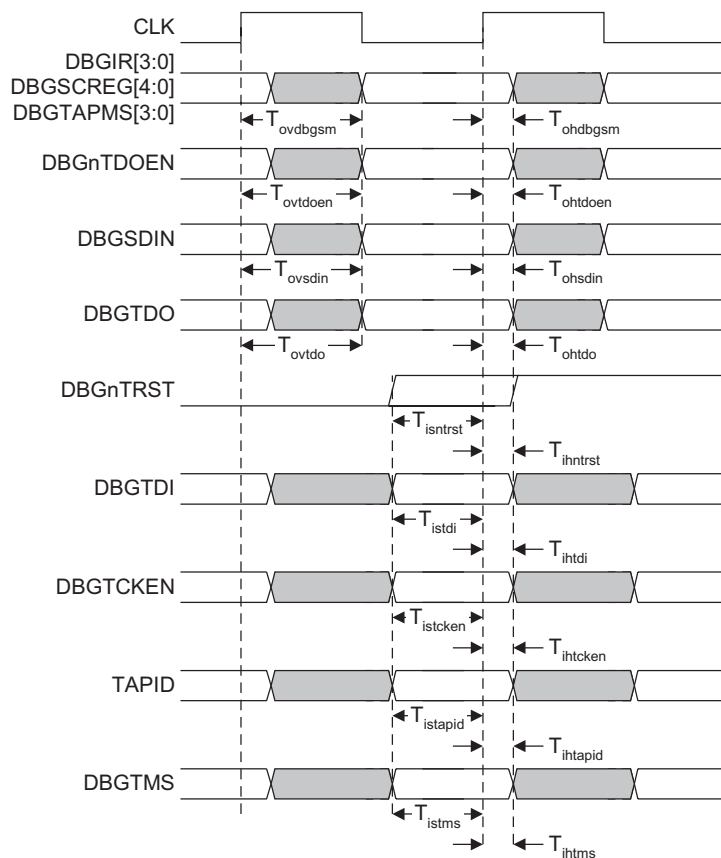

Figure 10-4 Debug interface timing

Table 10-4 Debug interface timing parameters

Symbol	Parameter	Min	Max
$T_{ihdbgebkpt}$	**DBGEBKPT** hold from rising **CLK**	15%	-
$T_{ihdbgin}$	Debug inputs input hold from rising **CLK**	15%	-
$T_{isdbgebkpt}$	**DBGEBKPT** setup to rising **CLK**	25%	-
$T_{isdbgin}$	Debug inputs input setup to rising **CLK**	25%	-
$T_{ohdbgack}$	**DBGACK** hold time from **rising CLK**	0%	-
$T_{ohdbgcomm}$	Communications channel output hold time from rising **CLK**	0%	-
$T_{ohdbgrng}$	**DBGRNG** hold time from **rising CLK**	0%	-
$T_{ohdbgrqi}$	**DBGRQI** hold time from **rising CLK**	0%	-

                     ARM DDI 0214B

Table 10-4 Debug interface timing parameters (continued)

Symbol	Parameter	Min	Max
$T_{ovdbgack}$	Rising **CLK** to **DBGACK** valid	-	60%
$T_{ovdbgcomm}$	Rising **CLK** to communications channel outputs valid	-	60%
$T_{ovdbgrng}$	Rising **CLK** to **DBGRNG** valid	-	80%
$T_{ovdbgrqi}$	Rising **CLK** to **DBGRQI** valid	-	45%

10.6   Interrupt sensitivity

Sensitivity to interrupt timing parameters are shown in Figure 10-5. The timing parameters used in Figure 10-5 are shown in Table 10-5.
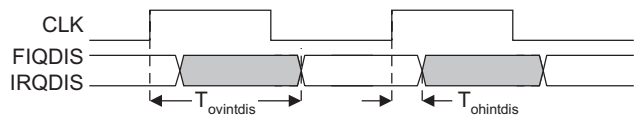
Figure 10-5 Interrupt sensitivity timing

Table 10-5 Interrupt sensitivity timing parameters

Symbol	Parameter	Min	Max
$T_{ohintdis}$	Sensitive to interrupt status hold from **CLK** rising	0%	-
$T_{ovintdis}$	Rising **CLK** to Sensitive to interrupt status valid	-	70%

 ARM DDI 0214B

10.7 JTAG interface

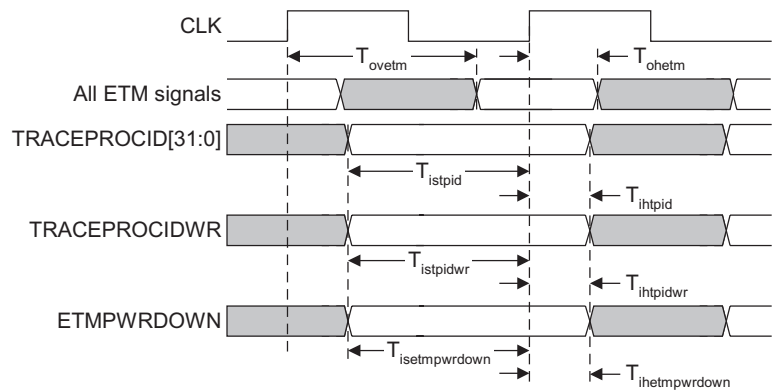JTAG interface timing parameters are shown in Figure 10-6. The timing parameters used in Figure 10-6 are shown in Table 10-6 on page 10-12.

Figure 10-6 JTAG interface timing

Table 10-6 JTAG interface timing parameters

Symbol	Parameter	Min	Max
$T_{ihntrst}$	**DBGnTRST** input hold from **rising CLK**	15%	-
$T_{ihtapid}$	**TAPID** input hold time from rising CLK	15%	-
$T_{ihtcken}$	**DBGTCKEN** input hold from rising CLK	15%	-
T_{ihtdi}	**DBGTDI** input hold from rising CLK	15%	-
T_{ihtms}	**DBGTMS** input hold from rising CLK	15%	-
$T_{isntrst}$	**DBGnTRST** input setup to rising CLK	25%	-
$T_{istapid}$	**TAPID** input setup to rising CLK	25%	-
$T_{istcken}$	**DBGTCKEN** input setup to rising CLK	35%	-
T_{istdi}	**DBGTDI** input setup to rising CLK	35%	-
T_{istms}	**DBGTMS** input setup to rising CLK	35%	-
$T_{ohdbgsm}$	Debug state hold from rising CLK	0%	-
T_{ohsdin}	**DBGSDIN** hold from rising CLK	0%	-
T_{ohtdo}	**DBGTDO** hold from rising CLK	0%	-
$T_{ohtdoen}$	**DBGnTDOEN** hold from rising CLK	0%	-
$T_{ovdbgsm}$	Rising **CLK** to debug state valid	-	50%
T_{ovsdin}	Rising **CLK** to **DBGSDIN** valid	-	50%
T_{ovtdo}	Rising **CLK** to **DBGTDO** valid	-	50%
$T_{ovtdoen}$	Rising **CLK** to **DBGnTDOEN** valid	-	60%

 ARM DDI 0214B

10.8   Boundary scan and debug logic output data

The timing relationship between **DBGSDOUT** and **DBGTDO** is shown in Figure 10-7. The timing parameters used in Figure 10-7 are shown in Table 10-7.
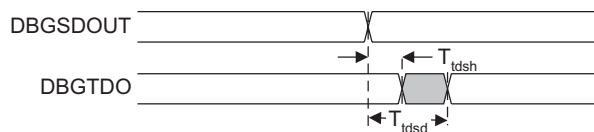
Figure 10-7 DBGSDOUT to DBGTDO relationship

Table 10-7 DBGSDOUT to DBGTDO relationship timing parameters

Symbol	Parameter	Min	Max
T_{tdsd}	**DBGTDO** delay from **DBGSDOUT** changing	-	-
T_{tdsh}	**DBGTDO** hold time from **DBGSDOUT** changing	-	-

10.9   ETM interface

ETM interface timing parameters are shown in Figure 10-8. The timing parameters used in Figure 10-8 are shown in Table 10-8.

Figure 10-8 ETM interface timing

Table 10-8 ETM interface timing parameters

Symbol	Parameter	Min	Max
$T_{ihetmpwrdown}$	**ETMPWRDOWN** input hold from rising **CLK**	15%	-
T_{ihtpid}	**TRACEPROCID[31:0]** input hold from rising **CLK**	15%	-
$T_{ihtpidwr}$	**TRACEPROCIDWR** input hold from rising **CLK**	15%	-
$T_{isetmpwrdown}$	**ETMPWRDOWN** input setup to rising **CLK**	25%	-
T_{istpid}	**TRACEPROCID[31:0]** input setup to rising **CLK**	25%	-
$T_{istpidwr}$	**TRACEPROCIDWR** input setup to rising **CLK**	25%	-
T_{ohetm}	ETM interface output signals hold from rising CLK	0%	-
T_{ovetm}	Rising **CLK** to ETM interface output signals valid	-	20%

10.10 AC timing parameter definitions

Table 10-9 shows target AC parameters. All figures are expressed as percentages of the **CLK** period at maximum operating frequency.

_____ **Note** _____

Where 0% is given, this indicates the hold time to clock edge plus the maximum clock skew for internal clock buffering.

Table 10-9 Target AC timing parameters

Symbol	Parameter	Min	Max	Cross reference
$T_{ihabort}$	**ABORT** input hold from rising **CLK**	15%	-	Figure 10-1 on page 10-3
T_{ihcfg}	Configuration input hold from rising **CLK**	15%	-	Figure 10-3 on page 10-7
T_{ihchsd}	**CHSD** input hold from rising **CLK**	15%	-	Figure 10-2 on page 10-5
T_{ihchse}	**CHSE** input hold from rising **CLK**	15%	-	Figure 10-2 on page 10-5
$T_{ihclken}$	**CLKEN** input hold from rising **CLK**	15%	-	Figure 10-1 on page 10-3
$T_{ihcpdin}$	**CPDIN** input hold from rising **CLK**	15%	-	Figure 10-2 on page 10-5
$T_{ihdbgebkpt}$	**DBGEBKPT** input hold from rising **CLK**	15%	-	Figure 10-4 on page 10-8
$T_{ihdbgin}$	Debug inputs input hold from rising **CLK**	15%	-	Figure 10-4 on page 10-8
$T_{ihetmpwrdown}$	**ETMPWRDOWN** input hold from rising **CLK**	15%	-	Figure 10-8 on page 10-14
T_{ihint}	Interrupt input hold from rising **CLK**	15%	-	Figure 10-3 on page 10-7
$T_{ihnreset}$	**nRESET** input hold from rising **CLK**	15%	-	Figure 10-3 on page 10-7
$T_{ihntrst}$	**DBGnTRST** input hold from rising CLK	15%	-	Figure 10-6 on page 10-11
$T_{ihrdata}$	**RDATA** input hold from rising **CLK**	15%	-	Figure 10-1 on page 10-3
$T_{ihtapid}$	**TAPID** input hold time from rising CLK	15%	-	Figure 10-6 on page 10-11
$T_{ihtcken}$	**DBGTCKEN** input hold from rising CLK	15%	-	Figure 10-6 on page 10-11
T_{ihtdi}	**DBGTDI** input hold from rising CLK	15%	-	Figure 10-6 on page 10-11
T_{ihtms}	**DBGTMS** input hold from rising CLK	15%	-	Figure 10-6 on page 10-11

