ARM7TDMI

(Rev 3)

Technical Reference Manual

ARM

ARM DDI 0029G

ARM7TDMI
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
October 1994	A	Released.
December 1994	B	First formal release.
December 1994	C	Review comments added.
March 1995	D	Technical changes.
August 1995	E	Review comments added.
November 2000	F	SGML, new layout, new title, incorporation of errata, and technical changes.
April 2001	G	Addition of timing parameters and editorial changes.

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM7TDMI Technical Reference Manual

List of Tables
ARM7TDMI Technical Reference Manual

List of Figures
ARM7TDMI Technical Reference Manual

Preface

This preface introduces the ARM7TDMI core and its reference documentation. It contains the following sections:

About this document

This document is a reference manual for the ARM7TDMI core.

Intended audience

This document has been written for experienced hardware and software engineers who are working with the ARM7TDMI processor.

Using this manual

This document is organized into the following chapters:

Chapter 1 ***Introduction***

Introduction to the architecture.

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

32-bit ARM and 16-bit Thumb instruction sets.

Chapter 3 ***Memory Interface***

Nonsequential, sequential, internal, and coprocessor register transfer memory cycles.

Chapter 4 ***Coprocessor Interface***

Implementation of the specialized additional instructions for use with coprocessors and a description of the interface.

Chapter 5 ***Debug Interface***

ARM7TDMI core hardware extensions for advanced debugging to make it simpler to develop application software, operating systems, and hardware.

Chapter 6 ***Instruction Cycle Timings***

Instruction cycle timings.

Chapter 7 ***AC and DC Parameters***

AC and DC parameters, timing diagrams, definitions, and operating data.

Appendix A ***Signal Description***

ARM7TDMI core signals.

Appendix B ***Debug in Depth***

Further information on the debug interface and EmbeddedICE macrocell.

Typographical conventions

The following typographical conventions are used in this book:

italic Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.

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

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

`type`writer Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

`typewriter italic`

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

`typewriter bold`

Denotes language keywords when used outside example code and ARM processor signal names.

Timing diagram conventions

The key provided in Figure P-1 explains the components used in timing diagrams. Any variations are labeled when they occur. Therefore, no additional meaning must be attached unless specifically stated.

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.

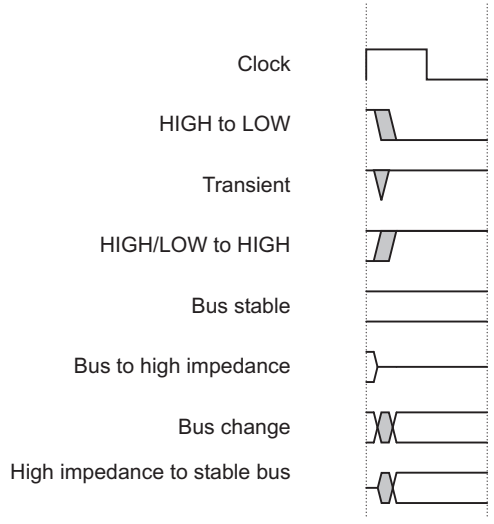

Figure P-1 Key to timing diagram conventions

Further reading

This section lists publications by ARM Limited and third parties.

ARM periodically provides updates and corrections to its documentation. For current errata sheets, addenda, and list of Frequently Asked Questions go to the ARM website:

`www.arm.com`

ARM publications

This document contains information that is specific to the ARM7TDMI core. Refer to the following documents for other relevant information:

- *ARM Architecture Reference Manual* (ARM DDI 0100).

Other publications

This section lists relevant documents published by third parties.

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

Feedback

ARM Limited welcomes feedback both on the ARM7TDMI core, and on the documentation.

Feedback on the ARM7TDMI core

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 this document

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

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

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the ARM7TDMI core. It contains the following sections:

1.1 About the ARM7TDMI core

The ARM7TDMI core is a member of the ARM family of general-purpose 32-bit microprocessors. The ARM family offers high performance for very low power consumption, and small size.

The ARM architecture is based on *Reduced Instruction Set Computer* (RISC) principles. The RISC 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.

This section describes:

- *The instruction pipeline* on page 1-2
- *Memory access* on page 1-3
- *Memory interface* on page 1-4.
- *EmbeddedICE Logic* on page 1-4.

1.1.1 The instruction pipeline

The ARM7TDMI core uses a pipeline to increase the speed of the flow of instructions to the processor. This allows several operations to take place simultaneously, and the processing and memory systems to operate continuously.

A three-stage pipeline is used, so instructions are executed in three stages:

- Fetch
- Decode
- Execute.

The instruction pipeline is shown in Figure 1-1 on page 1-3.

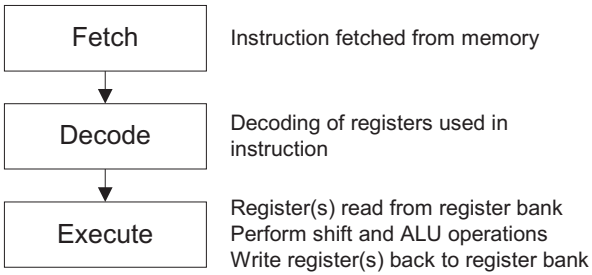

Figure 1-1 Instruction pipeline

During normal operation, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The program counter points to the instruction being fetched rather than to the instruction being executed. This is important because it means that the *Program Counter* (PC) value used in an executing instruction is always two instructions ahead of the address.

1.1.2 Memory access

The ARM7TDMI core 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.

Data can be:

- 8-bit (bytes)
- 16-bit (halfwords)
- 32-bit (words).

Words must be aligned to 4-byte boundaries. Halfwords must be aligned to 2-byte boundaries.

1.1.3 Memory interface

The ARM7TDMI processor memory interface has been designed to allow performance potential to be realized, while minimizing the use of memory. Speed-critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic. These control signals facilitate the exploitation of the fast-burst access modes supported by many on-chip and off-chip memory technologies.

The ARM7TDMI core has four basic types of memory cycle:

- idle cycle
- nonsequential cycle
- sequential cycle
- coprocessor register transfer cycle.

1.1.4 EmbeddedICE Logic

EmbeddedICE Logic is the additional hardware provided by debuggable ARM processors to aid debugging. It allows software tools to debug code running on a target processor. The EmbeddedICE Logic is controlled through the *Joint Test Action Group* (JTAG) test access port, using the EmbeddedICE interface. See Chapter 5 *Debug Interface* and Appendix B *Debug in Depth* for more information.

1.2 Architecture

The ARM7TDMI processor has two instruction sets:

- the 32-bit ARM instruction set
- the 16-bit Thumb instruction set.

The ARM7TDMI processor is an implementation of the ARMv4T architecture. For full details of both the ARM and Thumb instruction sets refer to the *ARM Architecture Reference Manual*.

This section describes:

- *Instruction compression* on page 1-5
- *The Thumb instruction set* on page 1-5.

1.2.1 Instruction compression

Microprocessor architectures traditionally had the same width for instructions, and data. Therefore 32-bit architectures had higher performance manipulating 32-bit data, and could address a large address space much more efficiently than 16-bit architectures.

16-bit architectures typically had higher code density than 32-bit architectures, but approximately half the performance.

Thumb implements a 16-bit instruction set on a 32-bit architecture to provide:

- higher performance than a 16-bit architecture
- higher code density than a 32-bit architecture.

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, allowing excellent interoperability between ARM and Thumb states.

On execution, 16-bit Thumb instructions are transparently decompressed to full 32-bit ARM instructions in real time, without performance loss.

Thumb 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.

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

Thumb code is typically 65% of the size of ARM code, and provides 160% of the performance of ARM code when running from a 16-bit memory system. Thumb, therefore, makes the ARM7TDMI core ideally suited to embedded applications with restricted memory bandwidth, where code density and footprint is important.

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 applications such as fast interrupts and DSP algorithms can be coded using the full ARM instruction set then linked with Thumb code.

1.3 Block, core, and functional diagrams

The ARM7TDMI processor architecture, core, and functional diagrams are illustrated in the following figures:

- Figure 1-2 shows a block diagram of the ARM7TDMI processor components and major signal paths
- Figure 1-3 on page 1-8 shows the main processor (this is the logic at the core of the ARM7TDMI
- Figure 1-4 on page 1-9 shows the major signal paths for the ARM7TDMI processor.

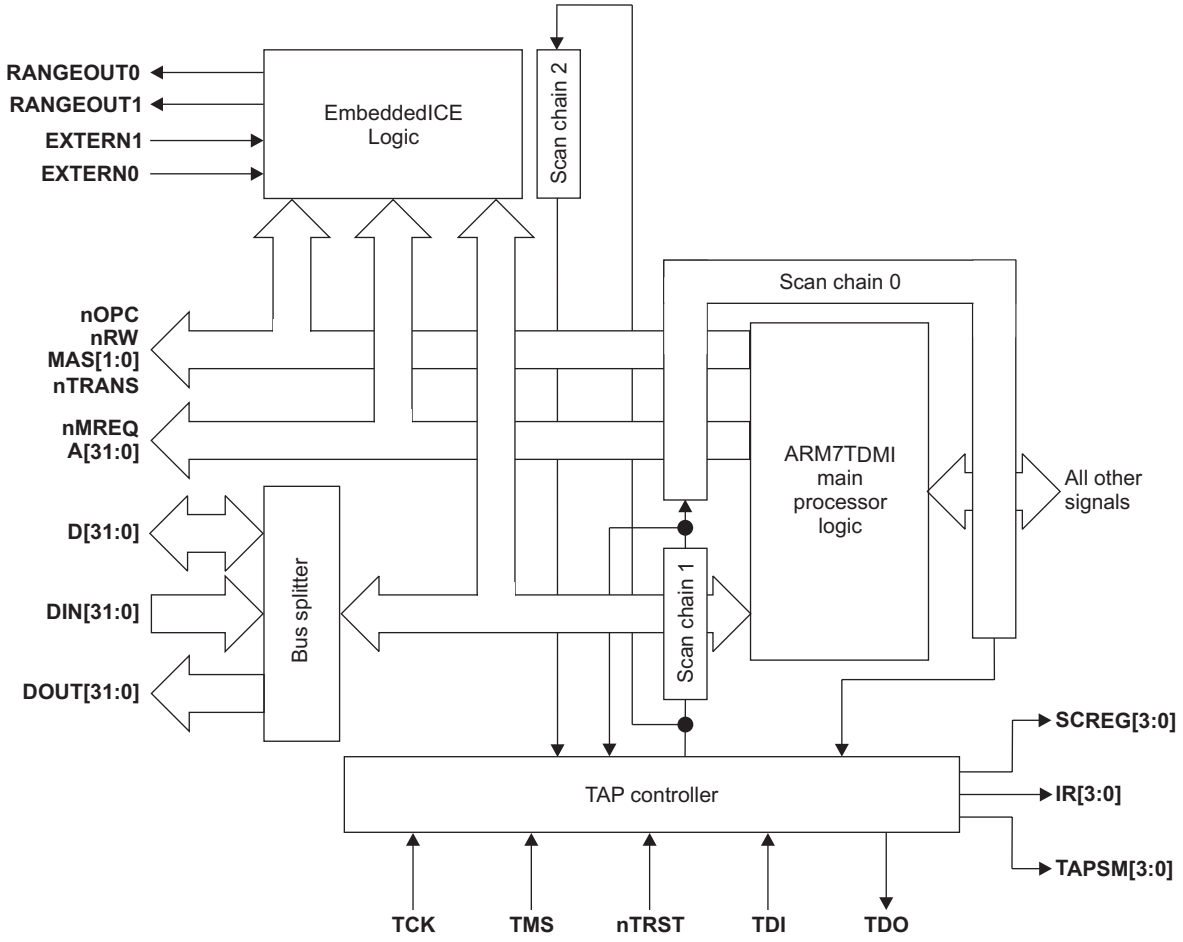

Figure 1-2 ARM7TDMI processor block diagram

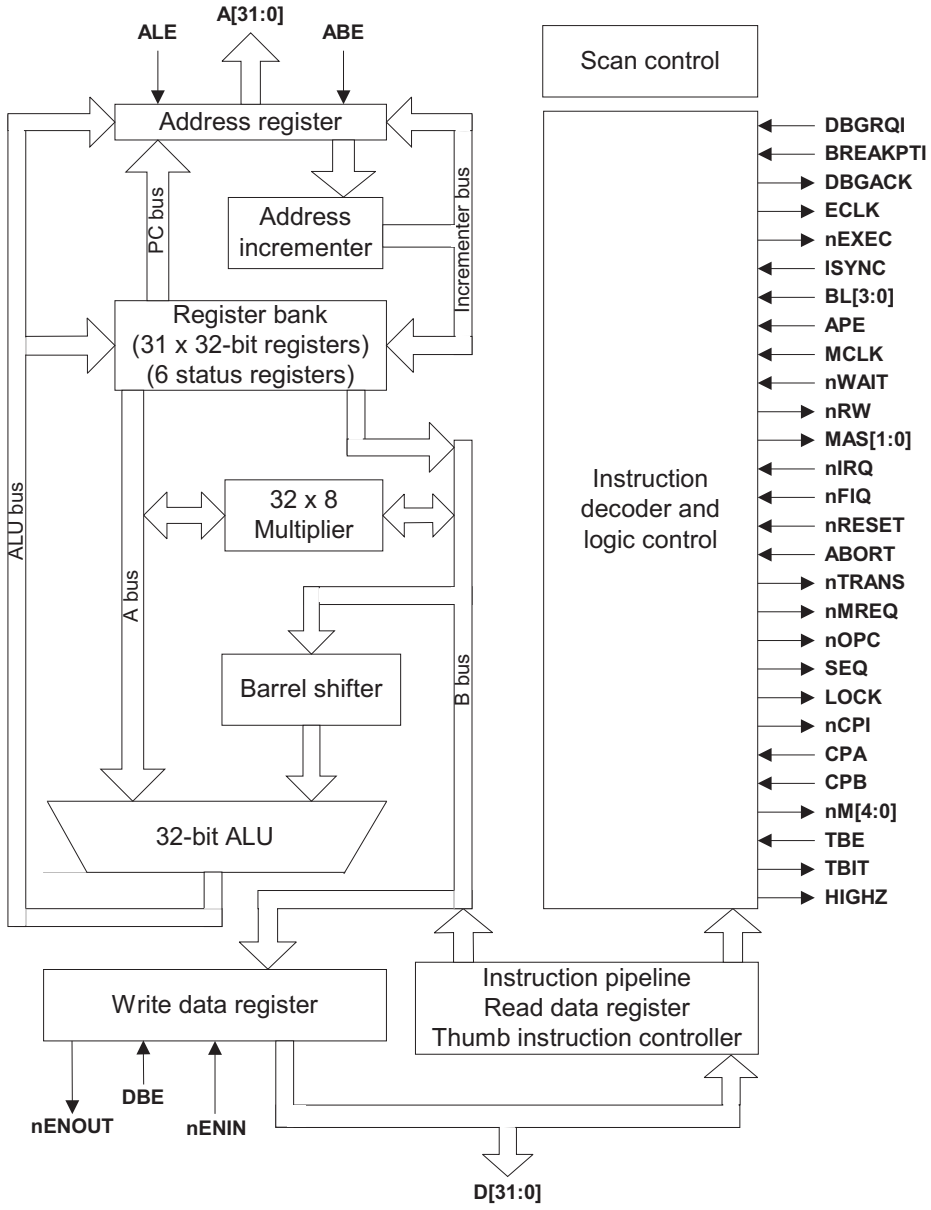

Figure 1-3 Main processor

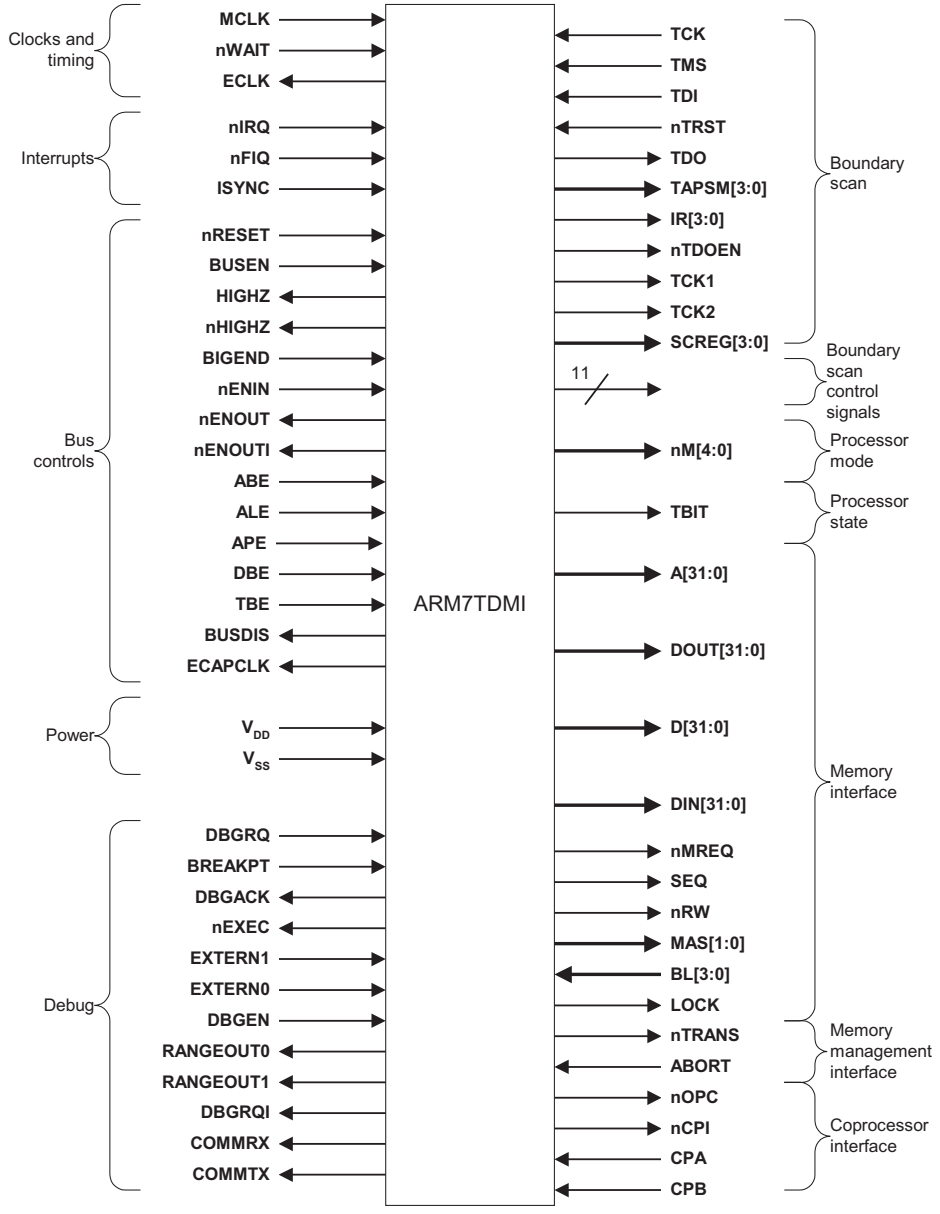

Figure 1-4 ARM7TDMI processor functional diagram

1.4 Instruction set summary

This section provides a description of the instruction sets used on the ARM7TDMI processor.

This section describes:

- *Format summary* on page 1-10
- *ARM instruction summary* on page 1-12
- *Thumb instruction summary* on page 1-19.

1.4.1 Format summary

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

- *ARM instruction summary* on page 1-12
- *Thumb instruction summary* on page 1-19.

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

The ARM7TDMI processor uses an implementation of the ARMv4T architecture. For a complete description of both instruction sets, refer to the *ARM Architecture Reference Manual*.

Table 1-1 Key to tables

Type	Description
`{cond}`	Condition field, see Table 1-6 on page 1-19.
`<Oprnd2>`	Operand2, see Table 1-4 on page 1-18.
`{field}`	Control 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 *Addressing modes* 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 }).

The ARM instruction set formats are shown in Figure 1-5 on page 1-11.

Refer to the *ARM Architectural Reference Manual* for more information about the ARM instruction set formats.

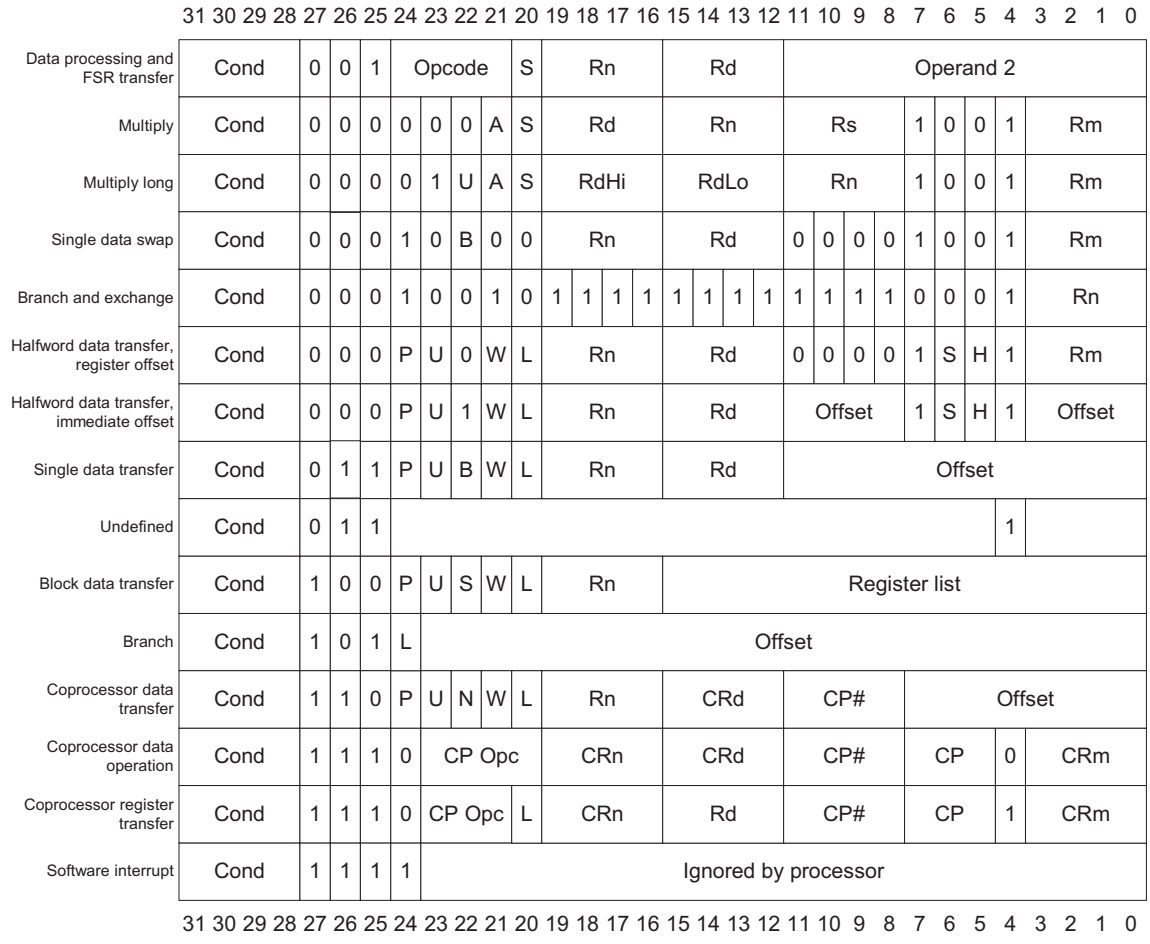

Figure 1-5 ARM instruction set formats

Note

Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken, for instance a multiply instruction with bit 6 changed to a 1. These instructions must not be used because their action might change in future ARM implementations. The behavior of these instruction codes on the ARM7TDMI processor is unpredictable.