Table 10-9 Target AC timing parameters (continued)

Symbol	Parameter	Min	Max	Cross reference
T_{ihtpid}	**TRACEPROCID[31:0]** input hold from rising **CLK**	15%	-	Figure 10-8 on page 10-14
$T_{ihtpidwr}$	**TRACEPROCIDWR** input hold from rising **CLK**	15%	-	Figure 10-8 on page 10-14
$T_{isabort}$	**ABORT** input setup to rising **CLK**	25%	-	Figure 10-1 on page 10-3
T_{iscfg}	Configuration input setup to rising **CLK**	30%	-	Figure 10-3 on page 10-7
T_{ischsd}	**CHSD** input setup to rising **CLK**	35%	-	Figure 10-2 on page 10-5
T_{ischse}	**CHSE** input setup to rising **CLK**	35%	-	Figure 10-2 on page 10-5
$T_{isclken}$	**CLKEN** input setup to rising **CLK**	40%	-	Figure 10-1 on page 10-3
$T_{iscpdin}$	**CPDIN** input setup to rising **CLK**	35%	-	Figure 10-2 on page 10-5
$T_{isdbgebkpt}$	**DBGEBKPT** input setup to rising **CLK**	25%	-	Figure 10-4 on page 10-8
$T_{isdbgin}$	Debug inputs input setup to rising **CLK**	25%	-	Figure 10-4 on page 10-8
$T_{isetmpwrdown}$	**ETMPWRDOWN** input setup to rising **CLK**	25%	-	Figure 10-8 on page 10-14
T_{isint}	Interrupt input setup to rising **CLK**	30%	-	Figure 10-3 on page 10-7
$T_{isnreset}$	**nRESET** input setup to rising **CLK**	35%	-	Figure 10-3 on page 10-7
$T_{isntrst}$	**DBGnTRST** input setup to rising CLK	25%	-	Figure 10-6 on page 10-11
$T_{isrdata}$	**RDATA** input setup to rising **CLK**	25%	-	Figure 10-1 on page 10-3
$T_{istapid}$	**TAPID** input setup to rising CLK	25%	-	Figure 10-6 on page 10-11
$T_{istcken}$	**DBGTCKEN** input setup to rising CLK	35%	-	Figure 10-6 on page 10-11
T_{istdi}	**DBGTDI** input setup to rising CLK	35%	-	Figure 10-6 on page 10-11
T_{istms}	**DBGTMS** input setup to rising CLK	35%	-	Figure 10-6 on page 10-11
T_{istpid}	**TRACEPROCID[31:0]** input setup time to rising **CLK**	25%	-	Figure 10-8 on page 10-14
$T_{istpidwr}$	**TRACEPROCIDWR** input setup time to rising **CLK**	25%	-	Figure 10-8 on page 10-14
T_{ohaddr}	**ADDR** hold time from rising **CLK**	0%	-	Figure 10-1 on page 10-3
$T_{ohcpcontrol}$	**CPnMREQ**, **CPnTRANS**, **CPTBIT**, and **CPJBIT** hold from rising CLK	0%	-	Figure 10-2 on page 10-5
$T_{ohcpdout}$	**CPDOUT** hold time from rising CLK	0%	-	Figure 10-2 on page 10-5

Table 10-9 Target AC timing parameters (continued)

Symbol	Parameter	Min	Max	Cross reference
$T_{ohcpinstr}$	**CPINSTR** hold time from rising CLK	0%	-	Figure 10-2 on page 10-5
T_{ohctl}	Control hold time from rising **CLK**	0%	-	Figure 10-1 on page 10-3
$T_{ohdbgack}$	**DBGACK** hold time from rising CLK	0%	-	Figure 10-4 on page 10-8
$T_{ohdbgcomm}$	Communications channel output hold time from rising **CLK**	0%	-	Figure 10-4 on page 10-8
$T_{ohdbgrng}$	**DBGRNG** hold time from rising CLK	0%	-	Figure 10-4 on page 10-8
$T_{ohdbgrqi}$	**DBGRQI** hold time from rising CLK	0%	-	Figure 10-4 on page 10-8
$T_{ohdbgsm}$	Debug state hold from rising CLK	0%	-	Figure 10-6 on page 10-11
T_{ohetm}	ETM interface output signals hold from rising CLK	0%	-	Figure 10-8 on page 10-14
$T_{ohintdis}$	Sensitive to interrupt status hold from rising CLK	0%	-	Figure 10-5 on page 10-10
T_{ohlate}	**CPLATECANCEL** hold from rising CLK	0%	-	Figure 10-2 on page 10-5
T_{ohpass}	**CPPASS** hold time from rising CLK	0%	-	Figure 10-2 on page 10-5
T_{ohsdin}	**DBGSDIN** hold from rising CLK	0%	-	Figure 10-6 on page 10-11
T_{ohtdo}	**DBGTDO** hold from rising CLK	0%	-	Figure 10-6 on page 10-11
$T_{ohtdoen}$	**DBGnTDOEN** hold from rising CLK	0%	-	Figure 10-6 on page 10-11
$T_{ohtrans}$	Transaction type hold time from rising **CLK**	0%	-	Figure 10-1 on page 10-3
$T_{ohwdata}$	**WDATA** hold time from rising **CLK**	0%	-	Figure 10-1 on page 10-3
T_{ovaddr}	Rising **CLK** to **ADDR** valid	-	65%	Figure 10-1 on page 10-3
$T_{ovcpcontrol}$	Rising **CLK** to **CPnMREQ**, **CPnTRANS**, **CPTBIT**, and **CPJBIT** valid	-	35%	Figure 10-2 on page 10-5
$T_{ovcpdout}$	Rising **CLK** to **CPDOUT** valid	-	35%	Figure 10-2 on page 10-5
$T_{ovcpinstr}$	Rising **CLK** to **CPINSTR** valid	-	35%	Figure 10-2 on page 10-5
T_{ovctl}	Rising **CLK** to control valid	-	65%	Figure 10-1 on page 10-3
$T_{ovdbgack}$	Rising **CLK** to **DBGACK** valid	-	60%	Figure 10-4 on page 10-8
$T_{ovdbgcomm}$	Rising **CLK** to communications channel outputs valid	-	60%	Figure 10-4 on page 10-8

Table 10-9 Target AC timing parameters (continued)

Symbol	Parameter	Min	Max	Cross reference
$T_{ovdbgrng}$	Rising **CLK** to **DBGRNG** valid	-	80%	Figure 10-4 on page 10-8
$T_{ovdbgrqi}$	Rising **CLK** to **DBGRQI** valid	-	45%	Figure 10-4 on page 10-8
$T_{ovdbgsm}$	Rising **CLK** to debug state valid	-	50%	Figure 10-6 on page 10-11
T_{ovetm}	Rising **CLK** to ETM interface output signals valid	-	20%	Figure 10-8 on page 10-14
$T_{ovintdis}$	Rising **CLK** to Sensitive to interrupt status valid	-	70%	Figure 10-5 on page 10-10
T_{ovlate}	Rising **CLK** to **CPLATECANCEL** valid	-	35%	Figure 10-2 on page 10-5
T_{ovpass}	Rising **CLK** to **CPPASS** valid	-	35%	Figure 10-2 on page 10-5
T_{ovsdin}	Rising **CLK** to **DBGSDIN** valid	-	50%	Figure 10-6 on page 10-11
T_{ovtdo}	Rising **CLK** to **DBGTDO** valid	-	50%	Figure 10-6 on page 10-11
$T_{ovtdoen}$	Rising **CLK** to **DBGnTDOEN** valid	-	60%	Figure 10-6 on page 10-11
$T_{ovtrans}$	Rising **CLK** to transaction type valid	-	65%	Figure 10-1 on page 10-3
$T_{ovwdata}$	Rising **CLK** to **WDATA** valid	-	60%	Figure 10-1 on page 10-3
T_{tdsd}	**DBGTDO** delay from **DBGSDOUT** changing	-	-	Figure 10-7 on page 10-13
T_{tdsh}	**DBGTDO** hold time from **DBGSDOUT** changing	-	-	Figure 10-7 on page 10-13

Appendix A
Signal Descriptions

This appendix lists and describes all of the ARM7EJ-S processor interface signals. It contains the following sections:

- *Clock interface signals* on page A-2
- *Memory interface signals* on page A-3
- *Interrupt signals* on page A-4
- *Miscellaneous signals* on page A-5
- *Coprocessor interface signals* on page A-6
- *Debug signals* on page A-8
- *ETM interface signals* on page A-10.

Copyright © 2001 ARM Limited. All rights reserved.

A.1   Clock interface signals

Clock interface signals are shown in Table A-1.

Table A-1 Clock interface signals

Name	Direction	Description
CLK	Input	System clock. This clock times all operations in the ARM7EJ-S processor. All outputs change from the rising edge and all inputs are sampled on the rising edge. The clock can be stretched in either the LOW or HIGH phase. Synchronous wait states can be added using the **CLKEN** signal. Through the use of the **DBGTCKEN** signal, this clock also times debug operations.
CLKEN	Input	Wait-state control. The ARM7EJ-S processor can be stalled for integer clock cycles by driving **CLKEN** LOW. This signal must be held HIGH at all other times.
CORECLKENOUT	Output	The principal state advance signal for the ARM7EJ-S processor.

A.2 Memory interface signals

Memory interface signals are shown in Table A-2.

Table A-2 Memory interface signals

Name	Direction	Description
ABORT	Input	The memory abort or bus error is issued by the memory system to signal to the processor that a requested access is disallowed.
ADDR[31:0]	Output	The processor address bus.
LOCK	Output	Locked transaction operation: • when HIGH, the processor is performing a locked memory access • when LOW, the arbiter can enable another device to access the memory.
PROT[1:0]	Output	These indicate whether the output is opcode or data and whether access is in User or Privileged mode.
RDATA[31:0]	Input	The read data input bus is used to transfer instructions and data between the processor and memory. The data on this bus is sampled by the processor at the end of the clock cycle during read accesses.
SIZE[1:0]	Output	Memory access width indicates to the external memory system when a word, halfword, or byte length transfer is required: • b00 indicates a byte transfer • b01 indicates a halfword transfer • b10 indicates a word transfer • b11 is reserved.
TRANS[1:0]	Output	The next transaction type outputs indicate the type of the next transaction (internal, coprocessor, sequential, or non-sequential): • b00 indicates an internal cycle • b01 indicates a coprocessor register transfer cycle • b10 indicates a nonsequential cycle • b11 indicates a sequential cycle.
WDATA[31:0]	Output	The write data output bus is used to transfer data from the processor to the memory or coprocessor system. Write data is set up to the end of the cycle of write accesses and remains valid throughout wait states.
WRITE	Output	Write/read access: • when HIGH indicates a processor write cycle • when LOW, indicates a processor read cycle.

A.3   Interrupt signals

Interrupt signals are shown in Table A-3.

Table A-3 Interrupt signals

Name	Direction	Description
FIQDIS	Output	FIQ disabled. When HIGH, indicates that the ARM7EJ-S processor ignores the state of **nFIQ**.
IRQDIS	Output	IRQ disabled. When HIGH, indicates that the ARM7EJ-S processor ignores the state of **nIRQ**.
nFIQ	Input	Not fast interrupt is a synchronous input to the core. It is not synchronized internally to the core.
nIRQ	Input	Not interrupt request is a synchronous input to the core. It is not synchronized internally to the core.

A.4   Miscellaneous signals

Miscellaneous signals are shown in Table A-4.

Table A-4 Miscellaneous signals

Name	Direction	Description
CFGBIGEND	Input	When HIGH, the ARM7EJ-S processor treats bytes in memory as being in big-endian format. When it is LOW, memory is treated as little-endian. This is a static configuration signal.
CFGDISLTBIT	Input	When HIGH, the ARM7EJ-S processor disables certain ARMv5TEJ defined behavior involving loading data to the PC. This input must be tied LOW for normal operation and full ARMv5TEJ compatibility. This is a static configuration signal.
CFGHIVECS	Input	When LOW, the ARM7EJ-S processor exception vectors start at address 0x0000 0000. When HIGH the ARM7EJ-S processor exception vectors start at address 0xFFFF 0000. This is a static configuration signal.
nRESET	Input	Not reset is used to start the processor from a known address. This is a level-sensitive asynchronous reset.

A.5 Coprocessor interface signals

Coprocessor interface signals are shown in Table A-5.

Table A-5 Coprocessor interface signals

Name	Direction	Description
CHSD[1:0]	Input	Coprocessor handshake decode. The handshake signals from the Decode stage of the pipeline follower of the coprocessor.
CHSE[1:0]	Input	Coprocessor handshake execute. The handshake signals from the Execute stage of the pipeline follower of the coprocessor.
CPCLKEN	Output	The coprocessor clock enable is the synchronous enable for the coprocessor pipeline follower. When HIGH on the rising edge of **CLK**, the pipeline follower logic is able to advance.
CPDIN[31:0]	Input	This is the 32-bit coprocessor write data bus for transferring data from the coprocessor.
CPDOUT[31:0]	Output	This is the 32-bit coprocessor read data bus for transferring data to the coprocessor.
CPEN	Input	Coprocessor enable. If no coprocessor is used in an embedded ARM7EJ-S processor design, this signal must be tied to LOW to save power. This is a static signal.
CPINSTR[31:0]	Output	This is the 32-bit coprocessor instruction bus over which instructions are transferred to the coprocessor pipeline follower.
CPJBIT	Output	The coprocessor instruction Jazelle bit. When HIGH, this indicates that the ARM7EJ-S processor is in Jazelle state. This signal is sampled by the coprocessor pipeline follower.
CPLATECANCEL	Output	If the coprocessor late cancel signal is HIGH during the first memory cycle of a coprocessor instruction, then the coprocessor must cancel the instruction without changing any internal state. This signal is only asserted in cycles where theprevious instruction accessed memory and a Data Abort occurred.
CPnMREQ	Output	This is the coprocessor not memory request. When the signal is LOW on the rising edge of **CLK** and **CPCLKEN** is HIGH, the instruction of **CPINSTR** must enter the coprocessor pipeline.

Table A-5 Coprocessor interface signals (continued)

Name	Direction	Description
CPnTRANS	Output	The not memory translate signal: • when LOW, indicates that the processor is in User mode • when HIGH, indicates that the processor is in Privileged mode. This signal is sampled by the coprocessor pipeline follower.
CPPASS	Output	This signal indicates that there is a coprocessor instruction in the Execute stage of the pipeline that must be executed.
CPTBIT	Output	Coprocessor instruction Thumb bit. When HIGH, indicates that the ARM7EJ-S processor is in Thumb state. This signal is sampled by the coprocessor pipeline follower.

A.6 Debug signals

The debug signals are shown in Table A-6.

Table A-6 Debug signals

Name	Direction	Description
DBGACK	Output	Debug acknowledge. When HIGH, indicates that the processor is in debug state.
DBGCOMMRX	Output	*Debug Communications Channel* (DCC) receive. When HIGH, this signal denotes that the DCC receive buffer contains valid data waiting to be read by the processor.
DBGCOMMTX	Output	DCC transmit. When HIGH, this signal denotes that the DCC transmit buffer is empty.
DBGEBKPT	Input	Debug breakpoint. This signal is asserted by the external hardware to halt execution of the processor for debug purposes. It will cause the processor to enterdebug state if it is HIGH at the end of an instruction fetch, and that instruction reaches the Execute stage of the processor pipeline, or if it is HIGH at the end of a data memory request cycle.
DBGEN	Input	Debug enable. This signal allows the debug features of the processor to be disabled, it must be tied LOW when debugging is not required.
DBGEXT[1:0]	Input	EmbeddedICE external input. This input to the EmbeddedICE logic enables breakpoints and watchpoints to be dependent on external conditions.
DBGIR[3:0]	Output	TAP controller instruction register These four bits reflect the current instruction loaded into the TAP controller instruction register. These bits change when the TAP state machine is in the UPDATE-IR state.
DBGnTDOEN	Output	Not **DBGTDO** enable. When LOW, this signal denotes that serial data is being driven out on the **DBGTDO** output. **DBGnTDOEN** is usually used as an output enable for a **DBGTDO** pin in a packaged part.
DBGnTRST	Input	Not test reset. This is the active LOW reset signal for the EmbeddedICE internal state. This signal is a level-sensitive asynchronous reset input.
DBGRNG[1:0]	Output	EmbeddedICE Rangeout. This output indicates that the corresponding EmbeddedICE watchpoint unit has matched the conditions currently present on the address, data, and control buses. This signal is independent of the state of the enable control bit of the watchpoint unit.
DBGRQI	Output	Internal debug request. This signal represents the state of bit 1 of the debug control register that is combined with **EDBGRQ** and presented to the core debug logic.
DBGSCREG[4:0]	Output	Scan chain select number. These five bits reflect currently selected scan chain by the TAP Scan Chain Register controller. These bits change when the TAP state machine is in the UPDATE-DR state.

Table A-6 Debug signals (continued)

Name	Direction	Description
DBGSDIN	Output	Output boundary scan serial input data. This signal contains the serial data to be applied to an external scan chain.
DBGSDOUT	Input	Input boundary scan serial output data. This is the serial data out of an external scan chain. When an external boundary scan chain is not connected, this input must be tied LOW.
DBGTAPSM[3:0]	Output	TAP controller state machine This bus reflects the current state of the TAP controller state machine.
DBGTCKEN	Input	Synchronous enable for debug logic accessed using the JTAG interface.
DBGTDI	Input	Test data input to the debug logic.
DBGTDO	Output	Test data output from the debug logic.
DBGTMS	Input	Test mode select for the TAP controller.
EDBGRQ	Input	External debug request. An external debugger can force the processor to enter debug state by asserting this signal.
TAPID[31:0]	Input	Boundary scan ID code. This input specifies the ID code value shifted out on **DBGTDO** when the IDCODE instruction is entered into the TAP controller.

A.7 ETM interface signals

ETM interface signals are shown in Table A-7. For other ETM signals and how to connect a core, see the *ETM9 (Rev 2a) Technical Reference Manual*.

Table A-7 ETM interface signals

Name	Direction	Description
ETMBIGEND	Output	Big-endian configuration indication for the ETM.
ETMCHSD[1:0]	Output	Coprocessor handshake decode indication for the ETM.
ETMCHSE[1:0]	Output	Coprocessor handshake execute indication for the ETM.
ETMCLKEN	Output	ETM clock enable.
ETMDA[31:0]	Output	Data address for the ETM.
ETMDABORT	Output	Data abort for the ETM.
ETMDBGACK	Output	Debug state indication for the ETM.
ETMDMAS[1:0]	Output	Data memory access size indication for the ETM.
ETMDnMREQ	Output	Data memory request for the ETM.
ETMDnRW	Output	Data read and write indication for the ETM.
ETMDSEQ	Output	Sequential data access indication for the ETM.
ETMHIVECS	Output	Exception vectors configuration indication for the ETM.
ETMIA[31:0]	Output	Instruction address for the ETM.
ETMID15TO11[15:11]	Output	Instruction data field for the ETM.
ETMID31TO25[31:25]	Output	Instruction data field for the ETM.
ETMIJBIT	Output	Jazelle state indication for the ETM.
ETMInMREQ	Output	Sequence pipeline follower. If LOW at the end the cycle, then the processor requires an ETM memory access during the following cycle.
ETMINSTREXEC	Output	Instruction execute indication for the ETM.
ETMINSTRVALID	Output	Instruction valid indication for the ETM.
ETMISEQ	Output	Sequential instruction access for the ETM.
ETMITBIT	Output	Thumb state indication for the ETM.
ETMLATECANCEL	Output	Coprocessor late cancel indication for the ETM.

Table A-7 ETM interface signals (continued)

Name	Direction	Description
ETMPASS	Output	Coprocessor instruction execute indication for the ETM.
ETMPROCID[31:0]	Output	Process ID for the ETM.
ETMPROCIDWR	Output	Process ID Write indication for the ETM.
ETMPWRDOWN	Input	This signal must be tied HIGH if an ETM is not present in the design.
ETMRDATA[31:0]	Output	Read data for the ETM.
ETMRNGOUT[1:0]	Output	Watchpoint register match indication for the ETM.
ETMWDATA[31:0]	Output	Write data for the ETM.
ETMZIFIRST	Output	Indicates the current ARM instruction is the first being traced for the current Jazelle instruction.
ETMZILAST	Output	Indicates the current ARM instruction is the last being traced for the current Jazelle instruction.
TRACEPROCID[31:0]	Input	Trace Process ID. This signal comes from a CP15, is registered, and then exported as **ETMPROCID[31:0]**.
TRACEPROCIDWR	Input	Trace Process ID Write. This signal is asserted when **TRACEPROCID[31:0]** is active.

Appendix B
Debug in Depth

This appendix describes in further detail the debug features of the ARM7EJ-S processor, and includes additional information about the EmbeddedICE-RT logic. It contains the following sections:

B.1 Scan chains and JTAG interface

There are two JTAG-style scan chains within the ARM7EJ-S processor. These enable debugging and EmbeddedICE-RT programming.

The scan chains enable commands to be serially shifted into the ARM core, enabling the state of the core and the system to be interrogated. The JTAG interface requires only five pins on the package.

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

B.1.1 Debug scan chains

The two scan paths used for debug purposes are referred to as scan chain 1 and scan chain 2, and are shown in Figure B-1.
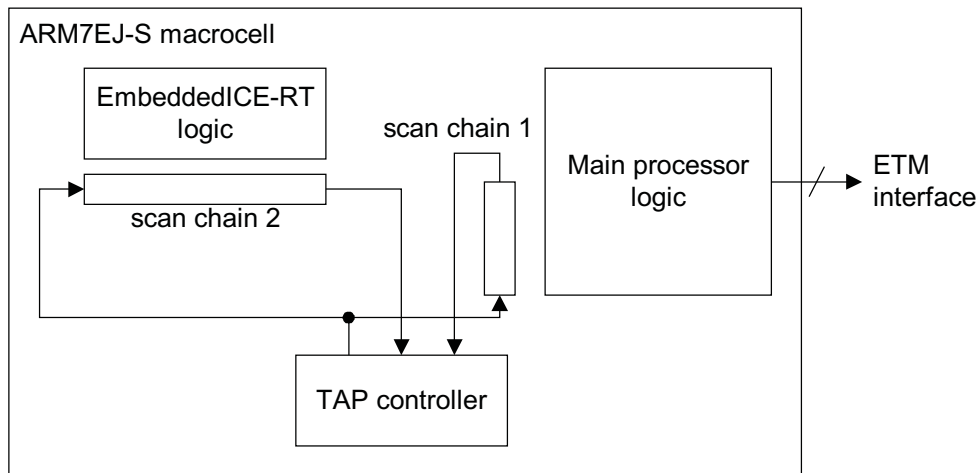
Figure B-1 Debug block diagram

Scan chain 1

Scan chain 1 is used for debugging the ARM7EJ-S processor when it has entered debug state. You can use it to:

- inject instructions into the processor pipeline
- read and write its registers
- perform memory accesses.