1.4.2 ARM instruction summary

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

Table 1-2 ARM instruction summary

Operation		Assembly syntax
Move	Move	`MOV{cond}{S} Rd, <Oprnd2>`
	Move NOT	`MVN{cond}{S} Rd, <Oprnd2>`
	Move SPSR to register	`MRS{cond} Rd, SPSR`
	Move CPSR to register	`MRS{cond} Rd, CPSR`
	Move register to SPSR	`MSR{cond} SPSR{field}, Rm`
	Move register to CPSR	`MSR{cond} CPSR{field}, Rm`
	Move immediate to SPSR flags	`MSR{cond} SPSR_f, #32bit_Imm`
	Move immediate to CPSR flags	`MSR{cond} CPSR_f, #32bit_Imm`
Arithmetic	Add	`ADD{cond}{S} Rd, Rn, <Oprnd2>`
	Add with carry	`ADC{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract	`SUB{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract with carry	`SBC{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract reverse subtract	`RSB{cond}{S} Rd, Rn, <Oprnd2>`
	Subtract reverse subtract with carry	`RSC{cond}{S} Rd, Rn, <Oprnd2>`
	Multiply	`MUL{cond}{S} Rd, Rm, Rs`
	Multiply accumulate	`MLA{cond}{S} Rd, Rm, Rs, Rn`
	Multiply unsigned long	`UMULL{cond}{S} RdLo, RdHi, Rm, Rs`

Table 1-2 ARM instruction summary (continued)

Operation		Assembly syntax
	Multiply unsigned accumulate long	`UMLAL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply signed long	`SMULL{cond}{S} RdLo, RdHi, Rm, Rs`
	Multiply signed accumulate long	`SMLAL{cond}{S} RdLo, RdHi, Rm, Rs`
	Compare	`CMP{cond} Rd, <Oprnd2>`
	Compare negative	`CMN{cond} Rd, <Oprnd2>`
Logical	Test	`TST{cond} Rn, <Oprnd2>`
	Test equivalence	`TEQ{cond} Rn, <Oprnd2>`
	AND	`AND{cond}{S} Rd, Rn, <Oprnd2>`
	EOR	`EOR{cond}{S} Rd, Rn, <Oprnd2>`
	ORR	`ORR{cond}{S} Rd, Rn, <Oprnd2>`
	Bit clear	`BIC{cond}{S} Rd, Rn, <Oprnd2>`
Branch	Branch	`B{cond} label`
	Branch with link	`BL{cond} label`
	Branch and exchange instruction set	`BX{cond} Rn`
Load	Word	`LDR{cond} Rd, <a_mode2>`
	Word with user-mode privilege	`LDR{cond}T Rd, <a_mode2P>`
	Byte	`LDR{cond}B Rd, <a_mode2>`
	Byte with user-mode privilege	`LDR{cond}BT Rd, <a_mode2P>`
	Byte signed	`LDR{cond}SB Rd, <a_mode3>`
	Halfword	`LDR{cond}H Rd, <a_mode3>`
	Halfword signed	`LDR{cond}SH Rd, <a_mode3>`
	Multiple block data operations	-
	• Increment before	`LDM{cond}IB Rd{!}, <reglist>{^}`
	• Increment after	`LDM{cond}IA Rd{!}, <reglist>{^}`
	• Decrement before	`LDM{cond}DB Rd{!}, <reglist>{^}`

Table 1-2 ARM instruction summary (continued)

Operation		Assembly syntax
	• Decrement after	`LDM{cond}DA Rd{!}, <reglist>{^}`
	• Stack operation	`LDM{cond}<a_mode4L> Rd{!}, <reglist>`
	• Stack operation, and restore CPSR	`LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^`
	• Stack operation with user registers	`LDM{cond}<a_mode4L> Rd{!}, <reglist>^`
Store	Word	`STR{cond} Rd, <a_mode2>`
	Word with user-mode privilege	`STR{cond}T Rd, <a_mode2P>`
	Byte	`STR{cond}B Rd, <a_mode2>`
	Byte with user-mode privilege	`STR{cond}BT Rd, <a_mode2P>`
	Halfword	`STR{cond}H Rd, <a_mode3>`
	Multiple block data operations	-
	• Increment before	`STM{cond}IB Rd{!}, <reglist>{^}`
	• Increment after	`STM{cond}IA Rd{!}, <reglist>{^}`
	• Decrement before	`STM{cond}DB Rd{!}, <reglist>{^}`
	• Decrement after	`STM{cond}DA Rd{!}, <reglist>{^}`
	• Stack operation	`STM{cond}<a_mode4S> Rd{!}, <reglist>`
	• Stack operation with user registers	`STM{cond}<a_mode4S> Rd{!}, <reglist>^`
Swap	Word	`SWP{cond} Rd, Rm, [Rn]`
	Byte	`SWP{cond}B Rd, Rm, [Rn]`
Coprocessors	Data operation	`CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>`
	Move to ARM register 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>`
	Load	`LDC{cond} p<cpnum>, CRd, <a_mode5>`
	Store	`STC{cond} p<cpnum>, CRd, <a_mode5>`
Software interrupt		`SWI 24bit_Imm`

Addressing modes

The addressing modes are procedures shared by different instructions for generating values used by the instructions. The five addressing modes used by the ARM7TDMI 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 listed with their types and mnemonics 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	-

Table 1-3 Addressing modes (continued)

Addressing mode	Type or addressing mode	Mnemonic or stack type
	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 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]`

Table 1-3 Addressing modes (continued)

Addressing mode	Type or addressing mode	Mnemonic or stack type
Mode 3, <a_mode3>	Immediate offset	`[Rn, #+/-8bit_Offset]`
	Pre-indexed	`[Rn, #+/-8bit_Offset]!`
	Post-indexed	`[Rn], #+/-8bit_Offset`
	Register	`[Rn, +/-Rm]`
	Pre-indexed	`[Rn, +/-Rm]!`
	Post-indexed	`[Rn], +/-Rm`
Mode 4, load <a_mode4L>	IA, increment after	FD, full descending
	IB, increment before	ED, empty descending
	DA, decrement after	FA, full ascending
	DB decrement before	EA, empty ascending
Mode 4, store <a_mode4S>	IA, increment after	FD, full descending
	IB, increment before	ED, empty descending
	DA, decrement after	FA, full ascending
	DB decrement before	EA, empty ascending
Mode 5, coprocessor data transfer <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. Operand 2 is listed in Table 1-4.

Table 1-4 Operand 2

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 listed in Table 1-5.

Table 1-5 Fields

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

Condition fields

Condition fields are listed 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
	CS	Unsigned higher, or same	C set
	CC	Unsigned lower	C clear
	MI	Negative	N set
	PL	Positive, or zero	N clear
	VS	Overflow	V set
	VC	No overflow	V clear
	HI	Unsigned higher	C set, Z clear
	LS	Unsigned lower, or same	C clear, Z set
	GE	Greater, or equal	N=V (N and V set or N and V clear)
	LT	Less than	N<>V (N set and V clear) or (N clear and V set)
	GT	Greater than	Z clear, N=V (N and V set or N and V clear)
	LE	Less than, or equal	Z set or N<>V (N set and V clear) or (N clear and V set)
	AL	Always	Flag ignored

1.4.3 Thumb instruction summary

The Thumb instruction set formats are shown in Figure 1-6 on page 1-20.

Refer to the *ARM Architectural Reference Manual* for more information about the ARM instruction set formats.

	Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move shifted register	01	0	0	0	Op		Offset5					Rs			Rd		
Add and subtract	02	0	0	0	1	1	1	Op	Rn/ offset3			Rs			Rd		
Move, compare, add, and subtract immediate	03	0	0	1	Op		Rd			Offset8							
ALU operation	04	0	1	0	0	0	0	Op				Rs			Rd		
High register operations and branch exchange	05	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			RdHd		
PC-relative load	06	0	1	0	0	1	Rd			Word8							
Load and store with relative offset	07	0	1	0	1	L	B	0	Ro			Rb			Rd		
Load and store sign-extended byte and halfword	08	0	1	0	1	H	S	1	Ro			Rb			Rd		
Load and store with immediate offset	09	0	1	1	B	L	Offset5					Rb			Rd		
Load and store halfword	10	1	0	0	0	L	Offset5					Rb			Rd		
SP-relative load and store	11	1	0	0	1	L	Rd			Word8							
Load address	12	1	0	1	0	SP	Rd			Word8							
Add offset to stack pointer	13	1	0	1	1	0	0	0	0	S	SWord7						
Push and pop registers	14	1	0	1	1	L	1	0	R	Rlist							
Multiple load and store	15	1	1	0	0	L	Rb			Rlist							
Conditional branch	16	1	1	0	1	Cond				Softset8							
Software interrupt	17	1	1	0	1	1	1	1	1	Value8							
Unconditional branch	18	1	1	1	0	0	Offset11										
Long branch with link	19	1	1	1	1	H	Offset										
	Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 1-6 Thumb instruction set formats

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

Table 1-7 Thumb instruction set summary

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

Table 1-7 Thumb instruction set summary (continued)

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

Table 1-7 Thumb instruction set summary (continued)

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

Table 1-7 Thumb instruction set summary (continued)

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

Chapter 2
Programmer's Model

This chapter describes the ARM7TDMI core programmer's model. It contains the following sections:

2.1 About the programmer's model

The ARM7TDMI processor core implements ARM architecture v4T, which includes the 32-bit ARM instruction set, and the 16-bit Thumb instruction set. The programmer's model is described in the *ARM Architecture Reference Manual*.

2.2 Processor operating states

The ARM7TDMI processor has two operating states:

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

Thumb 16-bit, halfword-aligned Thumb instructions are executed in this state.

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

Note

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

2.2.1 Switching state

The operating state of the ARM7TDMI core can be switched between ARM state and Thumb state using the BX instruction. This is described in the *ARM Architecture Reference Manual*.

All exception handling is entered in ARM state. If an exception occurs in Thumb state, the processor reverts to ARM state. The transition back to Thumb state occurs automatically on return. An exception handler can change to Thumb state but it must return to ARM state to allow the exception handler to terminate correctly.

2.3 Memory formats

The ARM7TDMI processor views memory as a linear collection of bytes numbered in ascending order from zero. For example:

- bytes zero to three hold the first stored word
- bytes four to seven hold the second stored word.

The ARM7TDMI processor is bi-endian and can treat words in memory as being stored in either:

- *Little-endian* on page 2-4.
- *Big-Endian* on page 2-5

Note

Little-endian is traditionally the default format for ARM processors.

The endian format of a CPU dictates where the most significant byte or digits must be placed in a word. Because numbers are calculated by the CPU starting with the least significant digits, little-endian numbers are already set up for the processing order.

Endian configuration has no relevance unless data is stored as words and then accessed in smaller sized quantities (halfwords or bytes).

2.3.1 Little-endian

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.

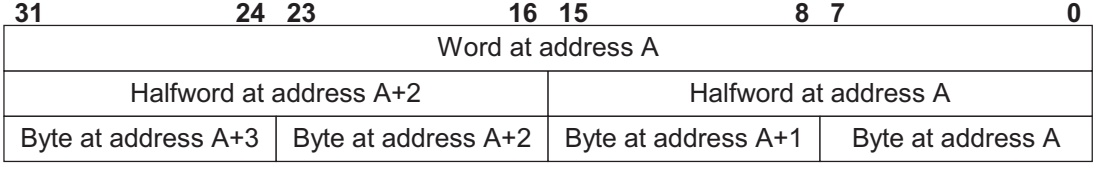

Figure 2-1 LIttle-endian addresses of bytes and halfwords within words

2.3.2 Big-Endian

In big-endian format, the ARM7TDMI 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 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.

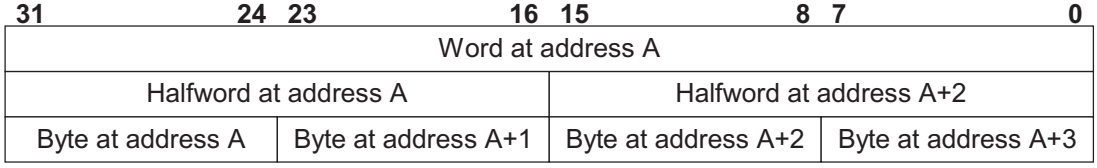

Figure 2-2 Big-endian addresses of bytes and halfwords within words

2.4 Data types

The ARM7TDMI processor supports the following data types:

- words, 32-bit
- halfwords, 16-bit
- bytes, 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.

Note

Memory systems are expected to support all data types. In particular, the system must support subword writes without corrupting neighboring bytes in that word.

2.5 Operating modes

The ARM7TDMI processor has 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 supports a data transfer or channel process.
- *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.

  Note

  You can only enter System mode from another privileged mode by modifying the mode bit of the *Current Program Status Register* (CPSR).

- Undefined mode is entered when an undefined instruction is executed.

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 a mode identifier as listed in Table 2-1.

Table 2-1 Register mode identifiers

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

2.6 Registers

The ARM7TDMI processor has a total of 37 registers:

- 31 general-purpose 32-bit registers
- 6 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.

2.6.1 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 2-10 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 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 and r15 have the following special functions:

Link register Register 14 is used as the subroutine *Link Register* (LR).

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

At all other times you can treat r14 as a general-purpose register. The corresponding banked registers r14_svc, r14_irq, r14_fiq, r14_abt and r14_und are similarly used to hold the return values of r15 when interrupts and exceptions arise, or when BL instructions are executed within interrupt or exception routines.

Program counter Register 15 holds the PC.

In ARM state, bits [1:0] of r15 are undefined and must be ignored. Bits [31:2] contain the PC.

In Thumb state, bit [0] is undefined and must be ignored. Bits [31:1] contain the PC.

By convention, r13 is used as the *Stack Pointer* (SP).

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 which 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.

FIQ mode has seven banked registers mapped to r8–r14 (r8_fiq–r14_fiq).

In ARM state, many FIQ handlers do not have to save any registers.

The User, IRQ, Supervisor, Abort, and undefined modes each have two banked registers mapped to r13 and r14, allowing a private SP and LR for each mode.

System mode shares the same registers as User mode.

Figure 2-3 shows the ARM-state registers.

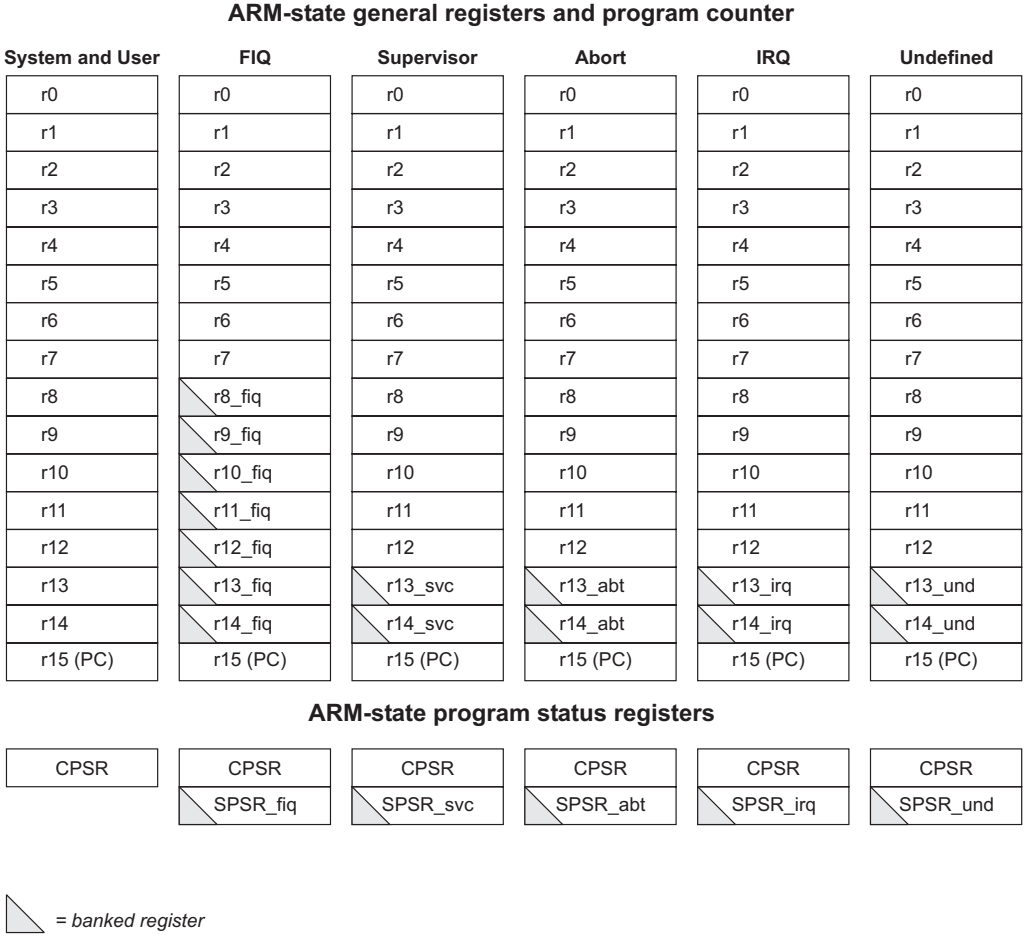

Figure 2-3 Register organization in ARM state

2.6.2 The Thumb-state register set

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

- 8 general registers, r0–r7
- the PC
- the SP
- the LR
- the CPSR.

There are banked SPs, LRs, and SPSRs for each privileged mode. This register set is shown in Figure 2-4.

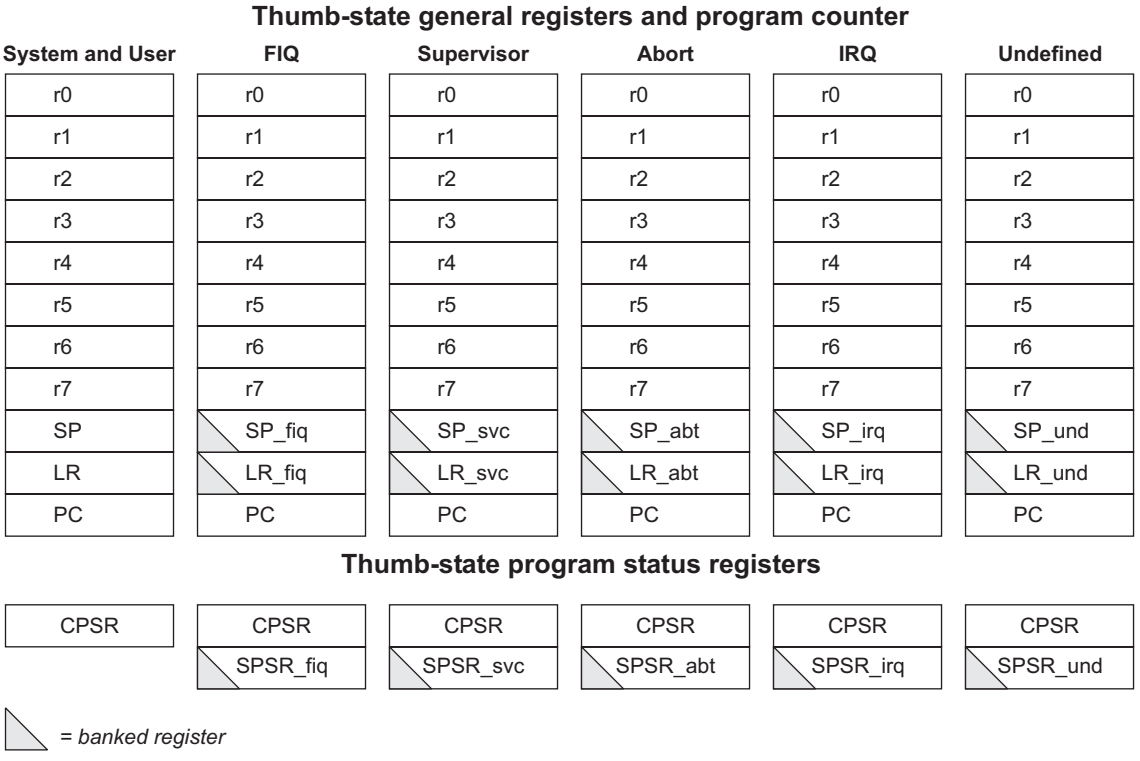

Figure 2-4 Register organization in Thumb state

2.6.3 The relationship between ARM-state and Thumb-state registers

The Thumb-state registers relate to the ARM-state registers in the following way:

- Thumb-state r0–r7 and ARM-state r0–r7 are identical
- Thumb-state CPSR and SPSRs and ARM-state CPSR and SPSRs are identical
- Thumb-state SP maps onto the ARM-state r13
- Thumb-state LR maps onto the ARM-state r14
- the Thumb-state PC maps onto the ARM-state PC (r15).

These relationships are shown in Figure 2-5.

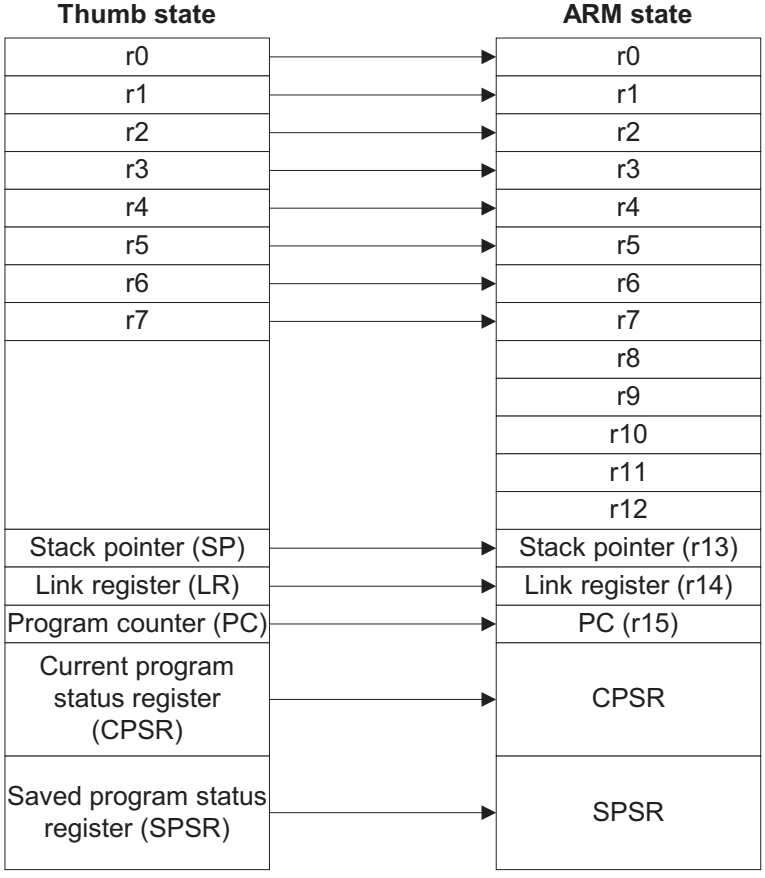

Figure 2-5 Mapping of Thumb-state registers onto ARM-state registers

Note

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

2.6.4 Accessing high registers in Thumb state

In Thumb state, the high registers, r8–r15, are not part of the standard register set. The assembly language programmer has 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 allows 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, please refer to the *ARM Architecture Reference Manual*.