Scan chain 2

Scan chain 2 enables access to the EmbeddedICE-RT registers. Refer to *Test data registers* on page B-10 for details.

B.1.2   TAP state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure B-2 on page B-4 shows the state transitions that occur in the TAP controller. The state numbers shown in the diagram are output from the ARM7EJ-S processor on the **DBGTAPSM[3:0]** signals.
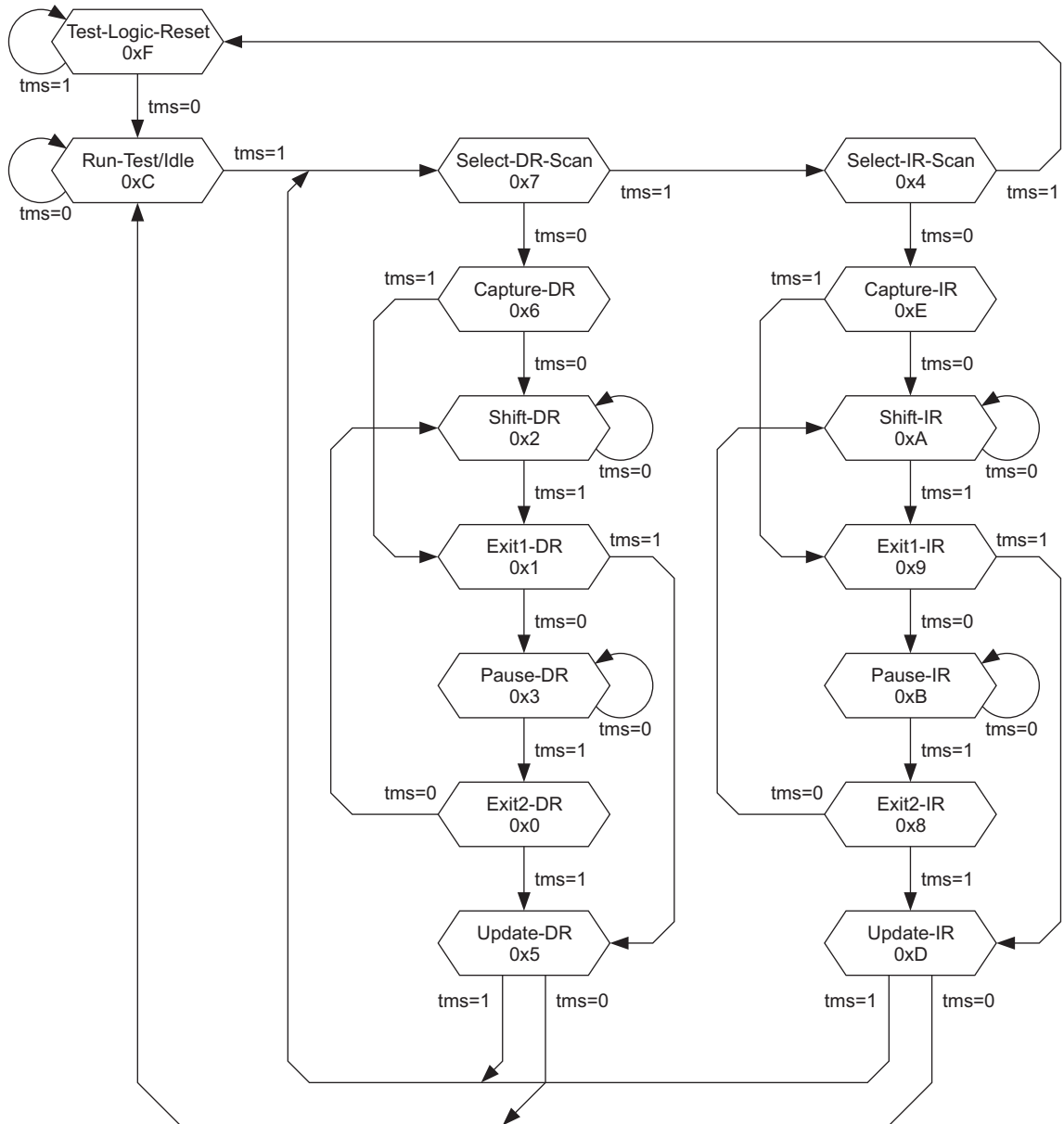
Figure B-2 Test access port controller state transitions

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

B.2   Resetting the TAP controller

The boundary-scan interface includes a state machine controller called 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:

•   to ready the boundary-scan interface for use, drive **DBGnTRST** LOW, and then HIGH again

•   to prevent the boundary scan interface from being used, the **DBGnTRST** input can be tied permanently LOW.

———— **Note** ————

A clock on **CLK** with **DBGTCKEN** HIGH is not necessary to reset the device.

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 **CLK** is pulsed while enabled by **DBGTCKEN**, the contents of the ID register are clocked out of **DBGTDO**.

B.3   Instruction register

The instruction register is four bits in length.

There is no parity bit.

The fixed value b0001 is loaded into the instruction register during the CAPTURE-IR controller state.

B.4   Public instructions

Instructions are loaded into the TAP state machine by scanning the appropriate bit pattern for the instruction when the TAP controller is in the SHIFT-IR state, and then advancing the TAP controller through the UPDATE-IR state.

Table B-1 shows the public instructions.

Table B-1 Public instructions

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

In the following descriptions, the ARM7EJ-S processor samples **DBGTDI** and **DBGTMS** on the rising edge of **CLK** with **DBGTCKEN** HIGH. All output transitions on **DBGTDO** occur as a result of the rising edge of **CLK** with **DBGTCKEN** HIGH.

B.4.1   EXTEST (b0000)

The EXTEST instruction enables a boundary scan chain to be connected between the **DBGSDIN** and **DBGSDOUT** pins. External logic, based on the **DBGTAPSM**, **DBGSCREG**, and **DBGIR** signals is required to use the EXTEST function for such a boundary scan chain. Using EXTEST with scan chain 1 or scan chain 2 selected is UNPREDICTABLE.

B.4.2   SAMPLE/PRELOAD (b0011)

You must use this instruction to preload the boundary scan register with known data prior to selecting INTEST or EXTEST instructions.

Copyright © 2001 ARM Limited. All rights reserved.

B.4.3 SCAN_N (b0010)

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.

B.4.4 INTEST (b1100)

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. For example, in test mode, input cells select the output of the scan chain to be applied to the core.

- 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.

B.4.5 IDCODE (b1110)

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 *Device IDentification (ID) code register* on page B-10 for 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.

B.4.6   BYPASS (b1111)

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, 0 is captured in the bypass register.

•   In the SHIFT-DR state, test data is shifted into the bypass register through **DBGTDI**, and shifted out through **DBGTDO** after a delay of one **CLK** 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.

B.4.7   RESTART (b0100)

The RESTART instruction is used to restart the processor on exit from debug state. The RESTART instruction connects the bypass register between **DBGTDI** and **DBGTDO**, and the TAP controller behaves as if the BYPASS instruction has been loaded.

The processor exits debug state when the RUN-TEST/IDLE state is entered.

B.5 Test data registers

There are six test data registers that can be selected to connect between **DBGTDI** and **DBGTDO**:

- bypass register
- ID code register
- instruction register
- scan path select register
- scan chain 1
- scan chain 2.

In addition, other scan chains can be added between **DBGSDOUT** and **DBGSDIN**, and selected when in INTEST mode.

In the following descriptions, data is shifted during every **CLK** cycle when **DBGTCKEN** enable is HIGH.

B.5.1 Bypass register

The Bypass register purpose, bit length, and operating mode description is given below:

Purpose	Bypasses the device during scan testing by providing a path between **DBGTDI** and **DBGTDO**.
Length	1 bit.
Operating mode	When the BYPASS instruction, or any undefined 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 **CLK** cycle enabled by **DBGTCKEN**.
	A logic 0 is loaded from the parallel input of the bypass register in the CAPTURE-DR state. There is no parallel output from the bypass register.

B.5.2 Device IDentification (ID) code register

Contact your supplier for the correct device identification code.

Purpose	Reads the 32-bit device identification code. No programmable supplementary identification code is provided.
Length	32 bits. The format of the register is as shown in Figure B-3 on page B-11.

31	28	27	12	11	1	0
Version		Part number		Manufacturer identity		1
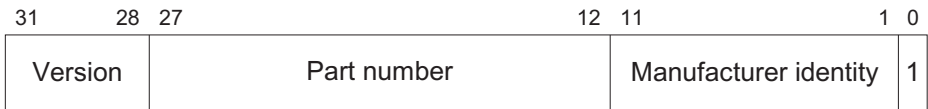
Figure B-3 ID code register format

Operating mode   When the IDCODE instruction is current, the ID register is selected as the serial path between **TDI** and **TDO**. 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.

The least significant bit of the register is scanned out first.

B.5.3   Instruction register

The Instruction register purpose, bit length, and operating mode description is given below:

Purpose   Specifies a 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, b0001 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 specifies the current instruction.

On reset, IDCODE specifies the current instruction.

B.5.4   Scan path select register

The Scan path select register purpose, bit length, and operating mode description is given below:

Purpose   Changes the current active scan chain.

Length   5 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, b10000 is loaded into this register. This value is shifted out during SHIFT-DR (least significant bit first), while a new value is shifted in (least significant bit first). During the UPDATE-DR state, the value in the scan path select register selects a scan chain to become the currently active scan chain. All further 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 3 is selected as the active scan chain.

——— **Note** ———

Scan chain 3 is not implemented in the ARM7EJ-S (Rev 1) 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.

The number of the currently-selected scan chain is reflected on the **DBGSCREG[4:0]** output bus. You can use the TAP controller to drive external chains in addition to those within the ARM7EJ-S processor. The external scan chain is connected between **DBGSDIN** and **DBGSDOUT**, and must be assigned a number. The control signals are derived from **DBGSCREG[4:0]**, **DBGIR[4:0]**, **DBGTAPSM[3:0]** and the clock, **CLK**, and clock enable, **DBGTCKEN**.

Table B-2 shows the scan chain numbers allocated by ARM Limited.

Table B-2 Scan chain number allocation

Scan chain number	Function
0	Reserved
1	Debug
2	EmbeddedICE-RT programming

Table B-2 Scan chain number allocation (continued)

Scan chain number	Function
3	External boundary scan signals for user-implemented scan chain
4–15	Reserved
16–31	Unassigned

The scan chain present between **DBGSDIN** and **DBGSDOUT** is connected between **DBGTDI** and **DBGTDO** whenever scan chain 3 is selected, or when any unassigned scan chain number is selected. If there is more than one external scan chain, a multiplexor must be built externally to apply the desired scan chain output to **DBGSDOUT**. The multiplexor can be controlled by decoding **DBGSCREG[4:0]**.

B.5.5   Scan chains 1 and 2

The scan chains enable serial access to the core logic and to the EmbeddedICE hardware for programming purposes. Each scan chain cell is simple, and comprises a serial register and a multiplexor. A typical cell is shown in Figure B-4.
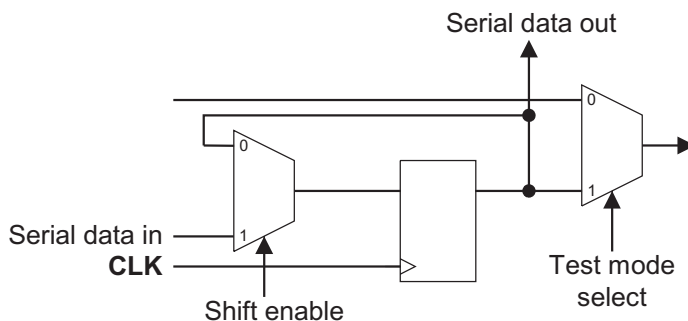
Figure B-4 Typical scan chain cell

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

The scan chain 1 purpose, bit length, and description is given below:

Purpose   Scan chain 1 is used for communication between the debugger and the ARM7EJ-S processor. It is used to read and write data, and to scan instructions into the instruction pipeline. The SCAN_N instruction is used to select scan chain 1.

Length   67 bits.

Scan chain 1 allows you to separately access the internal ARM7EJ-S processor data or instruction busses. For data you can access **RDATA_D[31:0]** when the core is performing a read. This is the same as **RDATA[31:0]** when accessing data. You can access **WDATA[31:0]** when the core is performing a write. For instructions, access is available to the **RDATA_I[31:0]** bus. This is the same as the **RDATA[31:0]** when performing instruction accesses.

Scan chain 1 also provides access to the control bits, SYSPEED and WPTANDBKPT, there is one additional unused bit that must be zero when writing, and is UNPREDICTABLE when reading.

There are 67 bits in this scan chain, the order being (from serial data in to out):
1. **RDATA_I[31:0]**.
2. **SYSPEED**.
3. **WPTANDBKPT**.
4. Unused bit.
5. **RDATA_D[31:0]** or **WDATA[31:0]**.

Bit 0 of **RDATA** or **WDATA** is therefore the first bit to be shifted out.

Table B-3 shows the bit allocations for scan chain 1.

Table B-3 Scan chain 1 bit order

Bit number	Function	Type
66	**RDATA_D[0]/WDATA[0]**	Bidirectional
...	...	Bidirectional
35	**RDATA_D[31]/WDATA[31]**	Bidirectional
34	Unused	-
33	WPTANDBKPT	Input
32	SYSSPEED	Input
31	**RDATA_I[31]**	Input
...	...	Input
0	**RDATA_I[0]**	Input

The two control bits serve the following purposes:

• While debugging, the value placed in the SYSSPEED control bit determines whether the ARM7EJ-S processor synchronizes back to system speed before executing the instruction. See *System speed access* on page B-25 for further details.

• After the ARM7EJ-S processor has entered debug state, the first time SYSSPEED is captured and scanned out, its value tells the debugger whether the core has entered debug state from a breakpoint (SYSSPEED LOW), or a watchpoint (SYSSPEED HIGH). If the instruction directly following one which causes a watchpoint has a breakpoint set on it, then the WPTANDBKPT bit is set. This situation does not affect how to restart the code.

• For a read the data value taken from the 32 bits in the scan chain allocated for data is used to deliver the **RDATA_D[31:0]** value to the core.

• When a write is being performed by the processor the **WDATA[31:0]** value is returned in the data part of the scanned out value.

Scan chain 2

The scan chain 2 purpose, bit length, scan chain order and operation description is given below:

Purpose Scan chain 2 enables access to the EmbeddedICE registers. To do this, scan chain 2 must be selected using the SCAN_N instruction, and then the TAP controller instruction must be changed to INTEST.

Length 38 bits.

Scan chain order From **DBGTDI** to **DBGTDO**. Read/write, register address bits 4 to 0, data values bits 31 to 0.

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 register to be accessed.

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

B.6   Determining the core and system state

When the ARM7EJ-S processor is in debug state, you can examine the core and system state by forcing the load and store multiples into the instruction pipeline.

Before examining the core and system state, the debugger must determine whether the processor entered debug from Thumb state, ARM state or Jazelle state by examining bits 4 and 5 of the EmbeddedICE-RT debug status register. When bit 4 is set, the core has entered debug from Thumb state. When bit 5 is set, the core has entered debug from Jazelle state. When bit 4 and 5 is clear the core has entered debug from ARM state.

B.6.1   Determining the core state

When the processor has entered debug state from Thumb state, the simplest method 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 sequence of Thumb instructions shown in Example B-1 on the core (with the SYSSPEED bit clear).

Example B-1 Forcing the processor from Thumb state to ARM state

```
STR R0, [R1] ; Save R0 before use
MOV R0, PC   ; Copy PC into R0
STR R0, [R1] ; 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, the simplest method, when shifting scan chain 1, is to repeat the instruction. 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.

When the processor has entered debug state from Jazelle state, the debugger must force the core into ARM state before doing anything else. The debugger can then execute the same sequence of instructions to determine the processor state.

To force the processor into ARM state from Jazelle state execute the sequence of ARM instructions shown in Example B-2 on page B-18 on the core (with the SYSSPEED bit clear).

Example B-2 Forcing the processor from Jazelle state to ARM state

```
STMIA r0,{r0,pc} ; save r0 before use and take note of PC at debug entry
LDMIA r0,{r0}    ; load r0 with a word aligned address (to jump to ARM state)
BX r0            ; jump to ARM state
```

———— **Note** ————

When the processor has entered debug state from Jazelle state the scan chain is used to scan instructions into the ARM instruction decode stage of the pipeline not the Jazelle decode stage.

You can use the sequences of ARM instructions shown in Example B-3 to determine the processor state.

With the processor in the ARM state, typically the first instruction to execute is:

```
STMIA R0, {R0-R15}
```

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, and you can use any register.

After you have determined the values in the bank of registers available in the current mode, you might want to access the other 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 can occur from any mode into any other mode.

The debugger must restore the original mode before exiting debug state. For example, if the debugger has been requested to return the state of the User mode registers and FIQ mode registers, and debug state is entered in Supervisor mode, the instruction sequence can be as shown in Example B-3.

Example B-3 Determining the core state

```
STMIA 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
STMIA R0, {R13,R14}; Save registers not previously visible
ORR R0, 0x01; Select FIQ mode
MSR CPSR, R0; Enter FIQ mode
STMIA 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, 67 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 ARM7EJ-S processor is fully static. However, you cannot use this method for determining the state of the rest of the system.

While in debug state, you can only scan the following ARM or Thumb instructions into the instruction pipeline for execution:

• all data processing operations

• all load, store, load multiple, and store multiple instructions

• MSR and MRS

• B, BL, and BX.

B.6.2 Determining the system state

To meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously. Therefore, the ARM7EJ-S processor must be forced to synchronize back to system speed. Bit 32 of scan chain 1, SYSSPEED, controls this.

You can place a legal debug instruction onto the instruction data bus of scan chain 1 with bit 32 (the SYSSPEED bit) clear. This instruction is then normally executed at debug speed. To execute an instruction at system speed, a NOP (such as MOV R0, R0) must be scanned in as the next instruction with bit 32 set.

After the system speed instructions are scanned into the instruction data bus and clocked into the pipeline, the RESTART instruction must be loaded into the TAP controller. This causes the ARM7EJ-S processor automatically to resynchronize back to **CLK** conditioned with **CLKEN** when the TAP controller enters RUN-TEST/IDLE state, and executes the instruction at system speed. Debug state is reentered once the instruction completes execution, when the processor switches itself back to **CLK** conditioned with **DBGTCKEN**. When the instruction completes, **DBGACK** is HIGH. At this point INTEST can be selected in the TAP controller, and debugging can resume.

To determine if a system speed instruction has completed, the debugger must look at SYSCOMP (bit 3 of the debug status register). The ARM7EJ-S processor must access memory through the data bus interface, as this access can be stalled indefinitely by **CLKEN**. Therefore, the only way to determine if the memory access has completed is to examine the SYSCOMP bit. When this bit is set, the instruction has completed.

The state of the system memory can be fed back to the debug host by using system speed load multiple instructions and debug speed store multiple instructions.

Instructions that can have the SYSSPEED bit set

There are restrictions on which instructions can have the SYSSPEED bit set. The valid instructions on which to set this bit are:

- loads
- stores
- load multiple
- store multiple.

When the ARM7EJ-S processor returns to debug state after a system speed access, the SYSSPEED bit is clear. The state of this bit gives the debugger information about why the core entered debug state the first time this scan chain is read.

B.6.3   Exit from debug state

Leaving debug state involves:
- restoring the internal state of the ARM7EJ-S processor
- causing a branch to the next instruction to be executed
- synchronizing back to **CLK** conditioned with **CLKEN**.

After restoring the internal state, a branch instruction must be loaded into the pipeline. See *Behavior of the program counter during debug* on page B-23 for details on calculating the branch.

The SYSSPEED bit of scan chain 1 forces the ARM7EJ-S processor to resynchronize back to **CLK** conditioned with **CLKEN**. The penultimate instruction in the debug sequence is a branch to the instruction at which execution is to resume. This is scanned in with bit 32 (SYSSPEED) clear. The final instruction to be scanned in is a NOP (such as MOV R0, R0), with bit 32 set. The core is then clocked to load this instruction into the pipeline.

The ARM instructions shown in Example B-4 on page B-21 are used to exit from debug state to Jazelle state.

Example B-4 Forcing the processor from debug state to Jazelle state

```
BXJ r0          ; jump to Jazelle state
LDMIA r0,{pc}   ; reload pc
MOV pc,pc       ; system speed access to return to functional Jazelle state
```

Next, 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, and clock resynchronization to **CLK** conditioned with **CLKEN** occurs within the ARM7EJ-S processor. Normal operation then resumes, with instructions being fetched from memory.

The delay, waiting until the state machine is in RUN-TEST/IDLE state, enables conditions to be set up in other devices in a multiprocessor system without taking immediate effect. Then, when RUN-TEST/IDLE state is entered, all the processors resume operation simultaneously.

The function of **DBGACK** is to tell the rest of the system when the ARM7EJ-S processor is in debug state. You can use this signal to inhibit peripherals such as watchdog timers that have real-time characteristics. Also, you can use **DBGACK** to mask out memory accesses that are caused by the debugging process. For example, when the ARM7EJ-S processor enters debug state after a breakpoint, the instruction pipeline contains the breakpointed instruction plus two other instructions that have been prefetched. On entry to debug state, the pipeline is flushed. So, on exit from debug state, the pipeline must be refilled to its previous state. Therefore, because of the debugging process, more memory accesses occur than are normally expected. It is possible, using the **DBGACK** signal and a small amount of external logic, for a peripheral which is sensitive to the number of memory accesses to return the same result with and without debugging.

--- **Note** ---

You can only use **DBGACK** in this way using breakpoints. It does not mask the correct number of memory accesses after a watchpoint.

For example, consider a peripheral that simply counts the number of instruction fetches. This device must return the same answer after a program has run both with and without debugging.