2.7 The program status registers

The ARM7TDMI processor contains a 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 is shown in Figure 2-6.

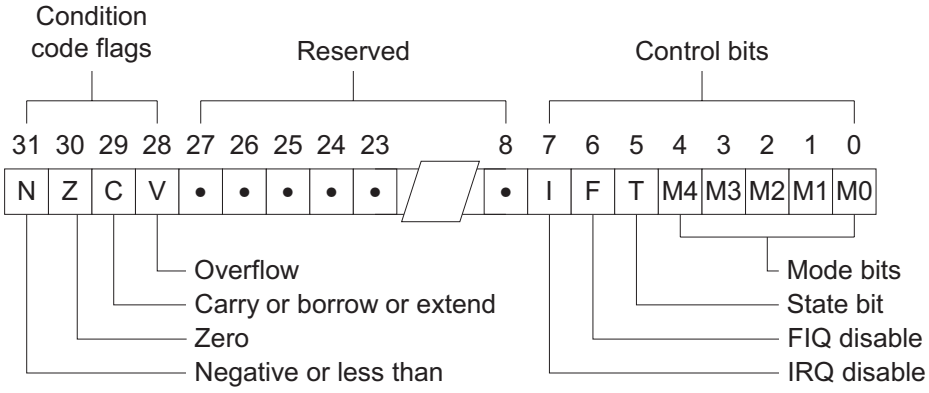

Figure 2-6 Program status register format

Note

To maintain compatibility with future ARM processors, you must not alter any of the reserved bits. One method of preserving these bits is to use a read-write-modify strategy when changing the CPSR.

The remainder of this section describes:

- *Condition code flags* on page 2-13
- *Control bits* on page 2-14
- *Reserved bits* on page 2-15.

2.7.1 Condition code flags

The N, Z, C, and V bits are the condition code flags, you can set them by arithmetic and logical operations. They can also be set by MSR and LDM instructions. The ARM7TDMI processor tests these flags to determine whether to execute an instruction.

All instructions can execute conditionally 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*.

2.7.2 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.

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

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 executing in ARM state.

The operating state is reflected on the external signal **TBIT**.

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.

Mode bits

Bits M[4:0] determine the processor operating mode as shown in Table 2-2. Not all combinations of the mode bits define a valid processor mode, so take care to use only the bit combinations shown.

Table 2-2 PSR mode bit values

M[4:0]	Mode	Visible Thumb-state registers	Visible ARM-state registers
10000	User	r0–r7, SP, LR, PC, CPSR	r0–r14, PC, CPSR
10001	FIQ	r0–r7, SP_fiq, LR_fiq, PC, CPSR, SPSR_fiq	r0–r7, r8_fiq–r14_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	r0–r7, SP_irq, LR_irq, PC, CPSR, SPSR_irq	r0–r12, r13_irq, r14_irq, PC, CPSR, SPSR_irq
10011	Supervisor	r0–r7, SP_svc, LR_svc, PC, CPSR, SPSR_svc	r0–r12, r13_svc, r14_svc, PC, CPSR, SPSR_svc
10111	Abort	r0–r7, SP_abt, LR_abt, PC, CPSR, SPSR_abt	r0–r12, r13_abt, r14_abt, PC, CPSR, SPSR_abt
11011	Undefined	r0–r7, SP_und, LR_und, PC, CPSR, SPSR_und	r0–r12, r13_und, r14_und, PC, CPSR, SPSR_und
11111	System	r0–r7, SP, LR, PC, CPSR	r0–r14, PC, CPSR

An illegal value programmed into M[4:0] causes the processor to enter an unrecoverable state. If this occurs, apply reset.

2.7.3 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. Also, make sure that your program does not rely on reserved bits containing specific values because future processors might have these bits set to 1 or 0.

2.8 Exceptions

Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example, to service an interrupt from a peripheral. Before attempting to handle an exception, the ARM7TDMI 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 Table 2-3.

This section provides details of the ARM7TDMI processor exception handling:

- *Exception entry and exit summary* on page 2-16
- *Entering an exception* on page 2-17
- *Leaving an exception* on page 2-18
- *Fast interrupt request* on page 2-18
- *Interrupt request* on page 2-19
- *Software interrupt instruction* on page 2-21
- *Undefined instruction* on page 2-21
- *Exception vectors* on page 2-21
- *Exception priorities* on page 2-22.

2.8.1 Exception entry and exit summary

Table 2-3 summarizes the PC value preserved in the relevant r14 on exception entry, and the recommended instruction for exiting the exception handler.

Table 2-3 Exception entry and exit

Exception or entry	Return instruction	Previous state ARM r14_x	Thumb r14_x	Remarks
BL	`MOV PC, R14`	PC+4	PC+2	Where PC is the address of the BL, SWI, or undefined instruction fetch that had the Prefetch Abort
SWI	`MOVS PC, R14_svc`	PC+4	PC+2	
UDEF	`MOVS PC, R14_und`	PC+4	PC+2	
PABT	`SUBS PC, R14_abt, #4`	PC+4	PC+4	

Table 2-3 Exception entry and exit (continued)

Exception or entry	Return instruction	Previous state ARM r14_x Thumb r14_x		Remarks
FIQ	`SUBS PC, R14_fiq, #4`	PC+4	PC+4	Where 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	
DABT	`SUBS PC, R14_abt, #8`	PC+8	PC+8	Where PC is the address of the Load or Store instruction that generated the Data Abort
RESET	Not applicable	-	-	The value saved in r14_svc upon reset is unpredictable

2.8.2 Entering an exception

The ARM7TDMI processor handles an exception as follows:

1. Preserves the address of the next instruction in the appropriate LR.

   When the exception entry is from ARM state, the ARM7TDMI processor copies the address of the next instruction into the LR, current PC+4 or PC+8 depending on the exception.

   When the exception entry is from Thumb state, the ARM7TDMI 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 that depends on the exception.

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

The ARM7TDMI processor can also set the interrupt disable flags to prevent otherwise unmanageable nestings of exceptions.

Note

Exceptions are always entered in ARM state. When the processor is in Thumb state and an exception occurs, the switch to ARM state takes place automatically when the exception vector address is loaded into the PC. An exception handler might change to Thumb state but it must return to ARM state to allow the exception handler to terminate correctly.

2.8.3 Leaving an exception

When an exception is completed, the exception handler must:

1. 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 2-16.
2. Copy the SPSR back to the CPSR.
3. Clear the interrupt disable flags that were set on entry.

Note

The action of restoring the CPSR from the SPSR automatically resets the T bit to whatever value it held immediately prior to the exception.

2.8.4 Fast interrupt request

The *Fast Interrupt Request* (FIQ) exception supports data transfers or channel processes. In ARM state, FIQ mode has eight banked registers to remove the requirement for register saving. This minimizes the overhead of context switching.

An FIQ is externally generated by taking the **nFIQ** input LOW. The input passes into the core through a synchronizer.

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

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

FIQ exceptions can be disabled within a privileged mode by setting the CPSR F flag. When the F flag is clear, the ARM7TDMI processor checks for a LOW level on the output of the FIQ synchronizer at the end of each instruction.

2.8.5 Interrupt request

The *Interrupt Request* (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. As with the **nFIQ** input, **nIRQ** passes into the core through a synchronizer.

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

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

You can disable IRQ at any time, by setting the I bit in the CPSR from a privileged mode.

2.8.6 Abort

An abort indicates that the current memory access cannot be completed. An abort is signaled by the external ABORT input. The ARM7TDMI processor checks for the abort exception at the end of memory access cycles.

The abort mechanism allows the implementation of 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 and make the requested data available.
- Load the instruction that caused the abort using an `LDR Rn,[R14_abt,#-8]` instruction to determine whether that instruction specifies base register write-back. If it does, the abort handler must then:
  - — determine from the instruction what the offset applied to the base register by the write-back was
  - — apply the opposite offset to the value that will be reloaded into the base register when the abort handler returns.

  This ensures that when the instruction is retried, the base register will have been restored to the value it had when the instruction was originally executed.

The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

There are two types of abort:

- a Prefetch Abort occurs during an instruction prefetch
- a Data Abort occurs during a data access.

Prefetch Abort

When a Prefetch Abort occurs, the ARM7TDMI 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 it fails its condition codes or because a branch occurs while it is in the pipeline, the abort does not take place.

After dealing with the reason for the abort, the handler executes 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

When a Data Abort occurs, the action taken depends on the instruction type:

- Single data transfer instructions (LDR and STR). If write back base register is specified by the instruction then the abort handler must be aware of this. In the case of a load instruction the ARM7TDMI processor prevents overwriting of the destination register with the loaded data.
- Swap instruction (SWP):
  - on a read access suppresses the write access and the write to the destination register
  - on a write access suppresses the write to the destination register.
- Block data transfer instructions (LDM and STM) complete. When write-back is specified, the base register is updated.

  If the base register is in the transfer list and has already been overwritten with loaded data by the time that the abort is indicated then the base register reverts to the original value. The ARM7TDMI processor prevents all register overwriting with loaded data after an abort is indicated. This means that the final value of the base register is always the written-back value, if write-back is specified, at its original value. It also means that the ARM7TDMI core always preserves r15 in an aborted LDM instruction, because r15 is always either the last register in the transfer list or not present in the transfer list.

After fixing the reason for 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.

2.8.7 Software interrupt instruction

The *Software Interrupt instruction* (SWI) is used to enter Supervisor mode, usually to request a particular supervisor function. The SWI handler reads the opcode to extract the SWI function number.

A SWI handler returns by executing the following 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.

2.8.8 Undefined instruction

When the ARM7TDMI processor encounters an instruction that neither it, nor any coprocessor in the system can handle, the ARM7TDMI core 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 executes the following 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.

For more information about undefined instructions, see the *ARM Architecture Reference Manual*.

2.8.9 Exception vectors

Table 2-4 lists the exception vector addresses. In this table, I and F represent the previous value of the IRQ and FIQ interrupt disable bits respectively in the CPSR.

Table 2-4 Exception vectors

Address	Exception	Mode on entry	I state on entry	F state on entry
0x00000000	Reset	Supervisor	Set	Set
0x00000004	Undefined instruction	Undefined	Set	Unchanged
0x00000008	Software interrupt	Supervisor	Set	Unchanged
0x0000000C	Prefetch Abort	Abort	Set	Unchanged

Table 2-4 Exception vectors (continued)

Address	Exception	Mode on entry	I state on entry	F state on entry
0x00000010	Data Abort	Abort	Set	Unchanged
0x00000014	Reserved	Reserved	-	-
0x00000018	IRQ	IRQ	Set	Unchanged
0x0000001C	FIQ	FIQ	Set	Set

2.8.10 Exception priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they are handled. The priority order is listed in Table 2-5.

Table 2-5 Exception priority order

Priority	Exception
Highest	Reset
	Data Abort
	FIQ
	IRQ
	Prefetch Abort
Lowest	Undefined instruction and SWI

Some exceptions cannot occur together:

- The 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 ARM7TDMI 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.

2.9 Interrupt latencies

The calculations for maximum and minimum latency are described in:

- *Maximum interrupt latencies* on page 2-23
- *Minimum interrupt latencies* on page 2-23.

2.9.1 Maximum interrupt latencies

When FIQs are enabled, the worst-case latency for FIQ comprises a combination of:

- The longest time the request can take to pass through the synchronizer, $T_{syncmax}$ (four processor cycles).
- The time for the longest instruction to complete, T_{ldm}. The longest instruction, is an LDM which loads all the registers including the PC. T_{ldm} is 20 cycles in a zero wait state system.
- The time for the Data Abort entry, T_{exc} (three cycles).
- The time for FIQ entry, T_{fiq} (two cycles).

The total latency is therefore 29 processor cycles, just over 0.7 microseconds in a system that uses a continuous 40MHz processor clock. At the end of this time, the ARM7TDMI processor executes the instruction at `0x1c`.

The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ, having higher priority, can delay entry into the IRQ handling routine for an arbitrary length of time.

2.9.2 Minimum interrupt latencies

The minimum latency for FIQ or IRQ is the shortest time the request can take through the synchronizer, $T_{syncmin}$, plus T_{fiq}, a total of five processor cycles.

2.10 Reset

When the **nRESET** signal goes LOW a reset occurs, and the ARM7TDMI core abandons the executing instruction and continues to increment the address bus as if still fetching word or halfword instructions. **nMREQ** and **SEQ** indicates internal cycles during this time.

When **nRESET** goes HIGH again, the ARM7TDMI processor:

1. Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The values of the PC and CPSR are indeterminate.
2. Forces **M[4:0]** to b10011, Supervisor mode, sets the I and F bits, and clears the T-bit in the CPSR.
3. Forces the PC to fetch the next instruction from address `0x00`.
4. Reverts to ARM state if necessary and resumes execution.

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

More information is provided in *Reset sequence after power up* on page 3-33.

Chapter 3
Memory Interface

This chapter describes the ARM7TDMI processor memory interface. It contains the following sections:

3.1 About the memory interface

The ARM7TDMI 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.

3.2 Bus interface signals

The signals in the ARM7TDMI 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:

- **MCLK**
- **nWAIT**
- **ECLK**
- **nRESET**.

The address class signals are:

- **A[31:0]**
- **nRW**
- **MAS[1:0]**
- **nOPC**
- **nTRANS**
- **LOCK**
- **TBIT**.

The memory request signals are:

- **nMREQ**
- **SEQ**.

The data timed signals are:

- **D[31:0]**
- **DIN[31:0]**
- **DOUT[31:0]**
- **ABORT**
- **BL[3:0]**.

The ARM7TDMI processor uses both the rising and falling edges of **MCLK**.

Bus cycles can be extended using the **nWAIT** signal. This signal is described in *Stretching access times* on page 3-29. All other sections of this chapter describe a simple system in which **nWAIT** is permanently HIGH.

3.3 Bus cycle types

The ARM7TDMI processor bus interface is pipelined. This gives the maximum time for a memory cycle to decode the address and respond to the access request:

- memory request signals are broadcast in the bus cycle ahead of the bus cycle to which they refer
- address class signals are broadcast half a clock cycle ahead of the bus cycle to which they refer.

A single memory cycle is shown in Figure 3-1.

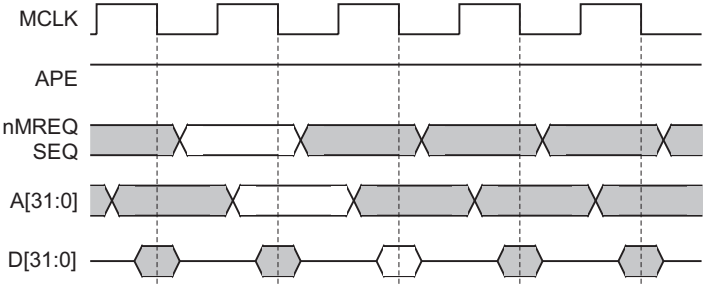

Figure 3-1 Simple memory cycle

The ARM7TDMI processor bus interface can perform four different types of bus cycle:

- a nonsequential cycle requests a transfer to or from an address which is unrelated to the address used in the preceding cycle
- a sequential cycle requests a transfer to or from an address which is either the same, one word, or one halfword greater than the address used in the preceding cycle
- an internal cycle does not require a transfer because it is performing an internal function, and no useful prefetching can be performed at the same time
- a coprocessor register transfer cycle uses the data bus to communicate with a coprocessor, but does not require any action by the memory system.

Bus cycle types are encoded on the **nMREQ** and **SEQ** signals as listed in Table 3-1.

Table 3-1 Bus cycle types

nMREQ	SEQ	Bus cycle type	Description
0	0	N-cycle	Nonsequential cycle
0	1	S-cycle	Sequential cycle
1	0	I-cycle	Internal cycle
1	1	C-cycle	Coprocessor register transfer cycle

A memory controller for the ARM7TDMI processor must commit to a memory access only on an N-cycle or an S-cycle.

3.3.1 Nonsequential cycles

A nonsequential cycle is the simplest form of bus cycle, and occurs when the 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 and (**nMREQ** and **SEQ**) signals that comprise an N-cycle are broadcast on the bus. At the end of the next bus cycle the data is transferred between the CPU and the memory. It is not uncommon for a memory system to require a longer access time (extending the clock cycle) for nonsequential accesses. This is to allow time for full address decoding or to latch both a row and column address into DRAM. This is illustrated in Figure 3-2 on page 3-6.

Note

In Figure 3-2 on page 3-6, **nMREQ** and **SEQ** are highlighted where they are valid to indicate the N-cycle.

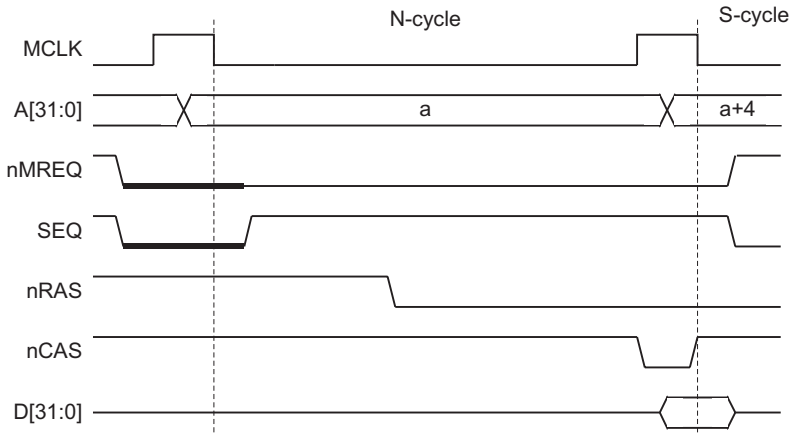

Figure 3-2 Nonsequential memory cycle

The ARM7TDMI processor can perform back-to-back, nonsequential memory cycles. This happens, for example, when an `STR` instruction is executed. If you are designing a memory controller for the ARM7TDMI core, and your memory system is unable to cope with this case, use the **nWAIT** signal to extend the bus cycle to allow sufficient cycles for the memory system. See *Stretching access times* on page 3-29.

3.3.2 Sequential cycles

Sequential cycles are used to perform burst transfers on the bus. This information can be used to optimize the design of a memory controller interfacing to a burst memory device, such as a DRAM.

During a sequential cycle, the ARM7TDMI processor requests a memory location that is part of a sequential burst. For 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 IS-cycle (see *Nonsequential cycles* on page 3-5), and continues with S-cycles. A burst comprises transfers of the same type. The **A[31:0]** signal increments during the burst. The other address class signals are unaffected by a burst.

The possible burst types are listed in Table 3-2.

Table 3-2 Burst types

Burst type	Address increment	Cause
Word read	4 bytes	ARM7TDMIcore 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 data width, direction, and protection type. For more details, see *Addressing signals* on page 3-11.

Memory systems can often respond faster to a sequential access and can require a shorter access time compared to a nonsequential access. An example of a burst access is shown in Figure 3-3.

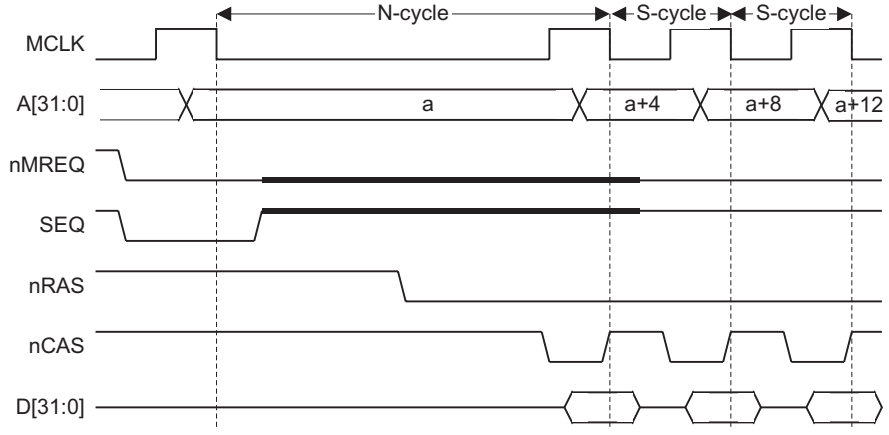

Figure 3-3 Sequential access cycles

3.3.3 Internal cycles

During an internal cycle, the ARM7TDMI processor does not require a memory access, as an internal function is being performed, and no useful prefetching can be performed at the same time.

Where possible the ARM7TDMI 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 shown in Figure 3-4 and, is further described in *Nonsequential memory cycle* on page 3-6.

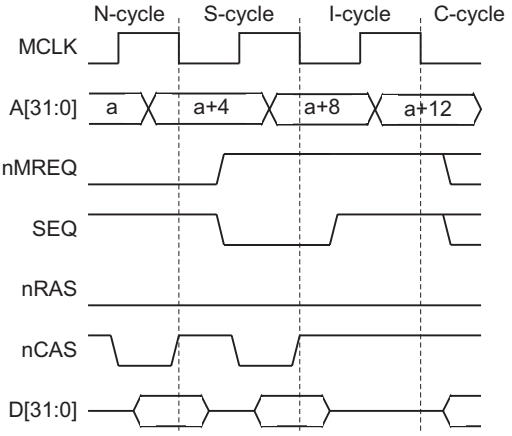

Figure 3-4 Internal cycles

3.3.4 Merged IS cycles

Where possible, the ARM7TDMI 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 on this bus during an internal cycle. This enables the memory controller to decode the address, but it must not initiate a memory access during this cycle. In a merged IS 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 on page 3-9.

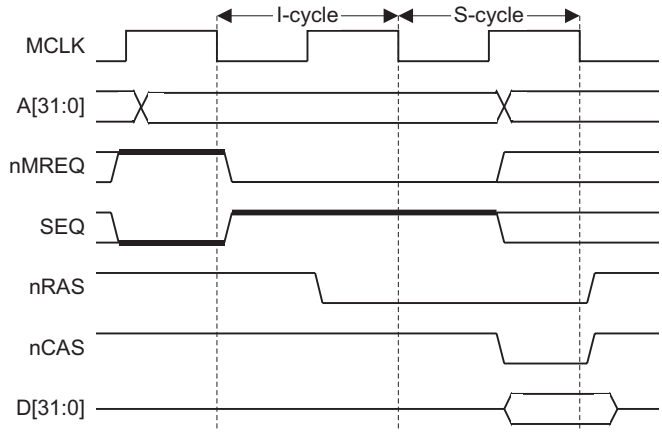

Figure 3-5 Merged IS cycle

Note

When designing a memory controller, ensure that the design also works when an I-cycle is followed by an N-cycle to a different address. This sequence can 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.