Figure B-5 on page B-22 shows the behavior of the ARM7EJ-S processor on exit from debug state.
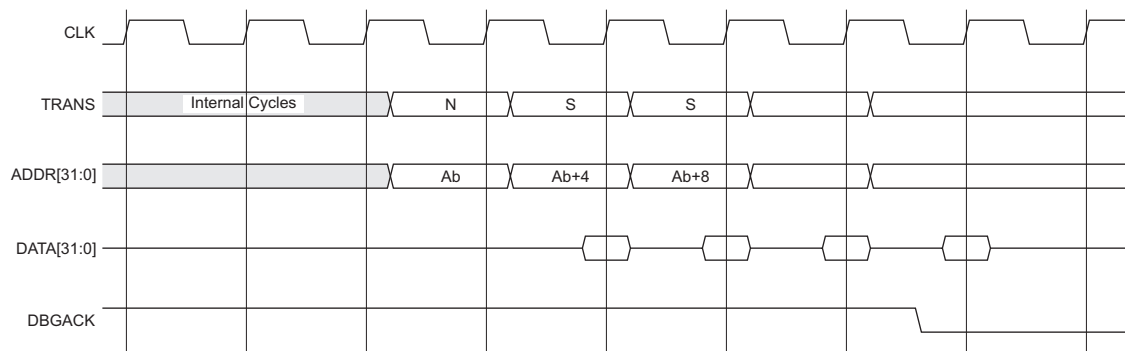
Figure B-5 Debug exit sequence

In Figure B-6 on page B-29, two instructions are fetched after the breakpointed instruction. Figure B-5 shows that **DBGACK** masks the first three instruction fetches out of the debug state, corresponding to the breakpoint instruction, and the two instructions prefetched after it.

Under some circumstances **DBGACK** can remain HIGH for more than three instruction fetches. Therefore, if you require precise instruction access counting, you must provide some external logic to generate a modified **DBGACK** that always falls after three instruction fetches.

———— **Note** ————

When system speed accesses occur, **DBGACK** remains HIGH throughout. It then falls after the system speed memory accesses are completed, and finally rises again as the processor reenters debug state. Therefore, **DBGACK** masks all system speed memory accesses.

—————————————

B.7　Behavior of the program counter during debug

The debugger must keep track of what happens to the PC, so that you can force the ARM7EJ-S processor to branch back to the place at which program flow was interrupted by debug. Program flow can be interrupted by any of the following:

- a breakpoint
- a watchpoint
- a watchpoint when another exception occurs
- a debug request
- a system speed access.

B.7.1　ARM and Thumb state breakpoints

Entry to debug state from a breakpointed instruction advances the PC by 16 bytes in ARM state, or 8 bytes in Thumb state. Each instruction executed in debug state advances the PC by one address (4 bytes). The normal 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 ARM7EJ-S processor entered debug state from a breakpoint set on a given address and two debug speed instructions were executed, a branch of seven addresses must occur (four for debug entry, plus two for the instructions, plus one for the final branch). The following sequence shows ARM instructions scanned into scan chain 1. This is the *Most Significant Bit* (MSB) first, so the first digit represents the value to be scanned into the SYSSPEED bit, followed by the instruction:

0 EAFFFFF9 ; Branch minus seven addresses (two's complement)

1 E1A00000 ; NOP (MOV R0, R0), SYSSPEED bit is set

After the ARM7EJ-S 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).

B.7.2　ARM and Thumb state watchpoints

To return to program execution after entry to debug state from a watchpoint, use the same procedure described in *ARM and Thumb state 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.

　　　Copyright © 2001 ARM Limited. All rights reserved.

B.7.3   Jazelle state breakpoints and watchpoints

On entry to debug from Jazelle the PC contains the address of the instruction not executed because of debug entry plus four bytes (except for software breakpoints where PC equals address of breakpoint instruction plus four bytes). The PC is then frozen until the core state is forced to ARM state. This behavior means that whatever the cause of debug entry from Jazelle state the return address is always the PC value at entry minus four bytes.

B.7.4   Watchpoint with another exception

If a watchpointed access also has a Data Abort returned, the ARM7EJ-S processor enters debug state in Abort mode. Entry into debug is prevented 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 ARM7EJ-S processor enters debug state in the mode of the exception. The debugger must check to see if 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 must be given the choice of servicing the exception before debugging.

For example, suppose that an abort has occurred on a watchpointed access and ten instructions have been executed in debug state. You can use the following sequence to return to program execution:

0 EAFFFFF1; Branch minus 15 addresses (two's complement)

1 E1A00000; NOP (MOV R0, R0), SYSSPEED bit is set

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 ARM7EJ-S processor re-enters debug state.

B.7.5   Watchpoint and breakpoint

It is possible to have a watchpoint and breakpoint condition occurring simultaneously. This can happen when an instruction causes a watchpoint, and the following instruction has been breakpointed. You must perform the same calculation as for *ARM and Thumb*

state breakpoints on page B-23 or *Jazelle state breakpoints and watchpoints* on page B-24 to determine where to resume. In this case, it is at the breakpoint instruction, because this has not been executed.

B.7.6 Debug request

Entry into debug state through a debug request is similar to a breakpoint. Entry to debug from ARM or Thumb state adds four addresses to the PC, and every instruction executed in debug state adds one address, and from Jazelle state adds four bytes.

For example, the following sequence handles a situation in which the user has invoked a debug request when in ARM or Thumb state, and then decides to return to program execution immediately:

```
0 EAFFFFFB; B minus five addresses (2's complement)
```

```
1 E1A00000; NOP (MOV R0, R0), SYSSPEED bit is set
```

This code restores the PC, and restarts the program from the next instruction.

B.7.7 System speed access

When a system speed access is performed during debug state, the value of the PC increases by five 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 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 is not caused by an instruction in the main program, and so the link register does not point to the instruction that caused the abort. An abort handler usually looks at the link register to determine the instruction that caused the abort, and the abort address. In this case, the value of the link register is invalid, but because the debugger can determine which location was being accessed, you can write the debugger to help the abort handler fix the memory system.

B.7.8 Summary of return address calculations

The calculation of the branch return address when entered from ARM or Thumb state can be summarized as:

PC-(4+N+5S)

where N is the number of debug speed instructionsexecuted (including the final branch), and S is the number of system speed instructions executed.

B.8　Priorities and exceptions

When a breakpoint or a debug request occurs, the normal flow of the program is interrupted. Therefore you can treat debug as another type of exception. The interaction of the debugger with other exceptions is described in *Behavior of the program counter during debug* on page B-23. This section covers the priorities.

B.8.1　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 might be data-dependent), the ARM7EJ-S processor enters debug state.

The Prefetch Abort, therefore, takes higher priority than the breakpoint.

B.8.2　Interrupts

When the ARM7EJ-S processor enters debug state, interrupts are automatically disabled.

If an interrupt is pending during the instruction prior to entering debug state, the ARM7EJ-S processor enters debug state in the mode of the interrupt. On entry to debug state, the debugger cannot assume that the ARM7EJ-S processor is in the mode expected by your program. The ARM7EJ-S processor 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 ARM7EJ-S processor does recognize that an interrupt has occurred.

B.8.3　Data Aborts

When a Data Abort occurs on a watchpointed access, the ARM7EJ-S processor enters debug state in Abort mode. The watchpoint, therefore, has higher priority than the abort, but the ARM7EJ-S processor remembers that the abort happened.

B.9   EmbeddedICE-RT logic

The EmbeddedICE-RT logic is integral to the ARM7EJ-S processor. It has two hardware breakpoint or watchpoint units, each of which can be configured to monitor either the instruction memory interface or the data memory interface. Each watchpoint unit has registers that set the address, data, and control fields for both values and masks. The registers used are shown in Table B-4.

When debugging, the ARM7EJ-S processor differentiates between the instruction and data flows. Therefore, you must specify if the watchpoint unit refers to instruction or data accesses by using bit 3 of the control value register:

*   when bit 3 is set, only data accesses are examined

*   when bit 3 is clear, only instruction accesses are examined.

Bit 3 of the control mask register is always clear and cannot be set. Bit 3 also determines whether the internal **IBREAKPT** or **DWPT** signal must be driven by the result of the comparison. Figure B-6 on page B-29 gives an overview of the operation of the EmbeddedICE-RT logic.

The general arrangement of the EmbeddedICE-RT logic is shown in Figure B-6 on page B-29.

B.9.1   Register map

The EmbeddedICE-RT logic register map is shown in Table B-4.

Table B-4 EmbeddedICE-RT logic register map

Address	Width	Function	Type
b00000	6	Debug control	Read/write
b00001	5	Debug status	Read-only
b00010	8	Vector catch control	Read/write
b00100	6	DCC control	Read-only[a]
b00101	32	DCC data	Read/write
b01000	32	Watchpoint 0 address value	Read/write
b01001	32	Watchpoint 0 address mask	Read/write
b01010	32	Watchpoint 0 data value	Read/write
b01011	32	Watchpoint 0 data mask	Read/write

Table B-4 EmbeddedICE-RT logic register map (continued)

Address	Width	Function	Type
b01100	9	Watchpoint 0 control value	Read/write
b01101	8	Watchpoint 0 control mask	Read/write
b10000	32	Watchpoint 1 address value	Read/write
b10001	32	Watchpoint 1 address mask	Read/write
b10010	32	Watchpoint 1 data value	Read/write
b10011	32	Watchpoint 1 data mask	Read/write
b10100	9	Watchpoint 1 control value	Read/write
b10101	8	Watchpoint 1 control mask	Read/write

a. An attempted write to the DCC control register can be used to reset bit 0 of that register.

B.9.2   Programming and reading EmbeddedICE-RT logic registers

An EmbeddedICE-RT logic register is programmed by shifting data into the EmbeddedICE scan chain (scan chain 2). The scan chain is a 38-bit register comprising:

* a 32-bit data field
* a 5-bit address field
* a read/write bit.