3.3.5 Coprocessor register transfer cycles

During a coprocessor register transfer cycle, the ARM7TDMI processor uses the data buses to transfer data to or from a coprocessor. A memory cycle is not required and the memory controller does not initiate a transaction. The memory system must not drive onto the data bus during a coprocessor register transfer cycle.

The coprocessor interface is described in Chapter 4 *Coprocessor Interface*. The coprocessor register transfer cycle is shown in Figure 3-6 on page 3-10.

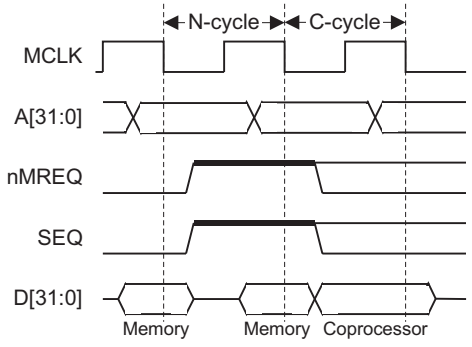

Figure 3-6 Coprocessor register transfer cycles

3.3.6 Summary of ARM memory cycle timing

A summary of ARM7TDMI processor memory cycle timing is shown in Figure 3-7.

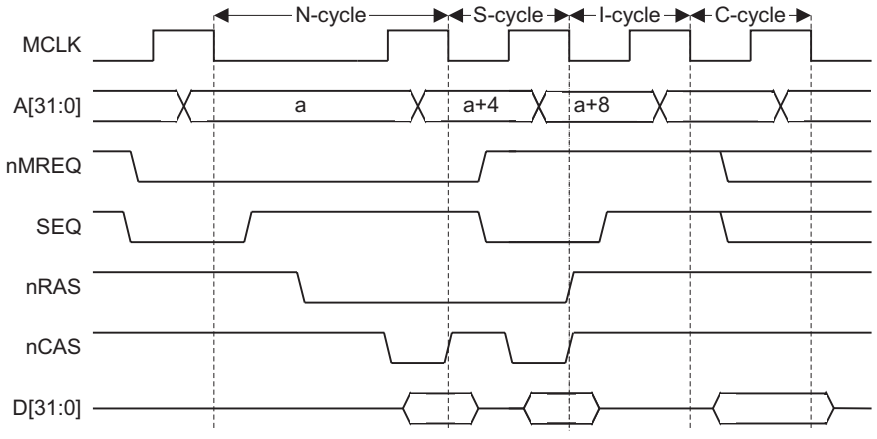

Figure 3-7 Memory cycle timing

3.4 Addressing signals

The address class signals are:

- *A[31:0]* on page 3-11
- *nRW* on page 3-11
- *MAS[1:0]* on page 3-11
- *nOPC* on page 3-12
- *nTRANS* on page 3-13
- *LOCK* on page 3-13
- *TBIT* on page 3-13.

3.4.1 A[31:0]

A[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 signaled the memory system ignores the bottom two bits, **A[1:0]**, and when a halfword access is signaled the memory system ignores the bottom bit, **A[0].**

All data values must be aligned on their natural boundaries. All words must be word-aligned.

3.4.2 nRW

nRW specifies the direction of the transfer. **nRW** indicates an ARM7TDMI processor write cycle when HIGH, and an ARM7TDMI 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 MAS[1:0]

The **MAS[1:0]** bus encodes the size of the transfer. The ARM7TDMI processor can transfer word, halfword, and byte quantities.

All writable memory in an ARM7TDMI processor based system must support the writing of individual bytes or halfwords to allow the use of the C Compiler and the ARM debug tool chain, for example Multi-ICE.

The address produced by the processor is always a byte address. However, the memory system must ignore the bottom redundant bits of the address. The significant address bits are listed in Table 3-3.

Table 3-3 Significant address bits

MAS[1:0]	Width	Significant address bits
00	Byte	**A[31:0]**
01	Halfword	**A[31:1]**
10	Word	**A[31:2]**
11	Reserved	-

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

The ARM7TDMI processor cannot generate bursts of byte transfers.

Note

During instruction accesses the redundant address bits are undefined. The memory system must ignore these redundant bits.

A writable memory system for the ARM7TDMI 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, you might not be able to use either of these tools without data corruption.

3.4.4 nOPC

The **nOPC** output conveys information about the transfer. An MMU can use this signal to determine whether an access is an opcode fetch or a data transfer. This signal can be used with **nTRANS** to implement an access permission scheme. The meaning of **nOPC** is listed in Table 3-4.

Table 3-4 nOPC

nOPC	Opcode/data
0	Opcode
1	Data

3.4.5 nTRANS

The **nTRANS** output conveys information about the transfer. A MMU can use this signal to determine whether an access is from a privileged mode or User mode. This signal can be used with **nOPC** to implement an access permission scheme. The meaning of **nTRANS** is listed in Table 3-5.

Table 3-5 nTRANS encoding

nTRANS	Mode
0	User
1	Privileged

More information relevant to the **nTRANS** signal and security is provided in *Privileged mode access* on page 3-32.

3.4.6 LOCK

LOCK is used to indicate 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/write operation, and can be used to implement semaphores.

3.4.7 TBIT

TBIT is used to indicate the operating state of the ARM7TDMI processor. When in:

- ARM state, the **TBIT** signal is LOW
- Thumb state, the **TBIT** signal is HIGH.

Note

Memory systems do not usually have to use **TBIT** because **MAS[1:0]** indicates the size of the instruction required.

3.5 Address timing

The ARM7TDMI processor address bus can operate in one of two configurations:

- pipelined
- depipelined.

Note

ARM Limited strongly recommends that pipelined address timing is used in new design to obtain optimum system performance.

ARM Limited strongly recommends that **ALE** is tied HIGH and not used in new designs.

Address depipelined configuration is controlled by the **APE** or **ALE** input signal. The configuration is provided to ease the design of the ARM7TDMI processor in both SRAM and DRAM-based systems.

APE affects the timing of the address bus **A[31:0]**, plus **nRW**, **MAS[1:0]**, **LOCK**, **nOPC**, and **nTRANS**.

In most systems, particularly a DRAM-based system, it is desirable to obtain the address from ARM7TDMI processor as early as possible. When **APE** is HIGH then the ARM7TDMI processor address becomes valid after the rising edge of **MCLK** before the memory cycle to which it refers. This timing allows longer periods for address decoding and the generation of DRAM control signals. Figure 3-8 shows the effect on the timing when **APE** is HIGH.

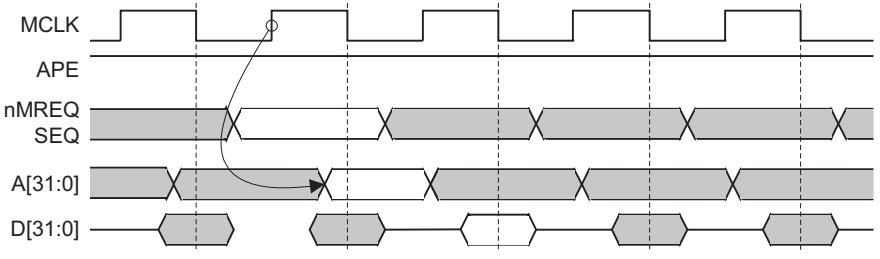

Figure 3-8 Pipelined addresses

SRAMs and ROMs require that the address is held stable throughout the memory cycle. In a system containing SRAM and ROM only, **APE** can be tied permanently LOW, producing the desired address timing. In this configuration the address becomes valid after the falling edge of **MCLK** as shown in Figure 3-9 on page 3-15.

Note

The AMBA specification for *Advanced High-performance Bus* (AHB) and *Advanced System Bus* (ASB) requires a pipelined address bus. This means that **APE** must be configured HIGH.

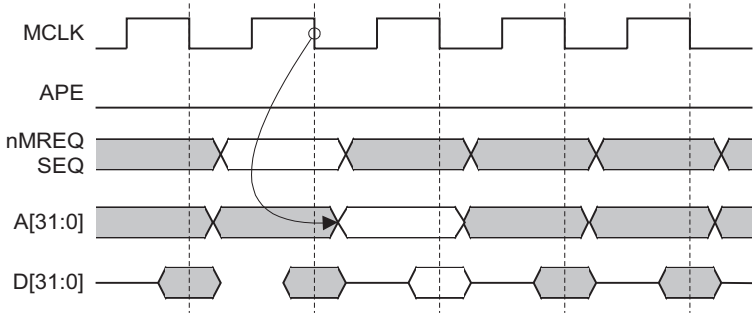

Figure 3-9 Depipelined addresses

Many systems contain a mixture of DRAM, SRAM and ROM. To cater for the different address timing requirements, **APE** can be safely changed during the LOW phase of **MCLK**. Typically, **APE** is held at one level during a burst of sequential accesses to one type of memory. When a nonsequential access occurs, the timing of most systems enforce a wait state to allow for address decoding. As a result of the address decode, **APE** can be driven to the correct value for the particular bank of memory being accessed. The value of **APE** can be held until the memory control signals denote another nonsequential access.

Previous ARM processors included the **ALE** signal, and this is retained for backwards compatibility. This signal also enables you to modify the address timing to achieve the same results as **APE**, but in a dynamic manner. To obtain clean **MCLK** low timing of the address bus by this mechanism, **ALE** must be driven HIGH with the falling edge of **MCLK**, and LOW with the rising edge of **MCLK**. **ALE** can simply be the inverse of **MCLK** but the delay from **MCLK** to **ALE** must be carefully controlled so that the T_{ald} timing constraint is achieved. Figure 3-10 on page 3-16 shows how you can use **ALE** to achieve SRAM compatible address timing. Refer to Chapter 7 *AC and DC Parameters* for details of the exact timing constraints.

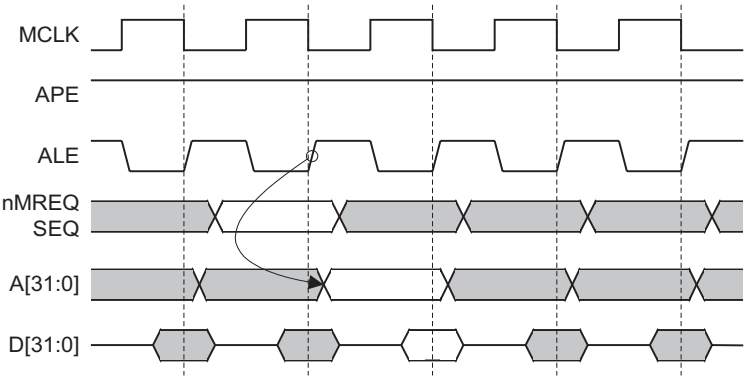

Figure 3-10 SRAM compatible address timing

Note

If **ALE** is to be used to change address timing, then you must tie **APE** HIGH. Similarly, if **APE** is to be used, **ALE** must be tied HIGH.

You can obtain better system performance when the address pipeline is enabled with **APE** HIGH. This allows longer time for address decoding.

3.6 Data timed signals

This section describes:

- *D[31:0], DOUT[31:0], and DIN[31:0]* on page 3-17
- *ABORT* on page 3-24
- *Byte latch enables* on page 3-24
- *Byte and halfword accesses* on page 3-26.

3.6.1 D[31:0], DOUT[31:0], and DIN[31:0]

The ARM7TDMI processor provides both unidirectional data buses, **DIN[31:0]**, **DOUT[31:0]**, and a bidirectional data bus, **D[31:0]**. The configuration input **BUSEN** is used to select which is active. Figure 3-11 shows the arrangement of the data buses and bus-splitter logic.

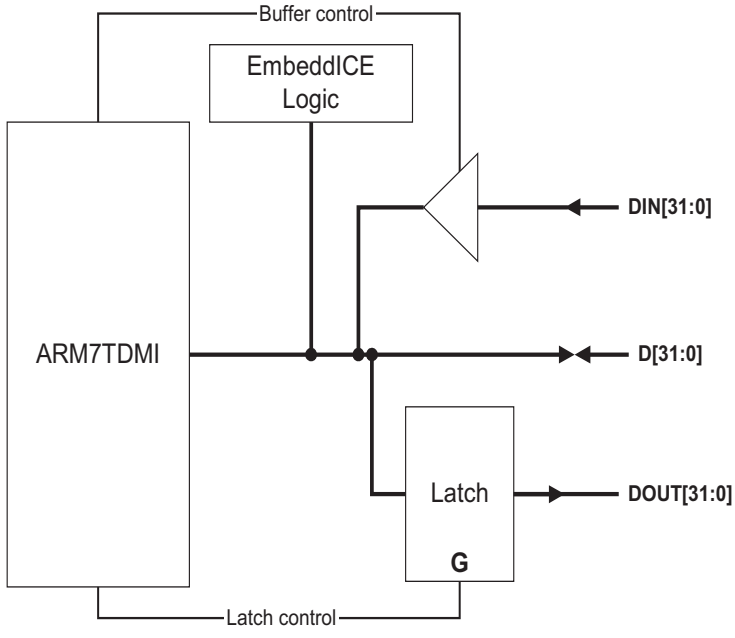

Figure 3-11 External bus arrangement

When the bidirectional data bus is being used then you must disable the unidirectional buses by driving **BUSEN** LOW. The timing of the bus for three cycles, load-store-load, is shown in Figure 3-12 on page 3-18.

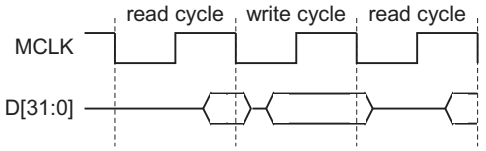

Figure 3-12 Bidirectional bus timing

Unidirectional data bus

When **BUSEN** is HIGH, all instructions and input data are presented on the input data bus, **DIN[31:0]**. The timing of this data is similar to that of the bidirectional bus when in input mode. Data must be set up and held to the falling edge of **MCLK**. For the exact timing requirements refer to Chapter 7 *AC and DC Parameters*.

In this configuration, all output data is presented on **DOUT[31:0]**. The value on this bus only changes when the processor performs a store cycle. Again, the timing of the data is similar to that of the bidirectional data bus. The value on **DOUT[31:0]** changes after the falling edge of **MCLK**.

The bus timing of a read-write-read cycle combination is shown in Figure 3-13.

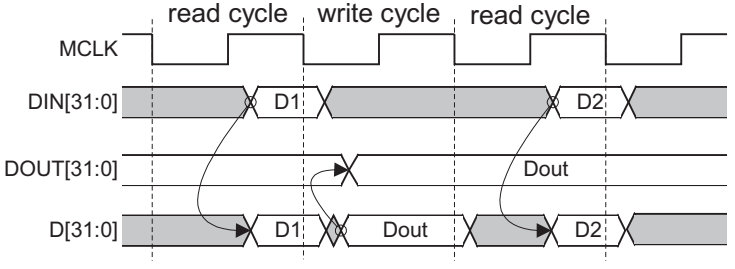

Figure 3-13 Unidirectional bus timing

When the unidirectional data buses are being used, and **BUSEN** is HIGH, the bidirectional bus, **D[31:0]**, must be left unconnected.

The unidirectional buses are typically used internally in ASIC embedded applications. Externally, most systems still require a bidirectional data bus to interface to external memory. Figure 3-14 on page 3-19 shows how you can join the unidirectional buses up at the pads of an ASIC to connect to an external bidirectional bus.

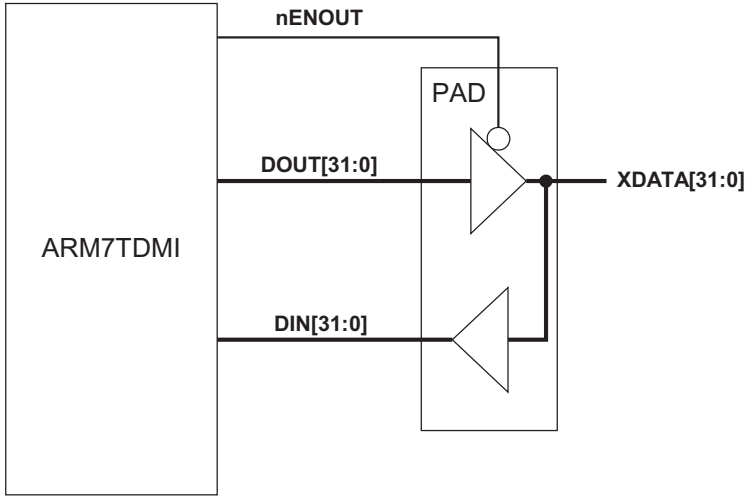

Figure 3-14 External connection of unidirectional buses

Bidirectional data bus

When **BUSEN** is LOW, the buffer between **DIN[31:0]** and **D[31:0]** is disabled. Any data presented on **DIN[31:0]** is ignored. Also, when **BUSEN** is LOW, the value on **DOUT[31:0]** is forced to `0x00000000`.

When the ARM7TDMI processor is reading from memory **DIN[31:0]** is acting as an input. During write cycles the ARM7TDMI core must output data. During phase 2 of the previous cycle, the signal **nRW** is driven HIGH to indicate a write cycle. During the actual cycle, **nENOUT** is driven LOW to indicate that the processor is driving **D[31:0]** as an output. Figure 3-15 on page 3-20 shows the bus timing with the data bus enabled. Figure 3-16 on page 3-20 shows the circuit that exists in the processor for controlling exactly when the external bus is driven out.

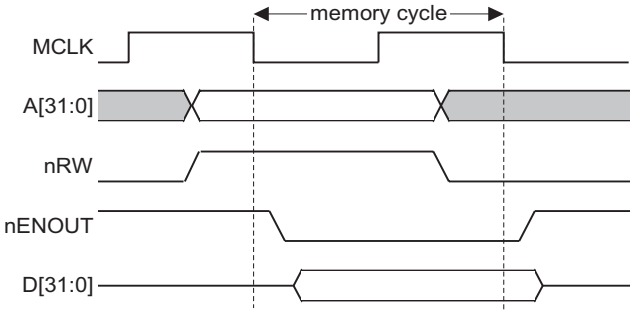

Figure 3-15 Data write bus cycle

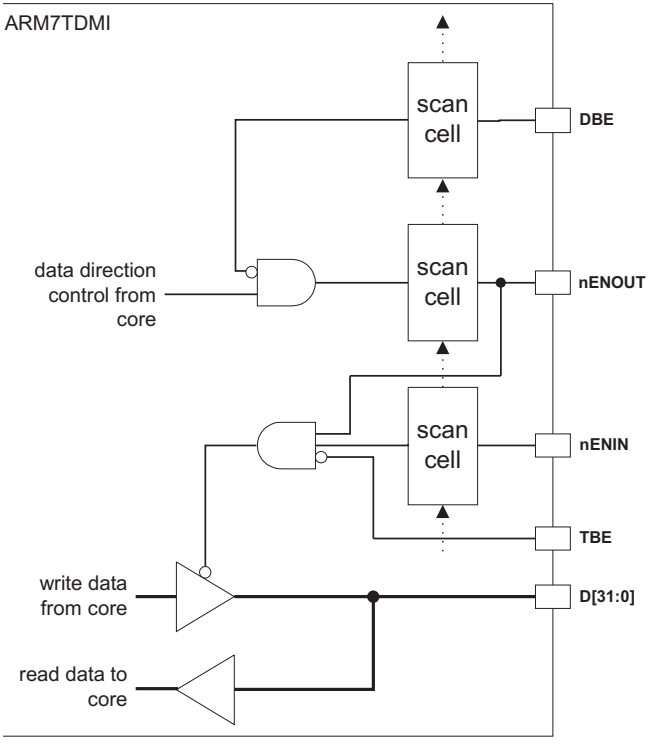

Figure 3-16 Data bus control circuit

The macrocell has an additional bus control signal, **nENIN** that allows the external system to manually tristate the bus. In the simplest systems, **nENIN** can be tied LOW and **nENOUT** can be ignored. In many applications, when the external data bus is a shared resource, greater control might be required. In this situation, **nENIN** can be used to delay when the external bus is driven.

Note

For backwards compatibility, **DBE** is also included. At the macrocell level, **DBE** and **nENIN** have almost identical functionality and in most applications one can be tied to keep the data bus enabled.

The processor has another output control signal called **TBE**. This signal is usually only used during test and must be tied HIGH when not in use. When driven LOW, **TBE** forces all tristateable outputs to high impedance, it is as though both **DBE** and **ABE** have been driven LOW, causing the data bus, the address bus, and all other signals normally controlled by **ABE** to become high impedance.

Note

There is no scan cell on **TBE**. Therefore, **TBE** is completely independent of scan data and can be used to put the outputs into a high impedance state while scan testing takes place.

Table 3-6 lists the tristate control of the processor outputs.

Table 3-6 Tristate control of processor outputs

Processor output	ABE	DBE	TBE
A[31:0]	Yes	-	Yes
D[31:0]	-	Yes	Yes
nRW	Yes	-	Yes
LOCK	Yes	-	Yes
MAS[1:0]	Yes	-	Yes
nOPC	Yes	-	Yes
nTRANS	Yes	-	Yes

ARM7TDMI core test chip example system

Connecting the ARM7TDMI processor data bus, **D[31:0]** to an external shared bus requires additional logic that varies between applications in the case of a test chip.

In this application, care must be taken to prevent bus clash on **D[31:0]** when the data bus drive changes direction. The timing of **nENIN**, and the pad control signals must be arranged so that when the core starts to drive out, the pad drive onto **D[31:0]** is disabled before the core starts to drive. Similarly, when the bus switches back to input, the core must stop driving before the pad is enabled.

The circuit implemented in the ARM7TDMI processor test chip is shown in Figure 3-17 on page 3-23.

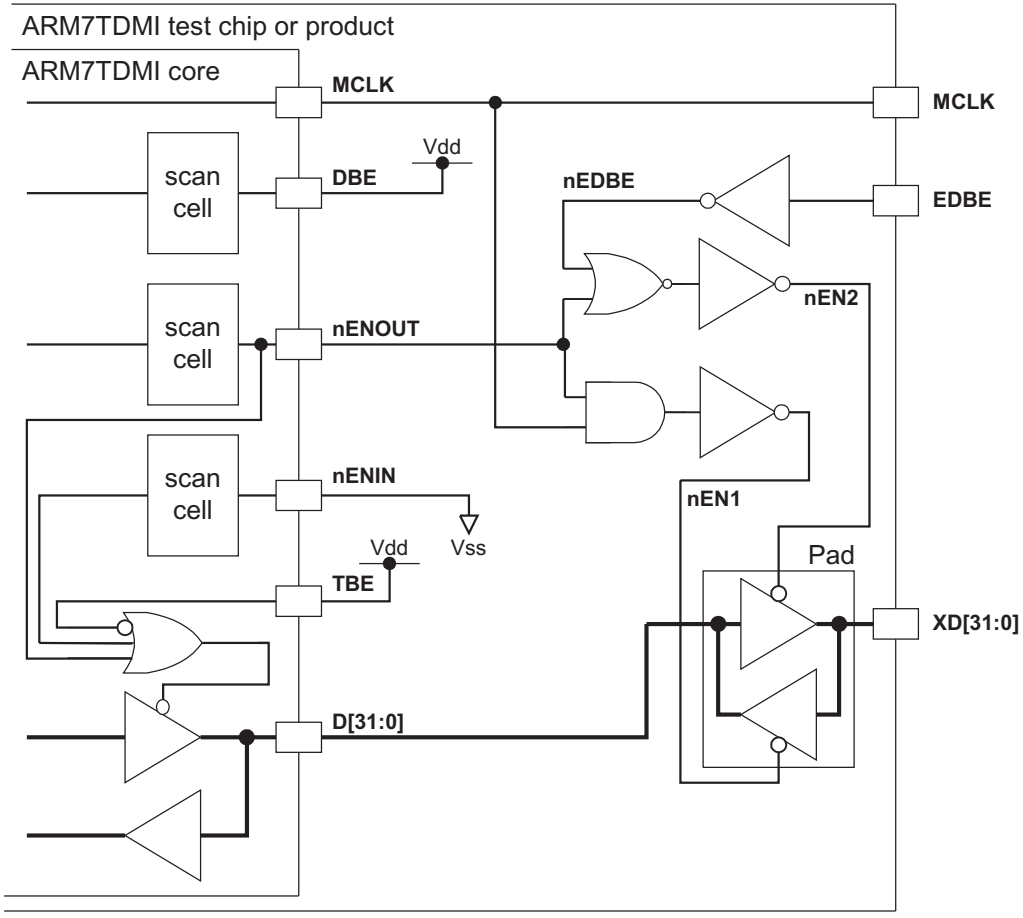

Figure 3-17 Test chip data bus circuit

Note

At the core level, **TBE** and **DBE** are inactive, tied HIGH, because in a packaged part you do not have to manually force the internal buses into a high impedance state. At the pad level, the test chip signal **EDBE** is used by the bus control logic to allow the external memory controller to arbitrate the bus and asynchronously disable the ARM7TDMI core test chip if necessary.

3.6.2 ABORT

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

If **ABORT** is asserted on a data access, it causes the 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.

3.6.3 Byte latch enables

To ease the connection of the ARM7TDMI core to sub-word sized memory systems, input data and instructions can be latched on a byte-by-byte basis. This is achieved by the use of the **BL[3:0]** signal as follows:

- **BL[3]** controls the latching of the data present on **D[31:24]**
- **BL[2]** controls the latching of the data present on **D[23:16]**
- **BL[1]** controls the latching of the data present on **D[15:8]**
- **BL[0]** controls the latching of the data present on **D[7:0]**.

Note

It is recommended that **BL[3:0]** is tied HIGH in new designs and word values from narrow memory systems are latched onto latches that are external to the ARM7TDMI core.

In a memory system that only contains word-wide memory, **BL[3:0]** can be tied HIGH. For sub-word wide memory systems, the **BL[3:0]** signals are used to latch the data as it is read out of memory. For example, a word access to halfword wide memory must take place in two memory cycles:

- in the first cycle, the data for **D[15:0]** is obtained from the memory and latched into the core on the falling edge of **MCLK** when **BL[1:0]** are both HIGH.
- in the second cycle, the data for **D[31:16]** is latched into the core on the falling edge of **MCLK** when **BL[3:2]** are both HIGH and **BL[1:0]** are both LOW.

In Figure 3-18 on page 3-25, a word access is performed from halfword wide memory in two cycles:

- in the first cycle, the read data is applied to the lower half of the bus
- in the second cycle, the read data is applied to the upper half of the bus.

Because two memory cycles are required, **nWAIT** is used to stretch the internal processor clock. **nWAIT** does not affect the operation of the data latches. Using this method, data can be taken from memory as word, halfword, or byte at a time and the memory can have as many wait states as required. In multi-cycle memory accesses, **nWAIT** must be held LOW until the final part is latched.

In the example shown in Figure 3-18, the **BL[3:0]** signals are driven to value `0x3` in the first cycle so that only the latches on **D[15:0]** are open. **BL[3:0]** can be driven to value `0xF` and all of the latches opened. This does not affect the operation of the core because the latches on **D[31:16]** are written with the correct data during the second cycle.

Note

BL[3:0] must be held HIGH during store cycles.

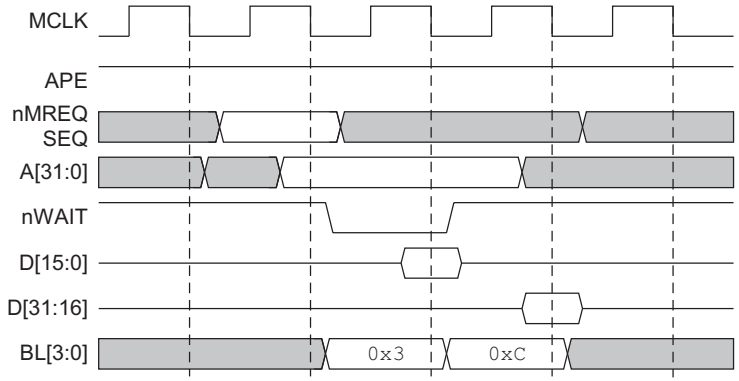

Figure 3-18 Memory access

Figure 3-19 on page 3-26 shows a halfword load from single-wait state byte wide memory. In the figure, each memory access takes two cycles:

- in the first access:
  - **BL[3:0]** are driven to `0xF`
  - the correct data is latched from **D[7:0]**
  - unknown data is latched from **D[31:8]**.
- in the second cycle, the byte for **D[15:8]** is latched so the halfword on **D[15:0]** is correctly read from memory. It does not matter that **D[31:16]** are unknown because the core only extracts the halfword that it is interested in.

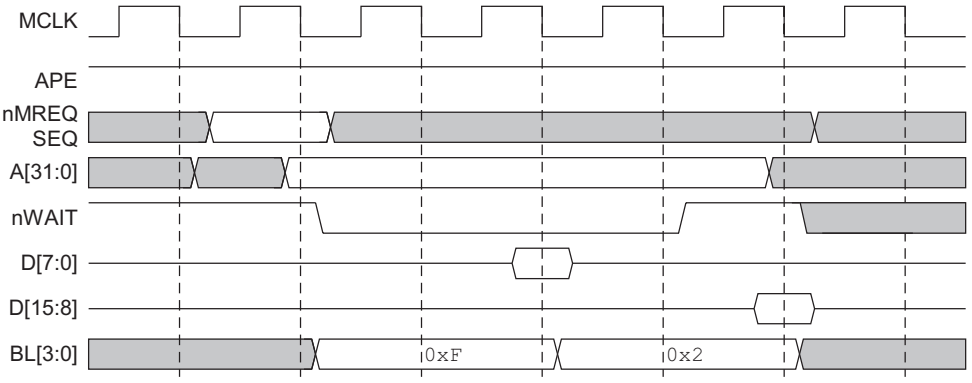

Figure 3-19 Two cycle memory access

3.6.4 Byte and halfword accesses

The processor indicates the size of a transfer by use of the **MAS[1:0]** signal as described in *MAS[1:0]* on page 3-11.

Byte, halfword, and word accesses are described in:

- *Reads* on page 3-26
- *Writes* on page 3-27.

Reads

When a halfword or byte read is performed, a 32-bit memory system can return the complete 32-bit word, and the processor extracts the valid halfword or byte field from it. The fields extracted depend on the state of the **BIGEND** signal, which determines the endian configuration of the system. See *Memory formats* on page 2-4.

A word read from 32-bit memory presents the word value on the whole data bus as listed in Table 3-7.

When connecting 8-bit to 16-bit memory systems to the processor, ensure that the data is presented to the correct byte lanes on the core as listed in Table 3-7 on page 3-27.

Table 3-7 Read accesses

Access type	MAS[1:0]	A[1:0]	Little-endian BIGEND = 0	Big-endian BIGEND = 1
Word	10	XX	**D[31:0]**	**D[31:0]**
Halfword	01	0X	**D[15:0]**	**D[31:16]**
	01	1X	**D[31:16]**	**D[15:0]**
Byte	00	00	**D[7:0]**	**D[31:24]**
	00	01	**D[15:8]**	**D[23:16]**
	00	10	**D[23:16]**	**D[15:8]**
	00	11	**D[31:24]**	**D[7:0]**

Note

For subword reads the value is placed in the ARM register in the least significant bits regardless of the byte lane used to read the data. For example, a byte read on **A[1:0]** = 01 in a little-endian system means that the byte is read on bits **D[15:8]** but is placed in the ARM register bits [7:0].

Writes

When the ARM7TDMI processor performs a byte or halfword write, the data being written is replicated across the data bus, as shown in Figure 3-20 on page 3-28. 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.

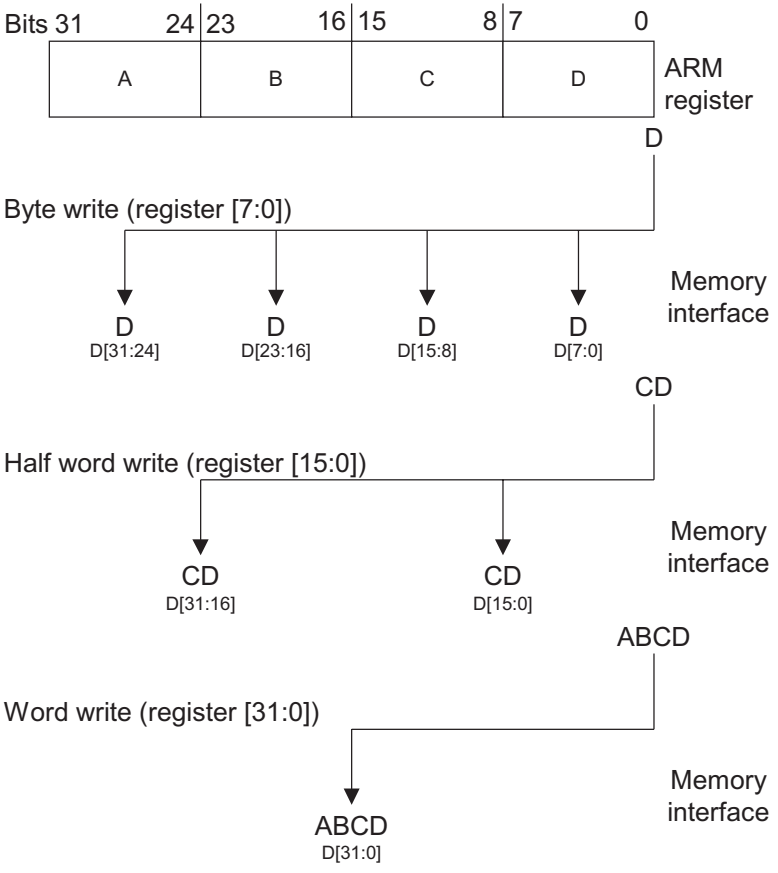

Figure 3-20 Data replication

3.7 Stretching access times

The ARM7TDMI processor does not contain any dynamic logic that relies on regular clocking to maintain the internal state. Therefore, there is no limit upon the maximum period for which **MCLK** can be stretched, or **nWAIT** held LOW. There are two methods available to stretch access times as described in:

- *Modulating MCLK* on page 3-29
- *Use of nWAIT to control bus cycles* on page 3-29.

Note

If you wish to use an *Embedded Trace Macrocell* (ETM) to obtain instruction and data trace information on a trace port then you must use the **nWAIT** signal to stretch access times.

3.7.1 Modulating MCLK

All memory timing is defined by **MCLK**, and long access times can be accommodated by stretching this clock. It is usual to stretch the LOW period of **MCLK**, as this allows the memory manager to abort the operation if the access is eventually unsuccessful.

MCLK can be stretched before being applied to the processor, or the **nWAIT** input can be used together with a free-running **MCLK**. Taking **nWAIT** LOW has the same effect as stretching the LOW period of **MCLK.**

3.7.2 Use of nWAIT to control bus cycles

The pipelined nature of the processor bus interface means that there is a distinction between *clock* cycles and *bus* cycles. **nWAIT** can be used to stretch a *bus* cycle, so that it lasts for many *clock* cycles. The **nWAIT** input allows the timing of bus cycles to be extended in increments of complete **MCLK** cycles:

- when **nWAIT** is HIGH on the falling edge of **MCLK**, a bus cycle completes
- when **nWAIT** is LOW, the bus cycle is extended by stretching the low phase of the internal clock.

nWAIT must only change during the LOW phase of **MCLK.**

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 **nWAIT** this can be more than one **MCLK** cycle. This is illustrated in Figure 3-21 on page 3-30, which shows **nWAIT** being used to extend a nonsequential cycle. In the example, the first N-cycle is followed a few cycles later by another N-cycle to an unrelated address, and the address for the second access is broadcast before the first access completes.

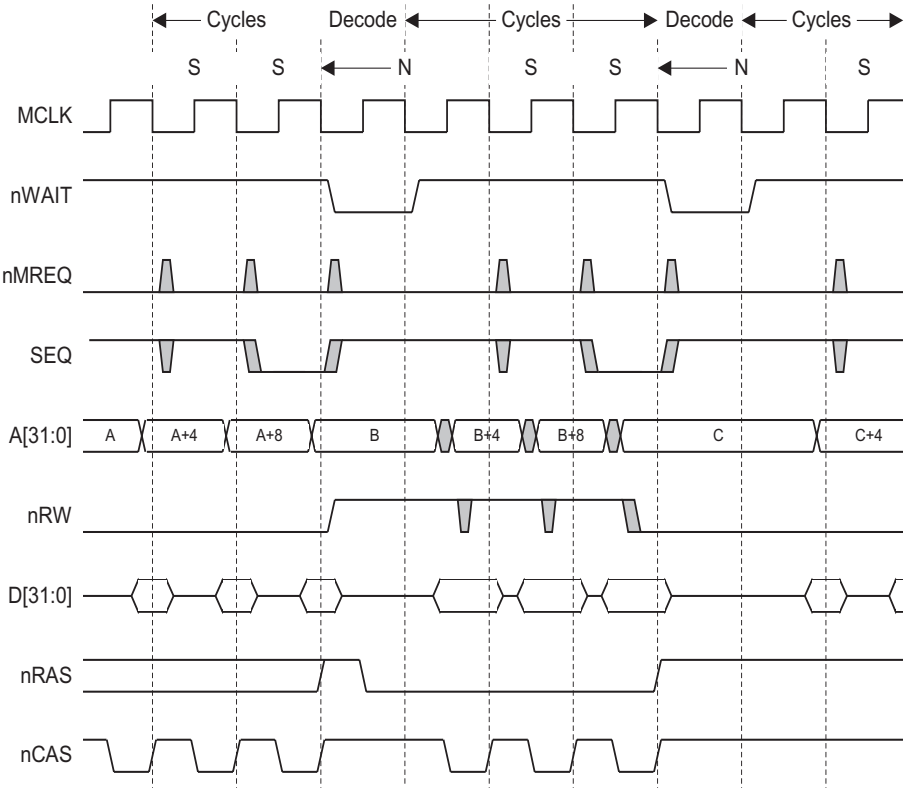

Figure 3-21 Typical system timing

Note

When designing a memory controller, you are strongly advised to sample the values of **nMREQ**, **SEQ**, and the address class signals only when **nWAIT** is HIGH. This ensures that the state of the memory controller is not accidentally updated during an extended bus cycle.

3.8 Action of ARM7TDMI core in debug state

When the ARM7TDMI core is in debug state, **nMREQ** and **SEQ** are forced to indicate internal cycles. This allows the rest of the memory system to ignore the processor and function as normal. Because the rest of the system continues operation, the core ignores aborts and interrupts while in debug state.

The **BIGEND** signal must not be changed by the system during debug. If **BIGEND** changes, not only is there a synchronization problem but the programmer view of the processor changes without the knowledge of the debugger. Signal **nRESET** must also be held stable during debug. If **nRESET** is driven LOW then the state of the processor changes without the knowledge of the debugger.

When instructions are executed in debug state, all bus interface outputs, except **nMREQ** and **SEQ**, change asynchronously to the memory system. For example, every time a new instruction is scanned into the pipeline, the address bus changes. Although this is asynchronous it does not affect the system as **nMREQ** and **SEQ** are forced to indicate internal cycles regardless of what the rest of the processor is doing. The memory controller must be designed to ensure that this asynchronous behavior does not affect the rest of the system.

3.9 Privileged mode access

ARM Limited usually recommends that if only privileged mode access is required from a memory system then you are advised to use the **nTRANS** pin on the core. This signal distinguishes between User and privileged accesses.

The reason that this is recommended is that if the *Operating System* (OS) accesses memory on behalf of the current application then it must perform these accesses in User mode. This is achieved using the LDRT and STRT instructions that set **nTRANS** appropriately.

This measure avoids the possibility of a hacker deliberately passing an invalid pointer to an OS and getting the OS to access this memory with privileged access. This technique could otherwise be used by a hacker to enable the user application to access any memory locations such as I/O space.

The least significant five bits of the CPSR are also output from the core as inverted signals, **nM[4:0]**. These indicate the current processor mode as listed in Table 3-8.

Table 3-8 Use of nM[4:0] to indicate current processor mode

M[4:0]	nM[4:0]	Mode
10000	01111	User
10001	01110	FIQ
10010	01101	IRQ
10011	01100	Supervisor
10111	01000	Abort
11011	00100	Undefined
11111	00000	System

Note

The only time to use the **nM[4:0]** signals is for diagnostic and debug purposes.

3.10 Reset sequence after power up

It is good practice to reset a static device immediately on power-up, to remove any undefined conditions within the device that can otherwise combine to cause a DC path and consequently increase current consumption. Most systems are reset by using a simple RC circuit on the reset pin to remove the undefined states within devices whilst clocking the device.

During reset, the signals **nMREQ** and **SEQ** show internal cycles where the address bus continues to increment by two or four bytes. The initial address and increment values are determined by the state of the core when **nRESET** was asserted. They are undefined after power up.

After **nRESET** has been taken HIGH, the ARM core does two further internal cycles before the first instruction is fetched from the reset vector (address `0x00000000`). It then takes three **MCLK** cycles to advance this instruction through the Fetch-Decode-Execute stages of the ARM instruction pipeline before this first instruction is executed. This is shown in Figure 3-22.

Note

nRESET must be held asserted for a minimum of two **MCLK** cycles to fully reset the core.

You must reset the EmbeddedICE Logic and the TAP controller as well, whether the debug features are used or are not. This is done by taking **nTRST** LOW for at least T_{bsr}, no later than **nRESET**.

In Figure 3-22, x, y, and z are incrementing address values.

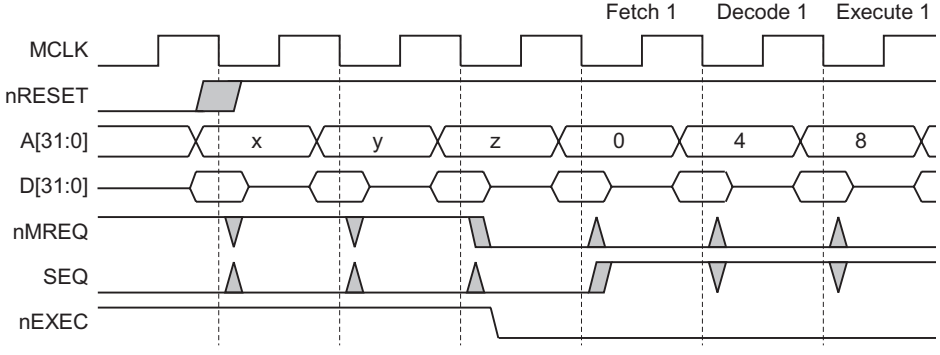

Figure 3-22 Reset sequence

Chapter 4
Coprocessor Interface

This chapter describes the ARM7TDMI core coprocessor interface. It contains the following sections:

4.1 About coprocessors

The ARM7TDMI core instruction set enables you to implement specialized additional instructions using coprocessors to extend functionality. These are separate processing units that are tightly coupled to the ARM7TDMI processor. A typical coprocessor contains:

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

A coprocessor is connected to the same data bus as the ARM7TDMI processor in the system, and tracks the pipeline in the ARM7TDMI processor. This means that the coprocessor can decode the instructions in the instruction stream, and execute those that it supports. Each instruction progresses down both the ARM7TDMI core pipeline and the coprocessor pipeline at the same time.

The execution of instructions is shared between the ARM7TDMI core and the coprocessor.

The ARM7TDMI processor:

1. Evaluates the instruction type and the condition codes to determine whether the instructions are executed by the coprocessor, and communicates this to any coprocessors in the system, using **nCPI**.
2. Generates any addresses that are required by the instruction, including prefetching the next instruction to refill the pipeline.
3. Takes the undefined instruction trap if no coprocessor accepts the instruction.

The coprocessor:

1. Decodes instructions to determine whether it can accept the instruction.
2. Indicates whether it can accept the instruction by using **CPA** and **CPB**.
3. Fetches any values required from its own register bank.
4. Performs the operation required by the instruction.

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

4.1.1 Coprocessor availability

Up to 16 coprocessors can be referenced by a system, each with a unique coprocessor ID number to identify it. The ARM7TDMI core contains one internal coprocessor:

- CP14, the debug communications channel coprocessor.

Other coprocessor numbers have also been reserved. Coprocessor availability is listed in Table 4-1.

Table 4-1 Coprocessor availability

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

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

4.2 Coprocessor interface signals

The signals used to interface the ARM7TDMI core to a coprocessor are grouped into four categories.

The clock and clock control signals are:

- **MCLK**
- **nWAIT**
- **nRESET**.

The pipeline following signals are:

- **nMREQ**
- **SEQ**
- **nTRANS**
- **nOPC**
- **TBIT**.

The handshake signals are:

- **nCPI**
- **CPA**
- **CPB**.

The data signals are:

- **D[31:0]**
- **DIN[31:0]**
- **DOUT[31:0]**.

4.3 Pipeline following signals

Every coprocessor in the system must contain a pipeline follower to track the instructions in the ARM7TDMI processor pipeline. The coprocessors connect to the configured ARM7TDMI core input data bus, **D[31:0]** or **DIN[31:0]**, over which instructions are fetched, and to **MCLK** and **nWAIT**.

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

- At reset, with **nRESET** LOW, the pipeline must either be marked as invalid, or filled with instructions that do not decode to valid instructions for that coprocessor.
- The coprocessor state must only change when **nWAIT** is HIGH, except during reset.
- An instruction must be loaded into the pipeline on the falling edge of **MCLK**, and only when **nOPC**, **nMREQ**, and **TBIT** were all LOW in the previous bus cycle.

  These conditions indicate that this cycle is an ARM instruction fetch, so the new opcode must be read into the pipeline.
- The pipeline must be advanced on the falling edge of **MCLK** when **nOPC**, **nMREQ** and **TBIT** are all LOW in the current bus cycle.

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

Any instructions that are flushed from the ARM7TDMI processor pipeline:

- never signal on **nCPI** that they have entered execute
- are automatically replaced in the coprocessor pipeline follower by the prefetches required to refill the core pipeline.

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

4.4 Coprocessor interface handshaking

Coprocessor interface handshaking is described as follows:

- *The coprocessor* on page 4-6
- *The ARM7TDMI processor* on page 4-7
- *Coprocessor signaling* on page 4-7
- *Consequences of busy-waiting* on page 4-8
- *Coprocessor register transfer instructions* on page 4-9
- *Coprocessor data operations* on page 4-10
- *Coprocessor load and store operations* on page 4-10.