This is shown in Figure B-6 on page B-29.
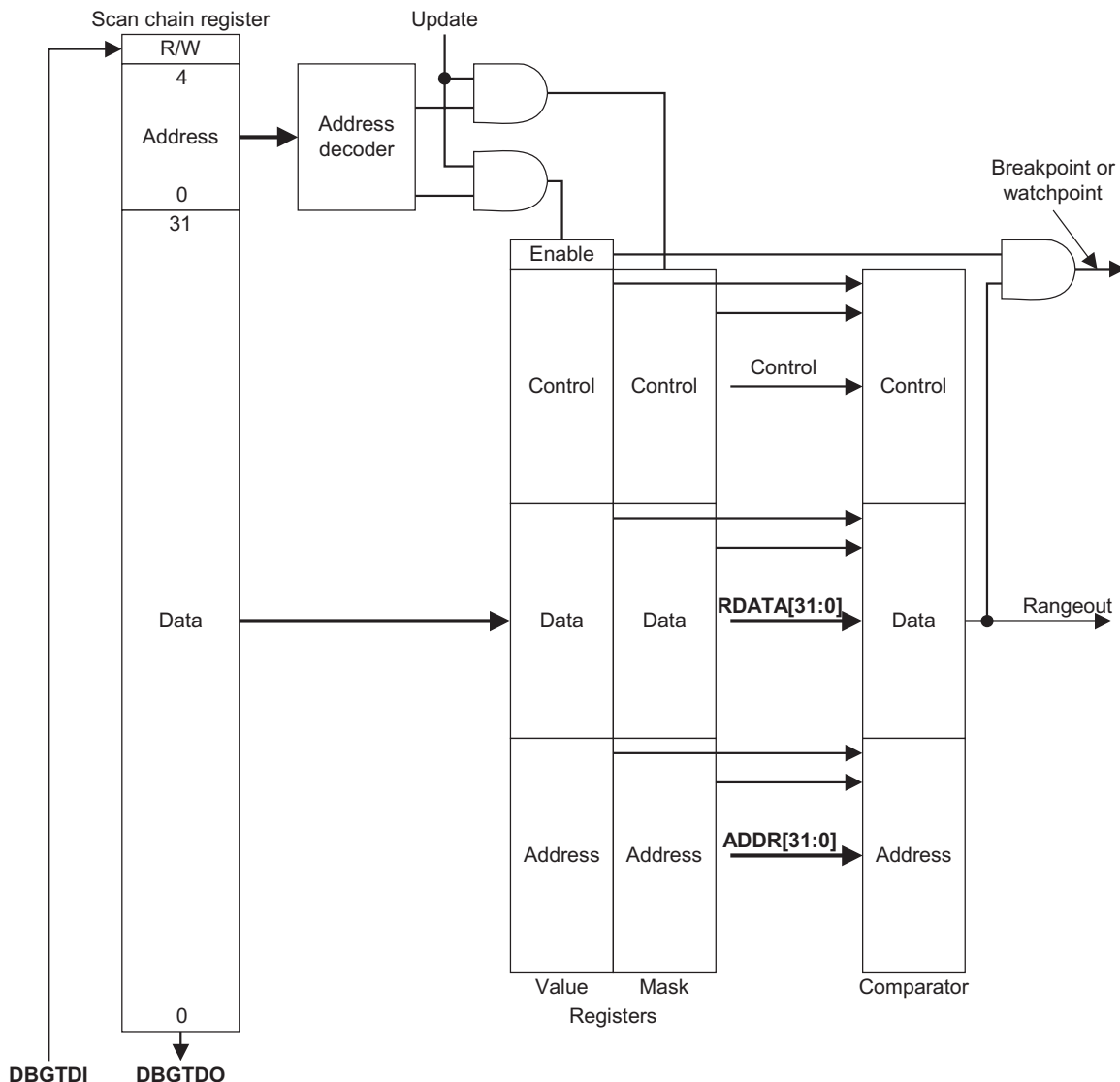
Figure B-6 EmbeddedICE macrocell overview

If a watchpoint is requested on a particular memory location but the data value is irrelevant, you can program the data mask register to 0xFFFF FFFF (all bits set), so that the entire data bus value is masked.

B.9.3   Using the mask registers

For each value register there is an associated mask register in the same format. Setting a bit in the mask register causes the corresponding bit in the value register to be ignored in any comparison.

B.9.4   Watchpoint control registers

The control value and control mask registers are mapped identically in the lower eight bits shown in Figure B-7.

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	DBGEXT	PROT[1]	PROT[0]	SIZE[1]	SIZE[0]	WRITE

Figure B-7 Watchpoint control value and mask register 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 to detect the direction of bus activity. Bit 0 is clear for a read cycle and set for a write cycle.

SIZE[1:0]   Compares against the **SIZE[1:0]** signal from the core to detect the size of bus activity.

The encoding is shown in Table B-5.

Table B-5 SIZE bits

Register bit 2	Register bit 1	Data size
Clear	Clear	Byte
Clear	Set	Halfword
Set	Clear	Word
Set	Set	(Reserved)

PROT[0]   Is used to detect whether the current cycle is an instruction fetch (bit 3 is clear), or a data access (bit 3 is set).

PROT[1]   Is used to compare against the not translate signal from the core to distinguish between User mode accesses when bit 4 is clear, and non-User mode accesses when bit 4 is set.

DBGEXT[1:0]

Is an external input to EmbeddedICE 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 to implement, for example, debugger requests of the form:`breakpoint on address YYY only when in process XXX`.

In the EmbeddedICE-RT logic, 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 **nTRST** is LOW.

RANGE   In the ARM7EJ-S EmbeddedICE-RT logic, the **RANGEOUT** 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, such as for 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. These bits remove the dependency on particular signals.

B.9.5   Debug control register

The debug control register is 6 bits wide. Writing control bits occurs during a register write access (with the read/write bit HIGH). Reading control bits occurs during a register read access (with the read/write bit LOW).

Figure B-8 on page B-32 shows the function of each bit in this register.

5	4	3	2	1	0
EmbeddedICE disable	Monitor mode enable	SBZ	INTDIS	DBGRQ	DBGACK

Figure B-8 Debug control register format

These functions are described in Table B-6 and Table B-7.

Table B-6 Debug control register bit functions

Bit number	Name	Function
5	EmbeddedICE disable	Controls the address and data comparison logic contained within the Embedded-ICE logic. When set to 1, the address and data comparators are disabled. When set to 0, the address and data comparators are enabled. You can use this bit to save power in a system where the Embedded-ICE functionality is not required. The reset state of this bit is 0 (comparators enabled). An extra piece of logic initialized by debug reset ensures that the Embedded-ICE logic is automatically disabled out of reset. This extra logic is set by debug reset and is automatically reset on the first access to scan chain 2.
4	Monitor mode enable	Controls the selection between monitor mode debug (monitor mode enable = 1) and halt mode debug. In monitor mode, breakpoints and watchpoints cause Prefetch Aborts and Data Aborts to be taken (respectively). At reset, the monitor mode enable bit is set to 1.
3	Reserved	Should be zero.
2	INTDIS	If bit 2 (**INTDIS**) is asserted, the interrupt signals to the processor are inhibited. Table C-8 shows interrupt signal control.
1:0	DBGRQ, DBGACK	These bits enable the values on **DBGRQ** and **DBGACK** to be forced.

Table B-7 Interrupt signal control

DBGACK	INTDIS	Interrupts
0	0	Permitted
1	x	Inhibited
x	1	Inhibited

Both **IRQ** and **FIQ** are disabled when the processor is in debug state (DBGACK =1), or when **INTDIS** is forced.

As shown in Figure B-10 on page B-35, the value stored in bit 1 of the control register is synchronized and then ORed with the external **EDBGRQ** before being applied to the processor.

In the case of **DBGACK**, the value of **DBGACK** from the core is ORed with the value held in bit 0 to generate the external value of **DBGACK** seen at the periphery of the processor. 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 (in which case the internal **DBGACK** signal from the core is LOW).

The structure of the debug control and status registers is shown in Figure B-10 on page B-35.

B.9.6   Debug status register

The debug status register is ten bits wide. If it is accessed for a read (with the read/write bit LOW), the status bits are read. The format of the debug status register is shown in Figure B-9.

9 8 7 6	5	4	3	2	1	0
MOE	CPJBIT	CPTBIT	SYSCOMP	IFEN	DBGRQ	DBGACK

Figure B-9 Debug status register

The function of each bit in this register is shown in Table B-8.

Table B-8 Debug status register bit functions

Bit number	Name	Function
1:0	**DBGRQ**, **DBGACK**	Enable the values on the synchronized versions of **EDBGRQ** and **DBGACK** to be read.
2	**IFEN**	Enables the state of the core interrupt enable signal to be read.
3	**SYSCOMP**	Enables the state of the **SYSCOMP** bit from the core to be read. This enables the debugger to determine that a memory access from the debug state has completed.

Table B-8 Debug status register bit functions (continued)

Bit number	Name	Function
4	**CPTBIT**	Enables the status of the output **CPTBIT** to be read. This enables the debugger to determine what state the processor is in, and therefore which instructions to execute.
5	**CPJBIT**	Enables the status of the output **CPJBIT** to be read. This enables the debugger to determine what state the processor is in, and therefore which instructions to execute.
6:9	**MOE**	This provides *Method of Entry* (MOE) information to the debugger. On entry into debug state, the MOE field is updated to indicate the cause of debug entry. The MOE field can be read from the TAP side by accessing the EICE Debug Status Register. Table B-9 shows bit values and the associated meanings.

The structure of the debug control and status registers is shown in Figure B-10 on page B-35.

Table B-9 Method of entry

MOE[3:0]	Meaning
b0000	No debug entry (since last restart/dbg rst)
b0001	Breakpoint from EICE unit 0
b0010	Breakpoint from EICE unit 1
b0011	Soft breakpoint (BKPT instruction)
b0100	Vector catch breakpoint
b0101	External breakpoint
b0110	Watchpoint from EICE unit 0
b0111	Watchpoint from EICE unit 1
b1000	External watchpoint
b1001	Internal debug request
b1010	External debug request
b1011	Debug re-entry from system speed access
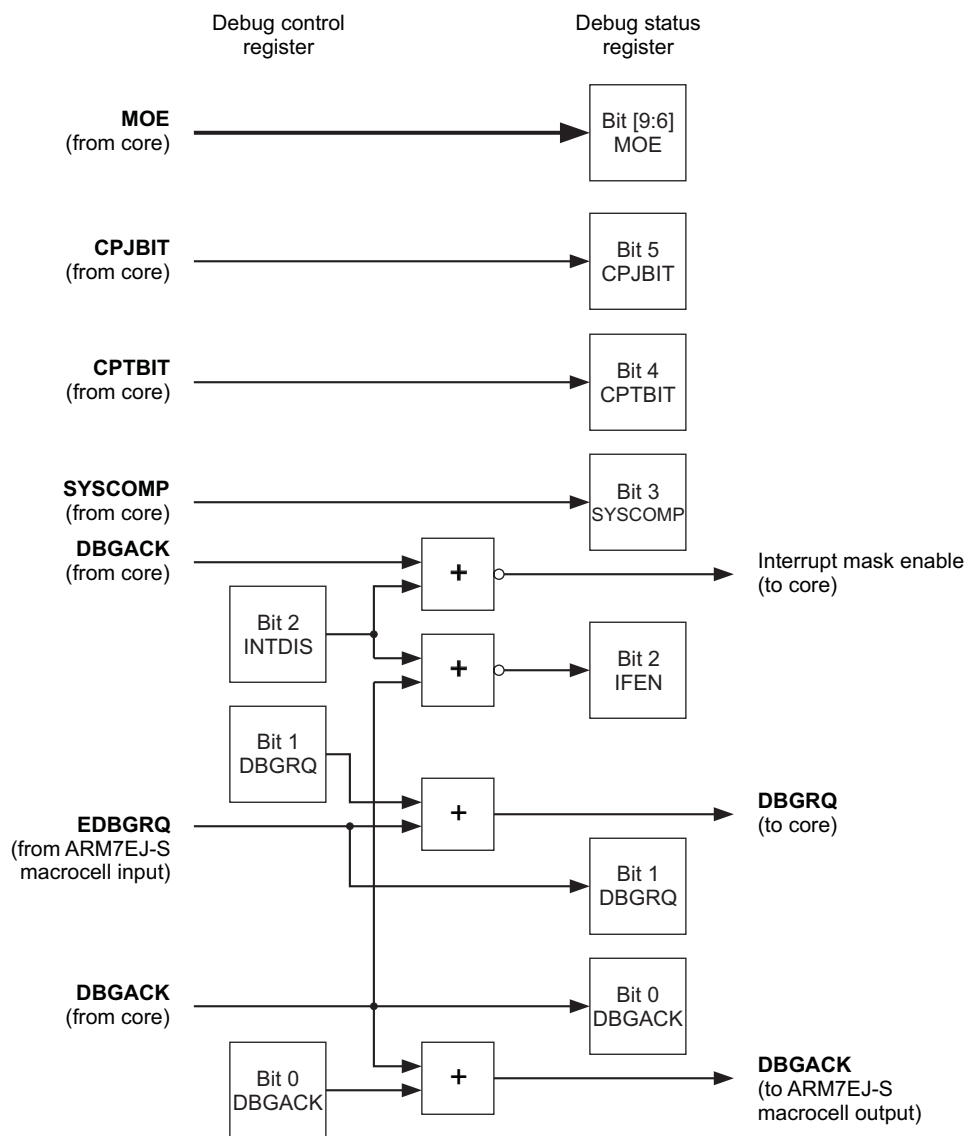
Figure B-10 Debug control and status register structure

B.9.7   Vector catch register

The EmbeddedICE-RT logic controls hardware to enable accesses to the exception vectors to be trapped in an efficient manner. This is controlled by the vector catch register, as shown in Figure B-11. The functionality is described in *Vector catching* on page B-37.