The ARM7TDMI core and any coprocessors in the system perform a handshake using the signals shown in Table 4-2.

Table 4-2 Handshaking signals

Signal	Direction	Meaning
nCPI	ARM7TDMI core to coprocessor	NOT coprocessor instruction
CPA	Coprocessor to ARM7TDMI core	Coprocessor absent
CPB	Coprocessor to ARM7TDMI core	Coprocessor busy

These signals are explained in more detail in *Coprocessor signaling* on page 4-7.

4.4.1 The coprocessor

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

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

1. The coprocessor attempts to execute the instruction.
2. The coprocessor handshakes with the ARM7TDMI core using **CPA** and **CPB**.

Note

The coprocessor can drive **CPA** and **CPB** as soon as it decodes the instruction. It does not have to wait for **nCPI** to be LOW but it must not commit to execute the instruction until **nCPI** has gone LOW.

4.4.2 The ARM7TDMI processor

Coprocessor instructions progress down the ARM7TDMI core pipeline in step with the coprocessor pipeline. A coprocessor instruction is executed if the following are true:

1. The coprocessor instruction has reached the Execute stage of the pipeline. It might not if it is preceded by a branch.
2. The ARM7TDMI processor cannot execute the instruction because the instruction is in the coprocessor or undefined part of the instruction set.
3. The instruction has passed its conditional execution tests.

If all these requirements are met, the ARM7TDMI core signals by taking **nCPI** LOW, this commits the coprocessor to the execution of the coprocessor instruction.

4.4.3 Coprocessor signaling

The coprocessor responses are listed in Table 4-3.

Table 4-3 Summary of coprocessor signaling

CPA	CPB	Response	Remarks
0	0	Coprocessor present	If a coprocessor can accept an instruction, and can start that instruction immediately, it must signal this by driving both **CPA** and **CPB** LOW. The ARM7TDMI processor then ignores the coprocessor instruction and executes the next instruction as normal.
0	1	Coprocessor busy	If a coprocessor can accept an instruction, but is currently unable to process that request, it can stall the ARM7TDMI processor by asserting busy-wait. This is signaled by driving **CPA** LOW, but leaving **CPB** HIGH. When the coprocessor is ready to start executing the instruction it signals this by driving **CPB** LOW. This is shown in Figure 4-1 on page 4-8.
1	0	Invalid response	-
1	1	Coprocessor absent	If a coprocessor cannot accept the instruction currently in Decode it must leave **CPA** and **CPB** both HIGH. The ARM7TDMI processor takes the undefined instruction trap.

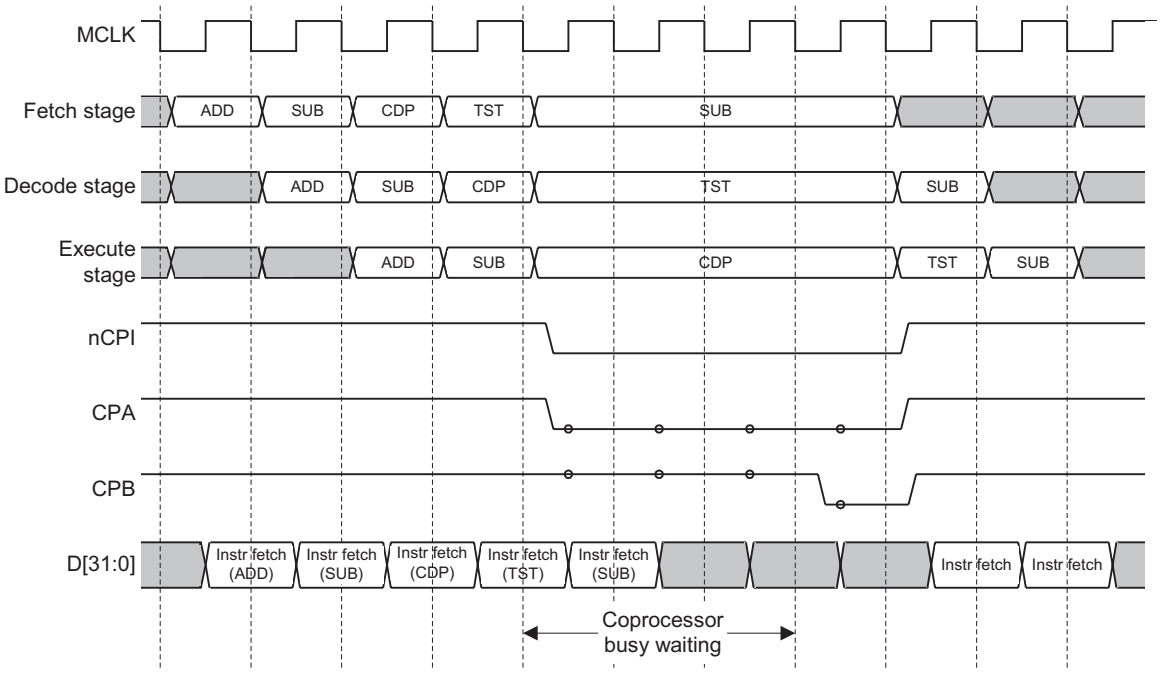

Figure 4-1 Coprocessor busy-wait sequence

CPA and **CPB** are ignored by the ARM7TDMI processor when it does not have a undefined or coprocessor instruction in the Execute stage of the pipeline.

A summary of coprocessor signaling is listed in Table 4-3 on page 4-7.

4.4.4 Consequences of busy-waiting

A busy-waited coprocessor instruction can be interrupted. If a valid **FIQ** or **IRQ** occurs and the appropriate bit is clear in the CSPR, then the ARM7TDMI processor abandons the coprocessor instruction, and signals this by taking **nCPI** HIGH. A coprocessor that is capable of busy-waiting must monitor **nCPI** to detect this condition. When the ARM7TDMI core abandons a coprocessor instruction, the coprocessor also abandons the instruction, and continues tracking the ARM7TDMI processor pipeline.

Caution

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

The ARM7TDMI processor usually returns from processing the interrupt to retry the coprocessor instruction. Other coprocessor instructions can be executed before the interrupted instruction is executed again.

4.4.5 Coprocessor register transfer instructions

The coprocessor register transfer instructions, MCR and MRC, are used to transfer data between a register in the ARM7TDMI processor register bank and a register in the coprocessor register bank. An example sequence for a coprocessor register transfer is shown in Figure 4-2.

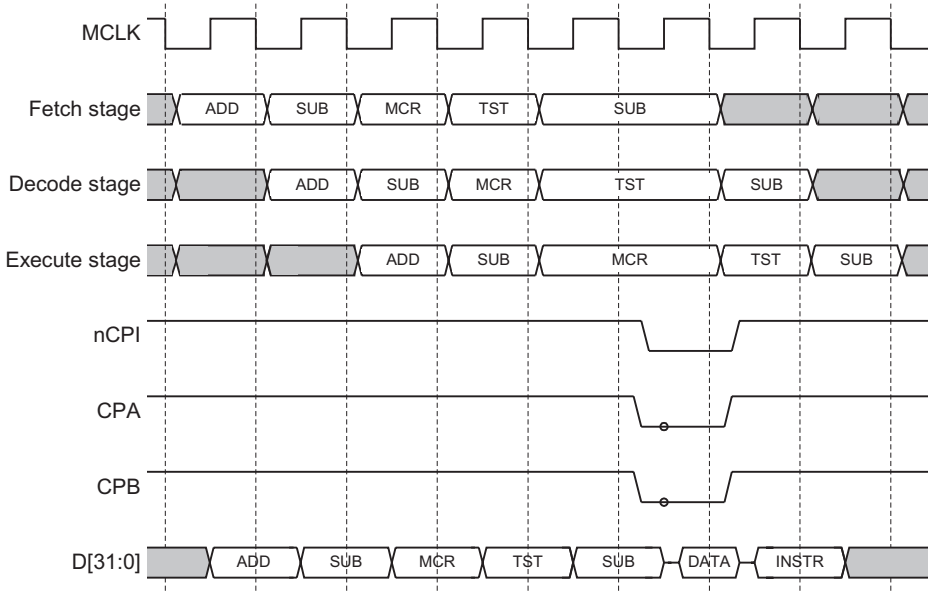

Figure 4-2 Coprocessor register transfer sequence

4.4.6 Coprocessor data operations

Coprocessor data operations, CDP instructions, perform processing operations on the data held in the coprocessor register bank. No information is transferred between the ARM7TDMI processor and the coprocessor as a result of this operation. An example sequence is shown in Figure 4-3.

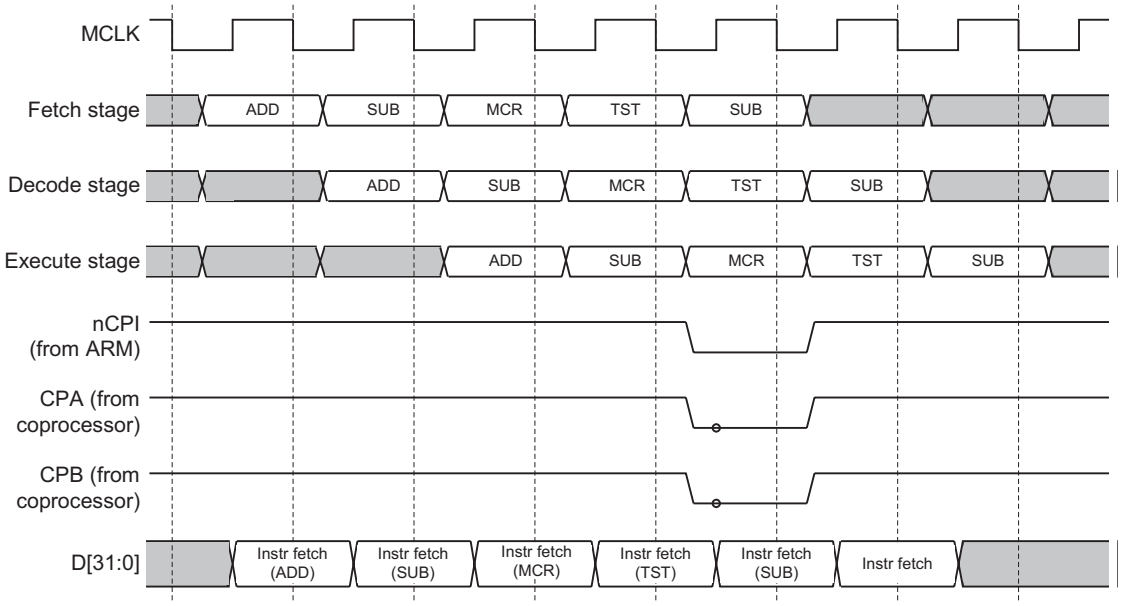

Figure 4-3 Coprocessor data operation sequence

4.4.7 Coprocessor load and store operations

The coprocessor load and store instructions are used to transfer data between a coprocessor and memory. They can be used to transfer either a single word of data, or a number of the coprocessor registers. There is no limit to the number of words of data that can be transferred by a single LDC or STC instruction, but by convention no more than 16 words should be transferred in a single instruction. An example sequence is shown in Figure 4-4 on page 4-11.

Note

If you transfer more than 16 words of data in a single instruction, the worst case interrupt latency of the ARM7TDMI processor increases.

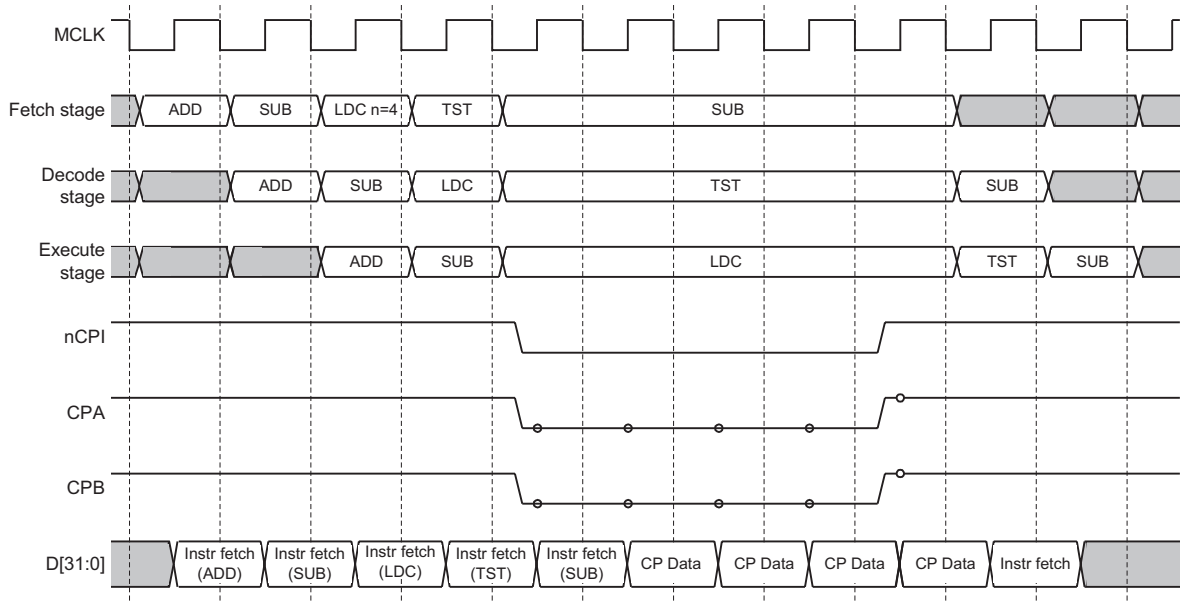

Figure 4-4 Coprocessor load sequence

4.5 Connecting coprocessors

A coprocessor in an ARM7TDMI processor system must have 32-bit connections to:

- the instruction stream from memory
- data written by the core, MCR
- data read by the core, MRC.

The coprocessor can optionally have connections to:

- data written from memory, LDC
- data read to memory, STC.

This section describes:

- *Connecting a single coprocessor* on page 4-12
- *Connecting multiple coprocessors* on page 4-13.

4.5.1 Connecting a single coprocessor

An example of how to connect:

- a coprocessor into an ARM7TDMI processor system if you are using a bidirectional bus is shown in Figure 4-5
- a coprocessor into an ARM7TDMI processor system if you are using a unidirectional bus is shown in Figure 4-6 on page 4-13.

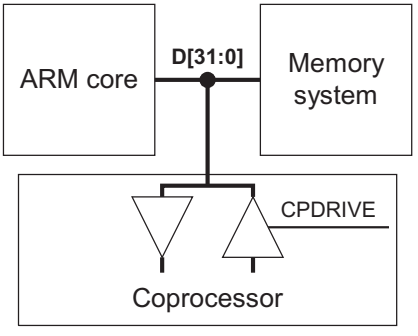

Figure 4-5 Coprocessor connections with bidirectional bus

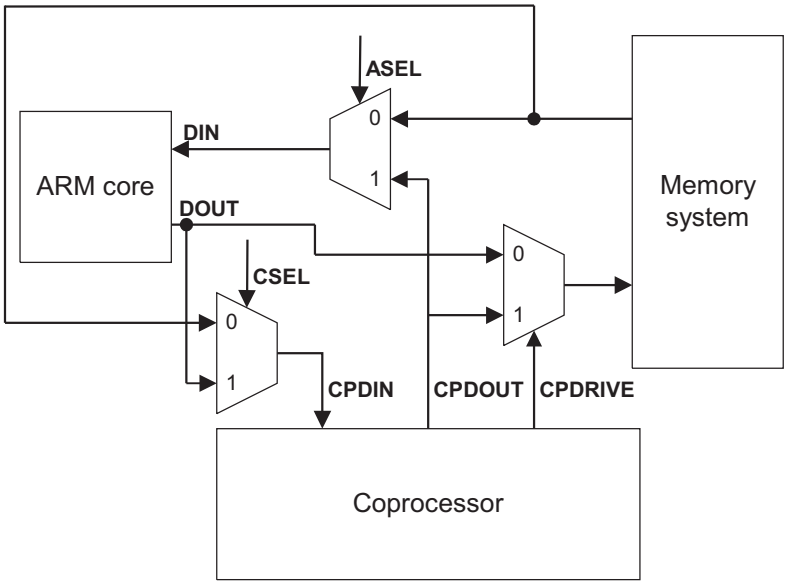

Figure 4-6 Coprocessor connections with unidirectional bus

The logic for Figure 4-6 is as follows:

```
on FALLING MCLK
  ASEL = ((nMREQ = 1 and SEQ = 1) and (not nRW))
  CSEL = ((nMREQ = 1 and SEQ = 1) and (nRW))
```

4.5.2 Connecting multiple coprocessors

If you have multiple coprocessors in your system, connect the handshake signals as follows:

nCPI Connect this signal to all coprocessors present in the system.

CPA and CPB

The individual **CPA** and **CPB** outputs from each coprocessor must be ANDed together, and connected to the **CPA** and **CPB** inputs on the ARM7TDMI processor.

You must multiplex the output data from the coprocessors.

Connecting multiple coprocessors is shown in Figure 4-7 on page 4-14.

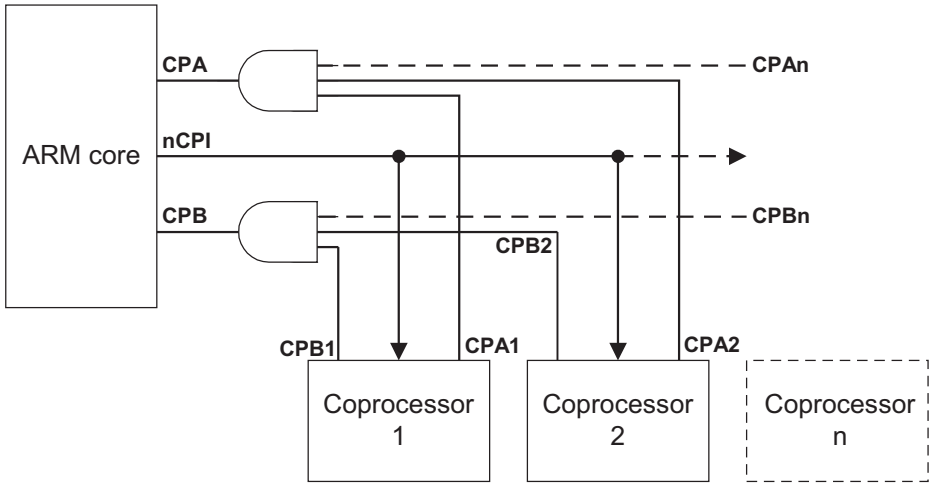

Figure 4-7 Connecting multiple coprocessors

4.6 If you are not using an external coprocessor

If you are implementing a system that does not include any external coprocessors, you must tie both **CPA** and **CPB** HIGH. This indicates that no external coprocessors are present in the system. If any coprocessor instructions are received, they take the undefined instruction trap so that they can be emulated in software if required. The internal coprocessor, CP14, can still be used.

The coprocessor outputs from the ARM7TDMI processor are usually left unconnected but these outputs can be used in other parts of a system as follows:.

- **nCPI**
- **nOPC**
- **TBIT**.

4.7 Undefined instructions

Undefined instructions are treated by the ARM7TDMI processor as coprocessor instructions. All coprocessors must be absent, **CPA** and **CPB** must be HIGH, when an undefined instruction is presented. The ARM7TDMI processor takes the undefined instruction trap.

For undefined instructions to be handled correctly, any coprocessors in a system must give the absent response (**CPA** and **CPB** HIGH) to an undefined instruction. This allows the core to take the undefined instruction exception.

The coprocessor must check bit 27 of the instruction to differentiate between the following instruction types:

- undefined instructions have 0 in bit 27
- coprocessor instructions have 1 in bit 27.

Coprocessor instructions are not supported in the Thumb instruction set but undefined instructions are. All coprocessors must monitor the state of the **TBIT** output from ARM7TDMI core. When the ARM7TDMI core is in Thumb state, coprocessors must drive **CPA** and **CPB** HIGH, and the instructions seen on the data bus must be ignored. In this way, coprocessors do not execute Thumb instructions in error, and all undefined instructions are handled correctly.

4.8 Privileged instructions

The output signal **nTRANS** allows the implementation of coprocessors, or coprocessor instructions, that can only be accessed from privileged modes. The signal meanings are given in Table 4-4.

Table 4-4 Mode identifier signal meanings (nTRANS)

nTRANS	Meaning
0	User mode instruction
1	Privileged mode instruction

If used, the **nTRANS** signal must be sampled at the same time as the coprocessor instruction is fetched and is used in the coprocessor pipeline Decode stage.

Note

If a User mode process, with **nTRANS** LOW, tries to access a coprocessor instruction that can only be executed in a privileged mode, the coprocessor responds with **CPA** and **CPB** HIGH. This causes the ARM7TDMI processor to take the undefined instruction trap.

Chapter 5
Debug Interface

This chapter describes the ARM7TDMI processor debug interface. It contains the following sections:

This chapter also describes the ARM7TDMI processor EmbeddedICE Logic module in the following sections:

5.1 About the debug interface

The ARM7TDMI 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 *Test Access Port* (TAP) controller states. A flow diagram of the TAP controller state transitions is provided in Figure B-2 on page B-5.

The ARM7TDMI processor contains hardware extensions for advanced debugging features. These make it easier to develop application software, operating systems and the hardware itself.

The debug extensions enable you to force the core into *debug state*. In debug state, the core is stopped and isolated from the rest of the system. This allows 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 completed, the debug host restores the core and system state, program execution resumes.

5.1.1 Stages of debug

A request on one of the external debug interface signals, or on an internal functional unit known as the *EmbeddedICE Logic*, forces the ARM7TDMI processor into debug state. The events that activate debug are:

- a breakpoint, an instruction fetch
- a watchpoint, a data access
- an external debug request.

The internal state of the ARM7TDMI processor is then examined using a JTAG-style serial interface. This allows instructions to be inserted serially into the core pipeline without using the external data bus. So, for example, when in debug state, a *Store Multiple* (STM) can be inserted into the instruction pipeline and this exports the contents of the ARM7TDMI core registers. This data can be serially shifted out without affecting the rest of the system.

5.1.2 Clocks

The ARM7TDMI core has two clocks:

- **MCLK** is the memory clock
- **DCLK** is an internal debug clock, generated by the test clock, **TCK**.

During normal operation, the core is clocked by **MCLK** and internal logic holds **DCLK** LOW.

When the ARM7TDMI processor is in the debug state, the core is clocked by **DCLK** under control of the TAP state machine and **MCLK** can free run. The selected clock is output on the signal **ECLK** for use by the external system.

Note

nWAIT has no effect if the CPU core is being debugged and is running from **DCLK**.

5.2 Debug systems

Figure 5-1 shows a typical debug system using an ARM core.

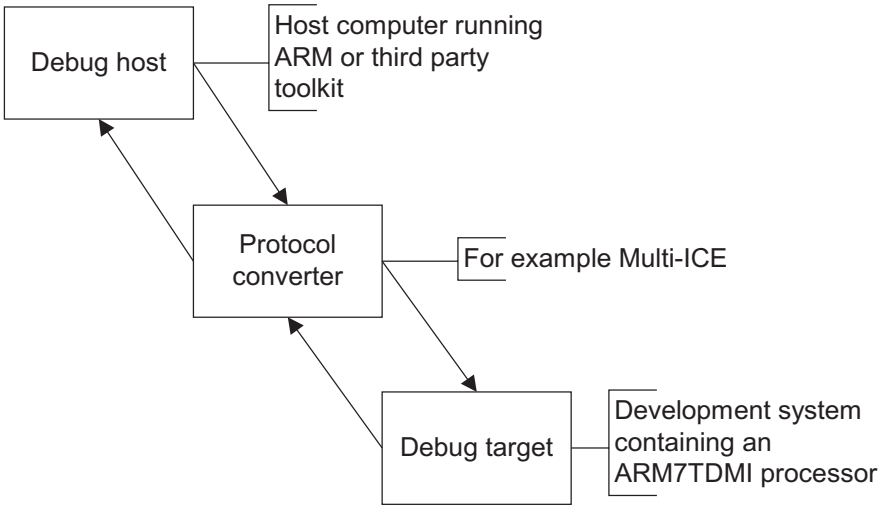

Figure 5-1 Typical debug system

A debug system typically has three parts:

- *Debug host* on page 5-4
- *Protocol converter* on page 5-4
- *Debug target* on page 5-5.

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

5.2.1 Debug host

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

5.2.2 Protocol converter

The protocol converter communicates with the high-level commands issued by the debug host and the low-level commands of the ARM7TDMI processor JTAG interface. Typically it interfaces to the host through an interface such as an enhanced parallel port.

The ARM7TDMI processor has hardware extensions that ease debugging at the lowest level. The debug extensions:

- allow you to halt program execution
- examine and modify the core internal state of the core
- view and modify the state of the memory system
- resume program execution.

5.2.3 Debug target

The major blocks of the debug target are shown in Figure 5-2.

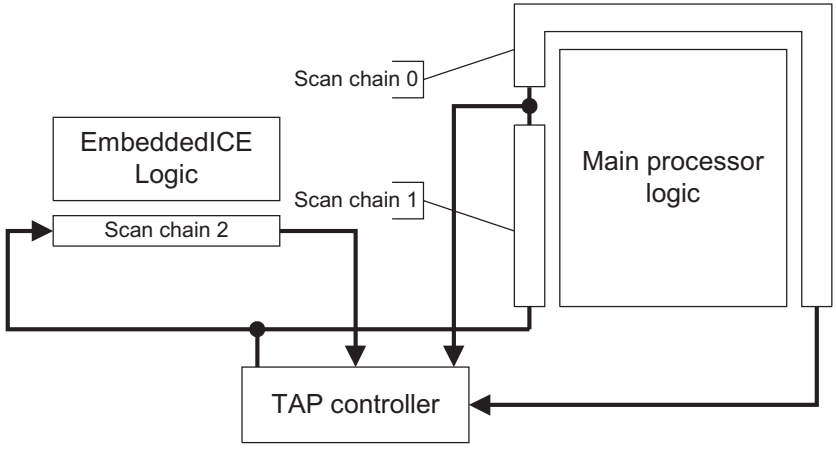

Figure 5-2 ARM7TDMI block diagram

The ARM CPU core

This has hardware support for debug.

The EmbeddedICE Logic

This is a set of registers and comparators used to generate debug exceptions such as breakpoints. This unit is described in *About EmbeddedICE Logic* on page 5-13.

The TAP controller

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

5.3 Debug interface signals

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

- **BREAKPT** and **DBGRQ** are system requests for the processor to enter debug state
- **DBGACK** is used to indicate that the core is in debug state.

Note

DBGEN must be configured HIGH to fully enable the debug features of the processor. Refer to *Disabling EmbeddedICE* on page 5-15.

The following sections describe:

- *Entry into debug state* on page 5-6
- *Action of the processor in debug state* on page 5-9.

5.3.1 Entry into debug state

The ARM7TDMI processor is forced into debug state following a breakpoint, watchpoint, or debug request.

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

- address bus
- data bus
- control signals.

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

When the processor enters debug state, the **DBGACK** signal is asserted. The timing for an externally-generated breakpoint is shown in Figure 5-3 on page 5-7.

The following sections describe:

- *Entry into debug state on breakpoint* on page 5-7
- *Entry into debug state on watchpoint* on page 5-8
- *Entry into debug state on debug request* on page 5-8.

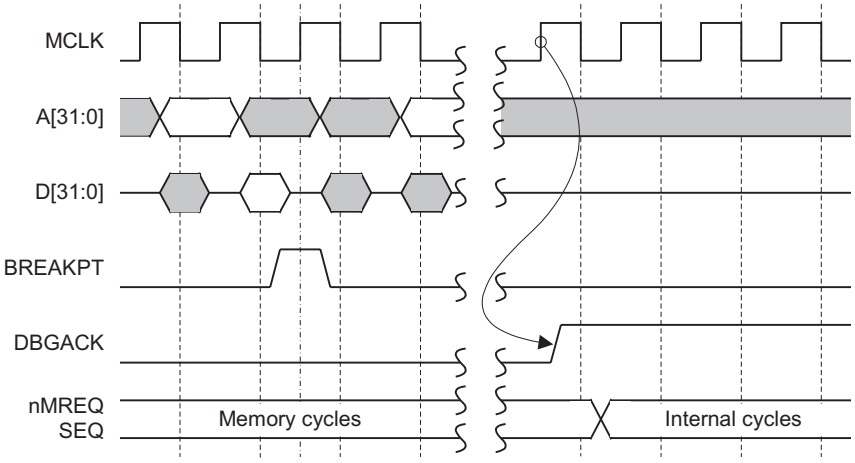

Figure 5-3 Debug state entry

Entry into debug state on breakpoint

The ARM7TDMI core marks instructions as being breakpointed as they enter the instruction pipeline, but the core does not enter debug state until the instruction reaches the Execute stage.

Breakpointed instructions are not executed. Instead, the processor enters debug state. When you examine the internal state, you see the state before the breakpointed instruction. When your examination is complete, remove the breakpoint. This is usually handled automatically by the debugger which also restarts program execution from the previously-breakpointed instruction.

When a breakpointed conditional instruction reaches the Execute stage of the pipeline, the breakpoint is always taken.

Note

The processor enters debug state regardless of whether the condition is met.

A breakpointed instruction does not cause the ARM7TDMI core to enter debug state when:

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

- An exception occurs, causing the processor to flush the instruction pipeline and cancel the breakpoint. In normal circumstances, on exiting from an exception, the ARM7TDMI core branches back to the next instruction to be executed before the exception occurred. In this case, the pipeline is refilled and the breakpoint is reflagged.

Entry into debug state on watchpoint

Watchpoints occur on data accesses. A watchpoint is always taken, but the core might not enter debug state immediately. In all cases, the current instruction completes. If the current instruction is a multi-word load or store, with an LDM or STM, many cycles can elapse before the watchpoint is taken.

When a watchpoint occurs, the current instruction completes, and all changes to the core state are made, load data is written into the destination registers and base write-back occurs.

Note

Watchpoints are similar to Data Aborts. The difference is that when a Data Abort occurs, although the instruction completes, the processor prevents all subsequent changes to the ARM7TDMI processor state. This action allows the abort handler to cure the cause of the abort and the instruction to be re-executed.

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

Entry into debug state on debug request

The ARM7TDMI processor can be forced into debug state on debug request in either of the following ways:

- through EmbeddedICE Logic programming (see *Programming breakpoints* on page B-45 and *Programming watchpoints* on page B-47)
- by asserting the **DBGRQ** pin.

The **DBGRQ** pin is an asynchronous input and is therefore synchronized by logic inside the ARM7TDMI processor before it takes effect. Following synchronization, the core normally enters debug state at the end of the current instruction. However, if the current instruction is a busy-waiting access to a coprocessor, the instruction terminates and ARM7TDMI processor enters debug state immediately. This is similar to the action of **nIRQ** and **nFIQ**.

5.3.2 Action of the processor in debug state

When the ARM7TDMI core enters debug state, the core forces **nMREQ** and **SEQ** to indicate internal cycles. This action allows the rest of the memory system to ignore the core and to function as normal. Because the rest of the system continues to operate, the ARM7TDMI is forced to ignore aborts and interrupts.

The system must not change the **BIGEND** signal during debug because the debugger is unaware that the core has been reconfigured.

nRESET must be held stable during debug because resetting the core while debugging causes the debugger to lose track of the core.

When the system applies reset to the ARM7TDMI processor, with **nRESET** driven LOW, the processor state changes with the debugger unaware that the core has reset.

When instructions are executed in debug state, all memory interface outputs, except **nMREQ** and **SEQ,** change asynchronously to the memory system. For example, every time a new instruction is scanned into the pipeline, the address bus changes. Although this is asynchronous it does not affect the system, as **nMREQ** and **SEQ** are forced to indicate internal cycles regardless of what the rest of the core is doing. The memory controller must be designed to ensure that this asynchronous behavior does not affect the rest of the system.

5.4 ARM7TDMI core clock domains

The ARM7TDMI clocks are described in *Clocks* on page 5-2.

This section describes:

- *Clock switch during debug* on page 5-10
- *Clock switch during test* on page 5-11.

5.4.1 Clock switch during debug

When the ARM7TDMI processor enters debug state, it switches automatically from **MCLK** to **DCLK**, it then asserts **DBGACK** in the HIGH phase of **MCLK**. The switch between the two clocks occurs on the next falling edge of **MCLK**. This is shown in Figure 5-4.

The core is forced to use **DCLK** as the primary clock until debugging is complete. On exit from debug, the core must be allowed to synchronize back to **MCLK**. This must be done by the debugger in the following sequence:

1. The final instruction of the debug sequence is shifted into the data bus scan chain and clocked in by asserting **DCLK**.
2. RESTART is clocked into the TAP instruction register.

The core now automatically resynchronizes back to **MCLK** and starts fetching instructions from memory at **MCLK** speed.

See *Exit from debug state* on page B-26.

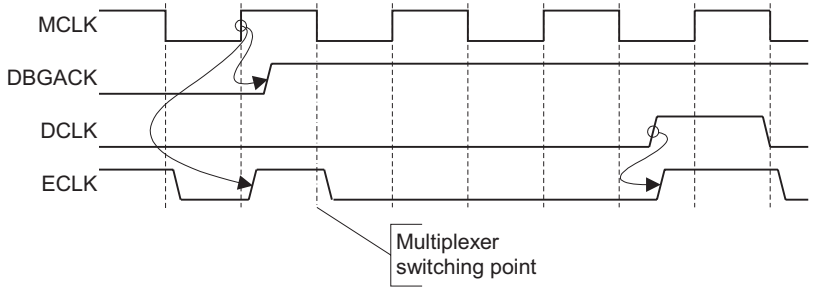

Figure 5-4 Clock switching on entry to debug state

5.4.2 Clock switch during test

When serial test patterns are being applied to the ARM7TDMI core through the JTAG interface, the processor must be clocked using **DCLK**, **MCLK** must be held LOW.

Entry into test is less automatic than debug and you must take care to prevent spurious clocking on the way into test.

The TAP controller can now be used to serially test the processor. If scan chain 0 and INTEST are selected, **DCLK** is generated while the state machine is in the RUN-TEST/IDLE state. During EXTEST, **DCLK** is not generated.

On exit from test, RESTART must be selected as the TAP controller instruction. When this is done, **MCLK** can be resumed. After INTEST testing, you must take care to ensure that the core is in a sensible state before reverting to normal operation. The safest ways to do this is are by using one of the following:

- select RESTART, then cause a system reset
- insert `MOV PC, #0` into the instruction pipeline before reverting.

5.5 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 multiples into the instruction pipeline.

Before you can examine the core and system state, the debugger must determine whether the processor entered debug from Thumb state or ARM state, by examining bit 4 of the EmbeddedICE Logic debug status register. When bit 4 is HIGH, 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-24.

5.6 About EmbeddedICE Logic

The ARM7TDMI processor EmbeddedICE Logic provides integrated on-chip debug support for the ARM7TDMI core.

The EmbeddedICE Logic is programmed serially using the ARM7TDMI processor TAP controller. Figure 5-5 illustrates the relationship between the core, EmbeddedICE Logic, and the TAP controller, showing only the pertinent signals.

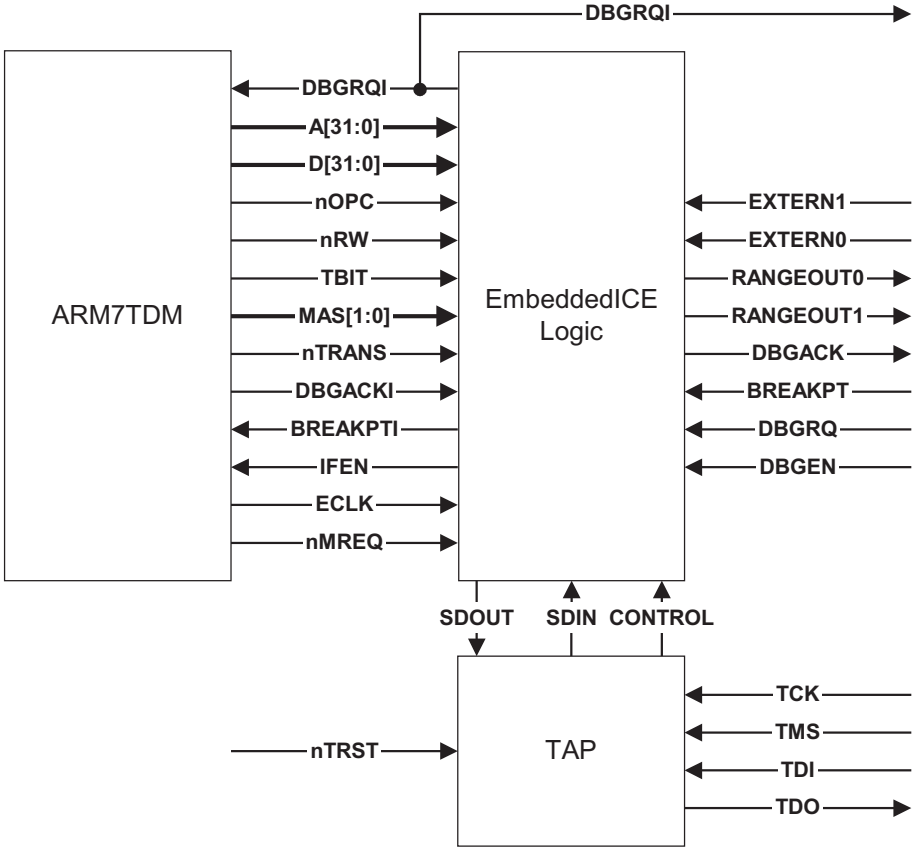

Figure 5-5 ARM7TDM, TAP controller, and EmbeddedICE Logic

The EmbeddedICE Logic comprises:

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

The debug control register and the debug status register provide overall control of EmbeddedICE 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 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 for either a watchpoint or a breakpoint. Watchpoints and breakpoints can be data-dependent.

5.7 Disabling EmbeddedICE

The EmbeddedICE Logic is disabled by setting **DBGEN** LOW.

Caution

Hard-wiring the **DBGEN** input LOW *permanently* disables the EmbeddedICE Logic. However, you must not rely upon this for system security.

When **DBGEN** is LOW:

- **BREAKPT** and **DBGRQ** are forced LOW to the core
- **DBGACK** is forced LOW from the ARM7TDMI core
- interrupts pass through to the processor uninhibited by the debug logic
- EmbeddedICE Logic is put into a low-power mode.

5.8 Debug Communications Channel

The ARM7TDMI processor EmbeddedICE Logic contains a DCC to pass information between the target and the host debugger. This is implemented as coprocessor 14 (CP14).

The DCC comprises:

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

These registers are located in fixed locations in the EmbeddedICE Logic register map, as shown in Figure B-7 on page B-41, and are accessed from the processor using MCR and MRC instructions to coprocessor 14.

The following sections describe:

- *DCC registers* on page 5-16
- *Communications through the DCC* on page 5-17.

5.8.1 DCC registers

The DCC control register is read-only. It controls synchronized handshaking between the processor and the debugger. The control register format is shown in Figure 5-6.

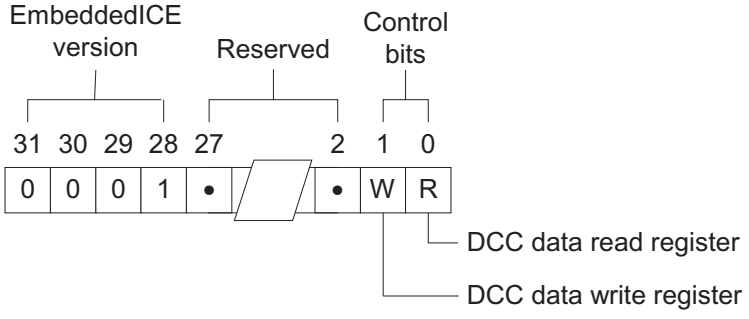

Figure 5-6 DCC control register format

The function of each register bit is as follows:

Bits 31:28 Contain a fixed pattern that denotes the EmbeddedICE version number, in this case 0001.

Bits 27:2 Reserved.

Bit 1 If W is clear, the DCC data write register is ready to accept data from the processor.

If W is set, there is data in the DCC data write register and the debugger can scan it out.

Bit 0 If R is clear, the DCC data read register is free and data can be placed there from the debugger.

If R is set, DCC data read register has data that has not been read by the processor and the debugger must wait.

From the point of view of the debugger, the registers are accessed through scan chain 2 in the usual way. From the point of view of the processor, these registers are accessed through coprocessor register transfer instructions.

Use the instructions listed in Table 5-1 to access the DCC registers.

Table 5-1 DCC register access instructions

Instructions	Explanation
`MRC CP14, 0, Rd, C0, C0, 0`	Places the value from the DCC control register into the destination register (Rd)
`MCR CP14, 0, Rn, C1, C0, 0`	Writes the value in the source register (Rn) to the DCC data write register
`MRC CP14, 0, Rd, C1, C0, 0`	Returns the value in the DCC data read register into Rd

Because the Thumb instruction set does not contain coprocessor instructions, you are advised to access this data through SWI instructions when in Thumb state.

5.8.2 Communications through the DCC

You can send and receive messages through the DCC. The following sections describe:

- *Sending a message to the debugger* on page 5-18
- *Receiving a message from the debugger* on page 5-18
- *Interrupt driven use of the DCC* on page 5-18.

Sending a message to the debugger

When the processor has to 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 debug communications control register is clear.

The processor reads the DCC control register to check status of the bit 1:

If the W bit is clear then 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.

As the data transfer occurs from the processor to the DCC data write register, the W bit is set in the DCC control register. When the debugger polls this register it sees a synchronized version of both the R and W bit. When the debugger sees that the W bit is set, it can read the DCC data write register and scan the data out. The action of reading this data register clears the W bit of the DCC control register. At this point, the communications process can begin again.

Receiving a message from the debugger

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

- if the R bit is clear, 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.

Interrupt driven use of the DCC

An alternative, and potentially more efficient, method to polling the debug communications control register is to use the **COMMTX** and **COMMRX** outputs from the ARM7TDMI processor. You can use these outputs to interrupt the processor when:

- a word is available to be read from the DCC data read register

- the DCC data write register is empty and available for use.

These outputs are usually connected to the system interrupt controller, that drives the **nIRQ** and **nFIQ** ARM7TDMI processor inputs.

Chapter 6
Instruction Cycle Timings

This chapter describes the ARM7TDMI processor instruction cycle operations. It contains the following sections:

6.1 About the instruction cycle timing tables

In the following tables:

- **nMREQ** and **SEQ**, are pipelined up to one cycle ahead of the cycle to which they apply. They are shown in the cycle in which they appear and indicate the next cycle type.
- The address, **MAS[1:0]**, **nRW**, **nOPC**, **nTRANS**, and **TBIT** signals, that appear up to half a cycle ahead, are shown in the cycle to which they apply. The address is incremented to prefetch instructions in most cases. Because the instruction width is four bytes in ARM state and two bytes in Thumb state, the increment varies accordingly.
- The letter L is used to indicate instruction length:
  - four bytes in ARM state
  - two bytes in Thumb state.
- The letter i is used to indicate the width of the instruction fetch output by **MAS[1:0]**:
  - i=2 in ARM state represents word accesses
  - i=1 in Thumb state represents halfword accesses.
- Terms placed inside brackets represent the contents of an address.
- The • symbol indicates zero or more cycles.

6.2 Branch and branch with link

A branch instruction calculates the branch destination in the first cycle, while performing a prefetch from the current PC. This prefetch is done 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.

During the second cycle a fetch is performed from the branch destination, and the return address is stored in register 14 if the link bit is set.

The third cycle performs a fetch from the destination +L, refilling the instruction pipeline. If the instruction is a branch with link (R14 is modified) four is subtracted from R14 to simplify the return instruction from `SUB PC,R14,#4` to `MOV PC,R14`. This allows subroutines to push R14 onto the stack and pop directly into PC upon completion.

The cycle timings are listed in Table 6-1 where:

- pc is the address of the branch instruction
- alu is the destination address calculated by the ARM7TDMI core
- (alu) is the contents of that address.

Table 6-1 Branch instruction cycle operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1	pc+2L	i	0	(pc+2L)	0	0	0
2	alu	i	0	(alu)	0	1	0
3	alu+L	i	0	(alu+L)	0	1	0
	alu+2L						

Note

Branch with link is not available in Thumb state.

6.3 Thumb branch with link

A Thumb Branch with Link operation consists of two consecutive Thumb instructions. Refer to the *ARM Architecture Reference Manual* for more information.

The first instruction acts like a simple data operation to add the PC to the upper part of the offset, storing the result in Register 14, LR.

The second instruction which takes a single cycle acts in a similar fashion to the ARM state branch with link instruction. The first cycle therefore calculates the final branch destination whilst performing a prefetch from the current PC.

The second cycle of the second instruction performs a fetch from the branch destination and the return address is stored in R14.

The third cycle of the second instruction performs a fetch from the destination +2, refilling the instruction pipeline and R14 is modified, with 2 subtracted from it, to simplify the return to `MOV PC, R14`. This makes the `PUSH {..,LR} ; POP {..,PC}` type of subroutine work correctly.

The cycle timings of the complete operation are listed in Table 6-2 where:

- pc is the address of the first instruction of the operation.

Table 6-2 Thumb long branch with link

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1	pc+4	1	0	(pc+4)	0	1	0
2	pc+6	1	0	(pc+6)	0	0	0
3	alu	1	0	(alu)	0	1	0
4	alu+2	1	0	(alu+2)	0	1	0
	alu+4						

6.4 Branch and Exchange

A *Branch and Exchange* (BX) operation takes three cycles and is similar to a branch. In the first cycle, the branch destination and the new core state are extracted from the register source, whilst performing a prefetch from the current PC. This prefetch is performed in all cases, since by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch.

During the second cycle, a fetch is performed from the branch destination address using the new instruction width, dependent on the state that has been selected.