6	6	5	4	3	2	1	0
FIQ	IRQ	Reserved	D_Abort	P_Abort	SWI	Undefined	Reset

Figure B-11 Vector catch register

B.10 Vector catching

The EmbeddedICE-RT logic contains hardware that enables efficient trapping of fetches from the vectors during exceptions. This is controlled by the vector catch register. If one of the bits in this register is set HIGH and the corresponding exception occurs, the processor enters debug state as if a breakpoint has been set on an instruction fetch from the relevant exception vector.

For example, if the processor executes a SWI instruction while bit 2 of the vector catch register is set, the processor fetches an instruction from location 0x8. The vector catch hardware detects this access and forces the internal **IBREAKPT** signal HIGH into the processor control logic. This, in turn, forces the processor to enter debug state.

The behavior of the hardware is independent of the watchpoint comparators, leaving them free for general use. The vector catch register is sensitive only to fetches from the vectors during exception entry. Therefore, if code branches to an address within the vectors during normal operation, and the corresponding bit in the vector catch register is set, the processor is not forced to enter debug state.

In monitor mode debug, vector catching is disabled on Data Aborts and Prefetch Aborts to avoid the processor being forced into an unrecoverable state as a result of the aborts that are generated for the monitor mode debug.

B.11 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.

B.11.1 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 ARM7EJ-S 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 ARM7EJ-S 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 ARM7EJ-S processor, 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 allows for quite complicated configurations of breakpoints and watchpoints.

——— **Note** ———

There is no **CHAIN** input to Watchpoint 1 and no **CHAIN** output from Watchpoint 0.

————————————

Take, for example, 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 set to off.

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.

B.11.2   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.

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.

B.12   Disabling EmbeddedICE-RT

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

When **DBGEN** is LOW:

*   **DBGEBKPT** is forced LOW to the core. (DBGRQ is the internal DBGRQ, which is a combination of the external input **EDBGRQ** and the debug control register bit 1 DBGRQ.)
*   **DBGACK** is forced LOW from the ARM7EJ-S processor.
*   Interrupts pass through to the processor uninhibited.

Copyright © 2001 ARM Limited. All rights reserved.

B.13   EmbeddedICE-RT timing

EmbeddedICE-RT samples the **DBGEXT[1]** and **DBGEXT[0]** inputs on the rising edge of **CLK**.

Refer to Chapter 10 *AC Parameters* for details of the required setup and hold times for these signals.

Glossary

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

Abort
A mechanism that indicates to a core that it should halt execution of an attempted illegal memory access. An abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction or data memory. An abort is classified as either a prefetch abort, a data abort, or an external abort.

See also Data abort, External abort, and Prefetch abort.

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

ALU
See Arithmetic Logic Unit.

Application Specific Integrated Circuit
An integrated circuit that has been designed to perform a specific application function. It can be custom-built or mass-produced.

Arithmetic Logic Unit
The part of a processor core that performs arithmetic and logic operations.

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

ASIC	*See* Application Specific Integrated Circuit.
Banked registers	Those physical registers whose use is defined by the current processor mode. The banked registers are R8 to R14.
Base register	A register specified by a load or store instruction that is used to hold the base value for the instruction's address calculation.
Big-endian	Byte ordering scheme in which bytes of decreasing significance in a data word are stored at increasing addresses in memory.
	See also Little-endian and Endianness.
Breakpoint	A breakpoint is a mechanism provided by debuggers to identify an instruction at which program execution is to be halted. Breakpoints are inserted by the programmer to enable inspection of register contents, memory locations, variable values at fixed points in the program execution to test that the program is operating correctly. Breakpoints are removed after the program is successfully tested.
	See also Watchpoint.
Byte	An 8-bit data item.
Central Processing Unit	The part of a processor that contains the ALU, the registers, and the instruction decode logic and control circuitry. Also commonly known as the processor core.
CISC	*See* Complex Instruction Set Computer.
Clock gating	Gating a clock signal for a macrocell with a control signal (such as **ETMPWRDOWN**) and using the modified clock that results to control the operating state of the macrocell.
Cold reset	Also known as power-on reset. Starting the processor by turning power on. Turning power off and then back on again clears memory and many internal settings. Some program failures lock up the processor and require a cold boot to use the system again. In other cases, only a warm reset is required.
	See also Warm reset.

Complex Instruction Set Computer

        The traditional architecture of a computer that uses microcode to execute very comprehensive instructions. Instructions can be variable in length and use all addressing modes, requiring complex circuitry to decode them.

        See also Reduced Instruction Set Computer.

Condition field	A 4-bit field in an instruction that is used to specify a condition under which the instruction can execute.

Content addressable memory

Memory that is identified by its contents. Content addressable memory is used in CAM-RAM architecture caches to store the tags for cache entries.

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.

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

Core reset *See* Warm reset.

CPU *See* Central Processing Unit.

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

See also Abort, External abort, and Prefetch abort.

Debug state A condition that allows the monitoring and control of the execution of a processor. It is usually used to find errors in the application program flow.

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

Digital Signal Processing

A category of techniques that analyze signals from sources such as sound, weather satellites, and earthquake monitors. Signals are converted into digital data and analyzed using various algorithms such as Fast Fourier Transform.

When a signal has been reduced to numbers, its components can be isolated, analyzed, and rearranged more easily than in analog form. DSP is used in many fields including biomedicine, sonar, radar, seismology, speech and music processing, imaging and communications.

Domain A collection of sections, large pages and small pages of memory, that can have their access permissions switched rapidly by writing to the Domain Access Control Register (CP15 register 3).

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

DSP *See* Digital Signal Processing.

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

Endianness
Byte ordering. The scheme that determines the order in which successive bytes of a data word are stored in memory.

See also Little-endian and Big-endian.

Exception vector
One of a number of fixed addresses in low memory, or in high memory if high vectors are configured, that contains the first instruction of the corresponding interrupt service routine.

External abort
An indication from an external memory system to a core that it should halt execution of an attempted illegal memory access. An external abort is caused by the external memory system as a result of attempting to access invalid memory.

See also Abort, Data abort, and Prefetch abort

Halfword
A 16-bit data item.

Instruction cycle count
The number of cycles for which an instruction occupies the Execute stage of the pipeline.

Java
Java is a platform independent object oriented programming language developed by Sun Microsystems. Java is designed to support multi-threaded programming.

See also Jazelle architecture.

Jazelle architecture
The ARM Jazelle architecture extends the Thumb and ARM operating states by adding a Java state (known as Jazelle state) to the processor. Instruction set support for entering and exiting Java applications, real-time interrupt handling, and debug support for mixed Java/ARM applications is present.

The processor has a new mode in which it behaves like a Java virtual machine. When in Jazelle state, the processor fetches and decodes Java byte codes and maintains the Java operand stack. The processor can switch, under operating system control, between Jazelle state and ARM state.

See also Java.

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. It is commonly known by the initials JTAG.

JTAG
See Joint Test Action Group.

Little-endian
Byte ordering scheme in which bytes of increasing significance in a data word are stored at increasing addresses in memory.

See also Big-endian and Endianness.

Macrocell
A complex logic block with a defined interface and behavior. A typical VLSI system will comprise several macrocells (such as a processor core, an ETM, and a memory block) plus application-specific logic.

Pipeline
A technique that provide simultaneous, or parallel, processing within the processor. It refers to overlapping operations by moving data or instructions into a conceptual pipe with all stages of the pipe processing simultaneously. For example, while one instruction is being executed, the processor is decoding the next instruction. In vector processors, several steps in a floating point operation can be processed simultaneously. The ARM7EJ-S processor uses a five-stage pipeline when in ARM and Thumb states, and uses a six-stage pipeline when in Jazelle state.

Power-on reset
See Cold reset.

Prefetch abort
An indication from a memory system to a core that it should halt execution of an attempted illegal memory access. A prefetch abort can be caused by the external or internal memory system as a result of attempting to access invalid instruction memory.

See also Data abort, External abort, and Abort.

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

Q flag
See Sticky overflow flag.

Reduced Instruction Set Computer
A computer architecture that reduces chip complexity by using simpler instructions. RISC compilers have to generate software routines to perform complex instructions that were previously done in hardware by CISC computers. In RISC, the microcode layer and associated overhead is eliminated.

RISC keeps instruction size constant, bans the indirect addressing mode and retains only those instructions that can be overlapped and made to execute in one machine cycle or less. The RISC chip is faster than its CISC counterpart and is designed and built more economically.

See also Complex Instruction Set Computer.

Region
A partition of instruction or data memory space.

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

RISC
See Reduced Instruction Set Computer.

SBO
See Should be one.

SBZ
See Should be zero.

SCREG	The currently selected scan chain number in an ARM TAP controller.
Should be one	Should be written as 1 (or all 1s for bit fields) by software. Writing a 0 will produce UNPREDICTABLE results.
Should be zero	Should be written as 0 (or all 0s for bit fields) by software. Writing a 1 will produce UNPREDICTABLE results.
Sticky overflow flag	Also known as Q flag. A flag that, when set to 1, is not affected by overflow during any other arithmetic instructions. The only instruction that can affect or be affected by the sticky overflow flag is MSR and MRS instructions.
Tag bits	The index or key field of a CAM entry.
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 **TRST**.
Thumb state	A processor that is executing Thumb (16-bit) half-word aligned instructions is operating in Thumb state.
UNDEFINED	An instruction that generates an undefined instruction exception.
UNPREDICTABLE	For reads, the data returned when reading from this location is unpredictable. It can have any value. For writes, writing to this location causes unpredictable behavior, or an unpredictable change in device configuration. UNPREDICTABLE instructions must not halt or hang the processor, or any part of the system.
Warm reset	Also known as core reset. Initializes the majority of the processor excluding the TAP controller and EmbeddedICE-RT Logic. This type of reset is useful if you are using the debugging features of the processor.
Watchpoint	A watchpoint is a mechanism provided by debuggers to halt program execution when the data contained by a particular memory address is changed. Watchpoints are inserted by the programmer to enable inspection of register contents, memory locations, and variable values when memory is written to test that the program is operating correctly. Watchpoints are removed after the program is successfully tested.
	See also Breakpoint.
Word	A 32-bit data item.

Index

Copyright © 2001 ARM Limited. All rights reserved.   ARM DDI 0214B