The third cycle performs a fetch from the destination address +2 or +4 (dependent on the new specified state), refilling the instruction pipeline.

The cycle timings are listed in Table 6-3 where:

- W and w represent the instruction width before and after the BX respectively. The width equals four bytes in ARM state and two bytes in Thumb state. For example, when changing from ARM to Thumb state, W equals four and w equals two
- I and i represent the memory access size before and after the BX respectively. **MAS[1:0]** equals two in ARM state and one in Thumb state. When changing from Thumb to ARM state, I equals one and i equals two.
- T and t represent the state of the **TBIT** before and after the BX respectively. **TBIT** equals 0 in ARM state and 1 in Thumb state. When changing from ARM to Thumb state, T equals 0 and t equals 1.

Table 6-3 Branch and exchange instruction cycle operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC	TBIT
1	pc + 2W	I	0	(pc+2W)	0	0	0	T
2	alu	i	0	(alu)	0	1	0	t
3	alu+w	i	0	(alu+w)	0	1	0	t
	alu + 2w							

6.5 Data operations

A data operation executes in a single datapath cycle unless a shift is determined by the contents of a register. A register is read onto the A bus, and a second register or the immediate field onto the B bus (see Figure 1-3 on page 1-8). The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction, and the result, when required, is written to the destination register.

Note

Compare and test operations do not produce results. Only the ALU status flags are affected.

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

When the shift length is specified by a register, an additional datapath cycle occurs during this cycle. The data operation occurs on the next cycle which is an internal cycle that does not access memory. This internal cycle can be merged with the following sequential access by the memory manager as the address remains stable through both cycles.

The PC can be one or more of the register operands. When it is the destination, external bus activity can be affected. If the result is written to the PC, the contents of the instruction pipeline are invalidated, and the address for the next instruction prefetch is taken from the ALU rather than the address incrementer. The instruction pipeline is refilled before any further execution takes place, and during this time exceptions are ignored.

PSR transfer operations (MSR and MRS) exhibit the same timing characteristics as the data operations except that the PC is never used as a source or destination register.

The cycle timings are listed in Table 6-4 on page 6-8 where:

- pc is the address of the branch instruction
- alu is the destination address calculated by the ARM7TDMI core
- (alu) is the contents of that address.

Table 6-4 Data operation instruction cycles

Operation type	Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
normal	1	pc+2L	i	0	(pc+2L)	0	1	0
		pc+3L						
dest=pc	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	i	0	(alu)	0	1	0
	3	alu+L	i	0	(alu+L)	0	1	0
		alu+2L						
shift(Rs)	1	pc+2L	i	0	(pc+2L)	1	0	0
	2	pc+3L	i	0	-	0	1	1
		pc+3L						
shift(Rs)	1	pc+8	2	0	(pc+8)	1	0	0
dest=pc	2	pc+12	2	0	-	0	0	1
	3	alu	2	0	(alu)	0	1	0
	4	alu+4	2	0	(alu+4)	0	1	0
		alu+8						

Note

The shifted register operations where the destination is the PC are not available in Thumb state.

6.6 Multiply and multiply accumulate

The multiply instructions use special hardware that implements integer multiplication with early termination. All cycles except the first are internal

The cycle timings are listed in the following tables:

- *multiply* instruction cycle operations are listed in Table 6-5
- *multiply accumulate* instruction cycle operations are listed in Table 6-6 on page 6-9
- *multiply long* instruction cycle operations are listed in Table 6-7 on page 6-10
- *multiply accumulate long* instruction cycle operations are listed in Table 6-8 on page 6-10.

In Table 6-5 to Table 6-8 on page 6-10:

- m is the number of cycles required by the multiplication algorithm. See *Instruction speed summary* on page 6-29.

Table 6-5 Multiply instruction cycle operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+2L	0	i	(pc+2L)	1	0	0
2	pc+3L	0	i	-	1	0	1
•	pc+3L	0	i	-	1	0	1
m	pc+3L	0	i	-	1	0	1
m+1	pc+3L	0	i	-	0	1	1
	pc+3L						

Table 6-6 Multiply accumulate instruction cycle operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+8	0	2	(pc+8)	1	0	0
2	pc+8	0	2	-	1	0	1
•	pc+12	0	2	-	1	0	1
m	pc+12	0	2	-	1	0	1

Table 6-6 Multiply accumulate instruction cycle operations (continued)

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
m+1	pc+12	0	2	-	1	0	1
m+2	pc+12	0	2	-	0	1	1
	pc+12						

Table 6-7 Multiply long instruction cycle operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+8	0	i	(pc+8)	1	0	0
2	pc+12	0	i	-	1	0	1
•	pc+12	0	i	-	1	0	1
m	pc+12	0	i	-	1	0	1
m+1	pc+12	0	i	-	1	0	1
m+2	pc+12	0	i	-	0	1	1
	pc+12						

Table 6-8 Multiply accumulate long instruction cycle operations

Cycle	Address	nRW	MAS[1:0]	Data	nMREQ	SEQ	nOPC
1	pc+8	0	2	(pc+8)	1	0	0
2	pc+8	0	2	-	1	0	1
•	pc+12	0	2	-	1	0	1
m	pc+12	0	2	-	1	0	1
m+1	pc+12	0	2	-	1	0	1
m+2	pc+12	0	2	-	1	0	1
m+3	pc+12	0	2	-	0	1	1
	pc+12						

Note

The multiply accumulate, multiply long, and multiply accumulate long operations are not available in Thumb state.

6.7 Load register

The first cycle of a load register instruction performs the address calculation. During the second cycle the data is fetched from memory and the base register modification is performed, if required. During the third cycle the data is transferred to the destination register, and external memory is unused. This third cycle can normally be merged with the next prefetch cycle to form one memory N-cycle.

Either the base, or destination, or both, can be the PC, and the prefetch sequence is changed if the PC is affected by the instruction.

The data fetch can abort, and in this case the destination modification is prevented. In addition, if the processor is configured for early abort, the base register write-back is also prevented.

The cycle timings are listed in Table 6-9 where:

- c represents the current processor mode:
  - c=0 for User mode
  - c=1 for all other modes
- d=0 if the T bit has been specified in the instruction (such as LDRT) and d=c at all other times
- s represents the size of the data transfer shown by **MAS[1:0]** (see Table 6-10 on page 6-13).

Table 6-9 Load register instruction cycle operations

Operation type	Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC	nTRANS
normal	1	pc+2L	i	0	(pc+2L)	0	0	0	c
	2	alu	s	0	(alu)	1	0	1	d
	3	pc+3L	i	0	-	0	1	1	c
		pc+3L							
dest=pc	1	pc+8	2	0	(pc+8)	0	0	0	c
	2	alu		0	pc’	1	0	1	d
	3	pc+12	2	0	-	0	0	1	c
	4	pc’	2	0	(pc’)	0	1	0	c
	5	pc’+4	2	0	(pc’+4)	0	1	0	c
		pc’+8							

Note

Operations where the destination is the PC are not available in Thumb state.

Table 6-10 MAS[1:0] signal encoding

bit 1	bit 0	Data size
0	0	byte
0	1	halfword
1	0	word
1	1	reserved

6.8 Store register

The first cycle of a store register instruction is similar to the first cycle of load register instruction. During the second cycle the base modification is performed, and at the same time the data is written to memory. There is no third cycle.

The cycle timings are listed in Table 6-11 where:

- c represents the current processor mode:
  - — c=0 for User mode
  - — c=1 for all other modes
- d=0 if the T bit has been specified in the instruction (such as LDRT) and d=c at all other times.
- s represents the size of the data transfer shown by **MAS[1:0]** (see Table 6-10 on page 6-13).

Table 6-11 Store register instruction cycle operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC	nTRANS
1	pc+2L	i	0	(pc+2L)	0	0	0	c
2	alu	s	1	Rd	0	0	1	d
	pc+3L							

6.9 Load multiple registers

The first cycle of the LDM instruction is used to calculate the address of the first word to be transferred, while performing a prefetch from memory. The second cycle fetches the first word, and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is latched internally in case it is needed to restore processor state after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The cycle timings are listed in Table 6-12.

The last cycle can be merged with the next instruction prefetch to form a single memory N-cycle. If an abort occurs, the instruction continues to completion, but all register modification after the abort is prevented. The final cycle is altered to restore the modified base register (that could have been overwritten by the load activity before the abort occurred).

When the PC is in the list of registers to be loaded the current instruction pipeline must be invalidated.

Note

The PC is always the last register to be loaded, so an abort at any point prevents the PC from being overwritten.

LDM with PC as a destination register is not available in Thumb state. Use `POP{Rlist,PC}` to perform the same function.

Table 6-12 Load multiple registers instruction cycle operations

Destination registers	Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
Single register	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	2	0	(alu)	1	0	1
	3	pc+3L	i	0	-	0	1	1
		pc+3L						

Table 6-12 Load multiple registers instruction cycle operations (continued)

Destination registers	Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
Single register dest=pc	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	2	0	pc’	1	0	1
	3	pc+3L	i	0	-	0	0	1
	4	pc’	i	0	(pc’)	0	1	0
	5	pc’+L	i	0	(pc’+L)	0	1	0
		pc’+2L						
n registers (n>1)	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	2	0	(alu)	0	1	1
	•	alu+•	2	0	(alu+•)	0	1	1
	n	alu+•	2	0	(alu+•)	0	1	1
	n+1	alu+•	2	0	(alu+•)	1	0	1
	n+2	pc+3L	i	0	-	0	1	1
		pc+3L						
n registers (n>1) including pc	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	2	0	(alu)	0	1	1
	•	alu+•	2	0	(alu+•)	0	1	1
	n	alu+•	2	0	(alu+•)	0	1	1
	n+1	alu+•	2	0	pc’	1	0	1
	n+2	pc+3L	i	0	-	0	0	1
	n+3	pc’	i	0	(pc’)	0	1	0
	n+4	pc’+L	i	0	(pc’+L)	0	1	0
		pc’+2L						

6.10 Store multiple registers

The store multiple instruction proceeds very much as load multiple instruction, without the final cycle. The abort handling is much more straightforward as there is no wholesale overwriting of registers.

The cycle timings are listed in Table 6-13 where:

- Ra is the first register specified
- R• are the subsequent registers specified.

Table 6-13 Store multiple registers instruction cycle operations

Register	Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
Single register	1	pc+2L	i	0	(pc+2L)	0	0	0
	2	alu	2	1	Ra	0	0	1
		pc+3L						
n registers (n>1)	1	pc+8	i	0	(pc+2L)	0	0	0
	2	alu	2	1	Ra	0	1	1
	•	alu+•	2	1	R•	0	1	1
	n	alu+•	2	1	R•	0	1	1
	n+1	alu+•	2	1	R•	0	0	1
		pc+12						

6.11 Data swap

This is similar to the load and store register instructions, but the actual swap takes place in the second and third cycles. In the second cycle, the data is fetched from external memory. In the third cycle, the contents of the source register are written out to the external memory. The data read in the second cycle is written into the destination register during the fourth cycle.

LOCK is driven HIGH during the second and third cycles to indicate that both cycles must be allowed to complete without interruption.

The data swapped can be a byte or word quantity. Halfword quantities cannot be specified.

The swap operation can be aborted in either the read or write cycle, and in both cases the destination register is not affected.

The cycle timings are listed in Table 6-14 where:

- s represents the size of the data transfer shown by **MAS[1:0]** (see Table 6-10 on page 6-13), s can only represent byte and word transfers. Halfword transfers are not available.

Table 6-14 Data swap instruction cycle operations

Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	LOCK
1	pc+8	2	0	(pc+8)	0	0	0	0
2	Rn	b/w	0	(Rn)	0	0	1	1
3	Rn	b/w	1	Rm	1	0	1	1
4	pc+12	2	0	-	0	1	1	0
	pc+12							

Note

The data swap operation is not available in Thumb state.

6.12 Software interrupt and exception entry

Exceptions (including software interrupts) force the PC to a particular value and cause the instruction pipeline to be refilled. During the first cycle the forced address is constructed, and a mode change can take place. The return address is moved to R14 and the CPSR to SPSR_svc.

During the second cycle the return address is modified to facilitate return, though this modification is less useful than in the case of the branch with link instruction.

The third cycle is required only to complete the refilling of the instruction pipeline.

The cycle timings are listed in Table 6-15 where:

- pc for:
  - — software interrupts is the address of the SWI instruction
  - — Prefetch Aborts is the address of the aborting instruction
  - — Data Aborts is the address of the instruction following the one which attempted the aborted data transfer
  - — other exceptions is the address of the instruction following the last one to be executed before entering the exception
- C represents the current mode-dependent value
- T represents the current state-dependent value
- Xn is the appropriate trap address.

Table 6-15 Software Interrupt instruction cycle operations

Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nTRANS	Mode	TBIT
1	pc+2L	i	0	(pc+2L)	0	0	0	C	old	T
2	Xn	2	0	(Xn)	0	1	0	1	exception	0
3	Xn+4	2	0	(Xn+4)	0	1	0	1	exception	0
	Xn+8									

6.13 Coprocessor data operation

A coprocessor data operation is a request from the core for the coprocessor to initiate some action. The action does not have to be completed for some time, but the coprocessor must commit to doing it before driving **CPB** LOW.

If the coprocessor is not capable of performing the requested task, it must leave **CPA** and **CPB** HIGH. If it can do the task, but cannot commit right now, it must drive **CPA** LOW but leave **CPB** HIGH until it can commit. The core busy-waits until **CPB** goes LOW.

The cycle timings are listed in Table 6-16 where:

- b represents the busy cycles.

Table 6-16 Coprocessor data operation instruction cycle operations

CP status	Cycle	Address	nRW	MAS [1:0]	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	0	2	(pc+8)	0	0	0	0	0	0
		pc+12									
not ready	1	pc+8	0	2	(pc+8)	1	0	0	0	0	1
	2	pc+8	0	2	-	1	0	1	0	0	1
	•	pc+8	0	2	-	1	0	1	0	0	1
	b	pc+8	0	2	-	0	0	1	0	0	0
		pc+12									

Note

Coprocessor data operations are not available in Thumb state.

6.14 Coprocessor data transfer from memory to coprocessor

For coprocessor transfer instructions from memory the coprocessor must commit to the transfer only when it is ready to accept the data. When **CPB** goes LOW, the processor produces the addresses and expects the coprocessor to take the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred, and indicates the last transfer cycle by driving **CPA** and **CPB** HIGH.

The ARM7TDMI processor spends the first cycle (and any busy-wait cycles) generating the transfer address, and updates the base address during the transfer cycles.

The cycle timings are listed in Table 6-17 where:

- b represents the busy cycles
- n represents the number of registers.

Table 6-17 Coprocessor data transfer instruction cycle operations

CP register status	Cycles	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
Single	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
register	2	alu	2	0	(alu)	0	0	1	1	1	1
ready		pc+12									
Single	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
register	2	pc+8	2	0	-	1	0	1	0	0	1
not ready	•	pc+8	2	0	-	1	0	1	0	0	1
	b	pc+8	2	0	-	0	0	1	0	0	0
	b+1	alu	2	0	(alu)	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	2	0	(alu)	0	1	1	1	0	0
ready	•	alu+•	2	0	(alu+•)	0	1	1	1	0	0
	n	alu+•	2	0	(alu+•)	0	1	1	1	0	0
	n+1	alu+•	2	0	(alu+•)	0	0	1	1	1	1
		pc+12									

Table 6-17 Coprocessor data transfer instruction cycle operations (continued)

CP register status	Cycles	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
n registers	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
(n>1)	2	pc+8	2	0	-	1	0	1	0	0	1
not ready	•	pc+8	2	0	-	1	0	1	0	0	1
	b	pc+8	2	0	-	0	0	1	0	0	0
	b+1	alu	2	0	(alu)	0	1	1	1	0	0
	•	alu+•		0	(alu+•)	0	1	1	1	0	0
	n+b	alu+•	2	0	(alu+•)	0	1	1	1	0	0
	n+b+1	alu+•	2	0	(alu+•)	0	0	1	1	1	1
		pc+12									

Note

Coprocessor data transfer operations are not available in Thumb state.

6.15 Coprocessor data transfer from coprocessor to memory

The ARM7TDMI processor controls these instructions in the same way as for memory to coprocessor transfers, with the exception that the **nRW** line is inverted during the transfer cycle.

The cycle timings are listed in Table 6-18 where:

- b represents the busy cycles
- n represents the number of registers.

Table 6-18 coprocessor data transfer instruction cycle operations

CP register status	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
Single	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
register	2	alu	2	1	CPdata	0	0	1	1	1	1
ready	-	pc+12	-	-	-	-	-	-	-	-	-
Single	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
register	2	pc+8	2	0	-	1	0	1	0	0	1
not ready	•	pc+8	2	0	-	1	0	1	0	0	1
	b	pc+8	2	0	-	0	0	1	0	0	0
	b+1	alu	2	1	CPdata	0	0	1	1	1	1
		pc+12									
n registers	1	pc+8	2	0	(pc+8)	0	0	0	0	0	0
(n>1)	2	alu	2	1	CPdata	0	1	1	1	0	0
ready	•	alu+•	2	1	CPdata	0	1	1	1	0	0
	n	alu+•	2	1	CPdata	0	1	1	1	0	0
	n+1	alu+•	2	1	CPdata	0	0	1	1	1	1
		pc+12									

Table 6-18 coprocessor data transfer instruction cycle operations (continued)

CP register status	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
n registers	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
(n>1)	2	pc+8	2	0	-	1	0	1	0	0	1
not ready	•	pc+8	2	0	-	1	0	1	0	0	1
	b	pc+8	2	0	-	0	0	1	0	0	0
	b+1	alu	2	1	CPdata	0	1	1	1	0	0
	•	alu+•	2	1	CPdata	0	1	1	1	0	0
	n+b	alu+•	2	1	CPdata	0	1	1	1	0	0
	n+b+1	alu+•	2	1	CPdata	0	0	1	1	1	1
		pc+12									

Note

Coprocessor data transfer operations are not available in Thumb state.

6.16 Coprocessor register transfer, load from coprocessor

The busy-wait cycles are similar to those described in *Coprocessor data transfer from memory to coprocessor* on page 6-21, but the transfer is limited to one word, and the ARM7TDMI core puts the data into the destination register in the third cycle. The third cycle can be merged with the next prefetch cycle into one memory N-cycle as with all processor register load instructions.

The cycle timings are listed in Table 6-19 where:

- b represents the busy cycles.

Table 6-19 Coprocessor register transfer, load from coprocessor

	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	2	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	2	0	CPdata	1	0	1	1	1	1
	3	pc+12	2	0	-	0	1	1	1	-	-
	-	pc+12									
not ready	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
	2	pc+8	2	0	-	1	0	1	0	0	1
	•	pc+8	2	0	-	1	0	1	0	0	1
	b	pc+8	2	0	-	1	1	1	0	0	0
	b+1	pc+12	2	0	CPdata	1	0	1	1	1	1
	b+2	pc+12	2	0	-	0	1	1	1	-	-
		pc+12									

Note

Coprocessor register transfer operations are not available in Thumb state.

6.17 Coprocessor register transfer, store to coprocessor

This is the same as described in *Coprocessor register transfer, load from coprocessor* on page 6-25, except that the last cycle is omitted.

The cycle timings are listed in Table 6-20 where:

- b represents the busy cycles.

Table 6-20 Coprocessor register transfer, store to coprocessor

	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
ready	1	pc+8	2	0	(pc+8)	1	1	0	0	0	0
	2	pc+12	2	1	Rd	0	0	1	1	1	1
		pc+12									
not ready	1	pc+8	2	0	(pc+8)	1	0	0	0	0	1
	2	pc+8	2	0	-	1	0	1	0	0	1
	•	pc+8	2	0	-	1	0	1	0	0	1
	b	pc+8	2	0	-	1	1	1	0	0	0
	b+1	pc+12	2	1	Rd	0	0	1	1	1	1
		pc+12									

Note

Coprocessor register transfer operations are not available in Thumb state.

6.18 Undefined instructions and coprocessor absent

When the processor attempts to execute an instruction that neither it nor a coprocessor can perform (including all undefined instructions) this causes the processor to take the undefined instruction trap.

Cycle timings are listed in Table 6-21 where:

- C represents the current mode-dependent value
- T represents the current state-dependent value.

Table 6-21 Undefined instruction cycle operations

Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	nTRANS	Mode	TBIT
1	pc+2L	i	0	(pc+2L)	1	0	0	0	C	Old	T
2	pc+2L	i	0	-	0	0	0	1	C	Old	T
3	Xn	2	0	(Xn)	0	1	0	1	1	00100	0
4	Xn+4	2	0	(Xn+4)	0	1	0	1	1	00100	0
	Xn+8										

Note

- Coprocessor instructions are not available in Thumb state.
- **CPA** and **CPB** are HIGH during the undefined instruction trap.

6.19 Unexecuted instructions

Any instruction whose condition code is not met does not execute and adds one cycle to the execution time of the code segment in which it is embedded (see Table 6-22).

Table 6-22 Unexecuted instruction cycle operations

Cycle	Address	MAS[1:0]	nRW	Data	nMREQ	SEQ	nOPC
1	pc+2L	i	0	(pc+2L)	0	1	0
	pc+3L						

6.20 Instruction speed summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle, one instruction can be using the data path while the next is being decoded and the one after that is being fetched. For this reason Table 6-23 presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time, in cycles, for a routine can be calculated from these figures listed in Table 6-23. These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

If the condition is not met then all instructions take one S-cycle. The cycle types N, S, I, and C are described in *Bus cycle types* on page 3-4.

In Table 6-23:

- b is the number of cycles spent in the coprocessor busy-wait loop
- m is:
  - 1 if bits [32:8] of the multiplier operand are all zero or one
  - 2 if bits [32:16] of the multiplier operand are all zero or one
  - 3 if bits [31:24] of the multiplier operand are all zero or all one
- n is the number of words transferred.

Table 6-23 ARM instruction speed summary

Instruction	Cycle count	Additional
Data Processing	S	+I for SHIFT(Rs) +S + N if R15 written
MSR, MRS	S	-
LDR	S+N+I	+S +N if R15 loaded
STR	2N	-
LDM	nS+N+I	+S +N if R15 loaded
STM	(n-1)S+2N	-
SWP	S+2N+I	-
B,BL	2S+N	-
SWI, trap	2S+N	-
MUL	S+mI	-
MLA	S+(m+1)I	-

Table 6-23 ARM instruction speed summary (continued)

Instruction	Cycle count	Additional
MULL	S+(m+1)I	-
MLAL	S+(m+2)I	-
CDP	S+bI	-
LDC, STC	(n-1)S+2N+bI	-
MCR	N+bI+C	-
MRC	S+(b+1)I+C	-

Chapter 7
AC and DC Parameters

This chapter gives the AC timing parameters of the ARM7TDMI core. It contains the following sections:

7.1 Timing diagram information

Each timing diagram in this chapter is provided with a table that shows the timing parameters. In the tables:

- the letter f at the end of a signal name indicates the falling edge
- the letter r at the end of a signal name indicates the rising edge.

7.2 General timing

Figure 7-1 shows the ARM7TDMI general timing. The timing parameters used in Figure 7-1 are listed in Table 7-1 on page 7-5.

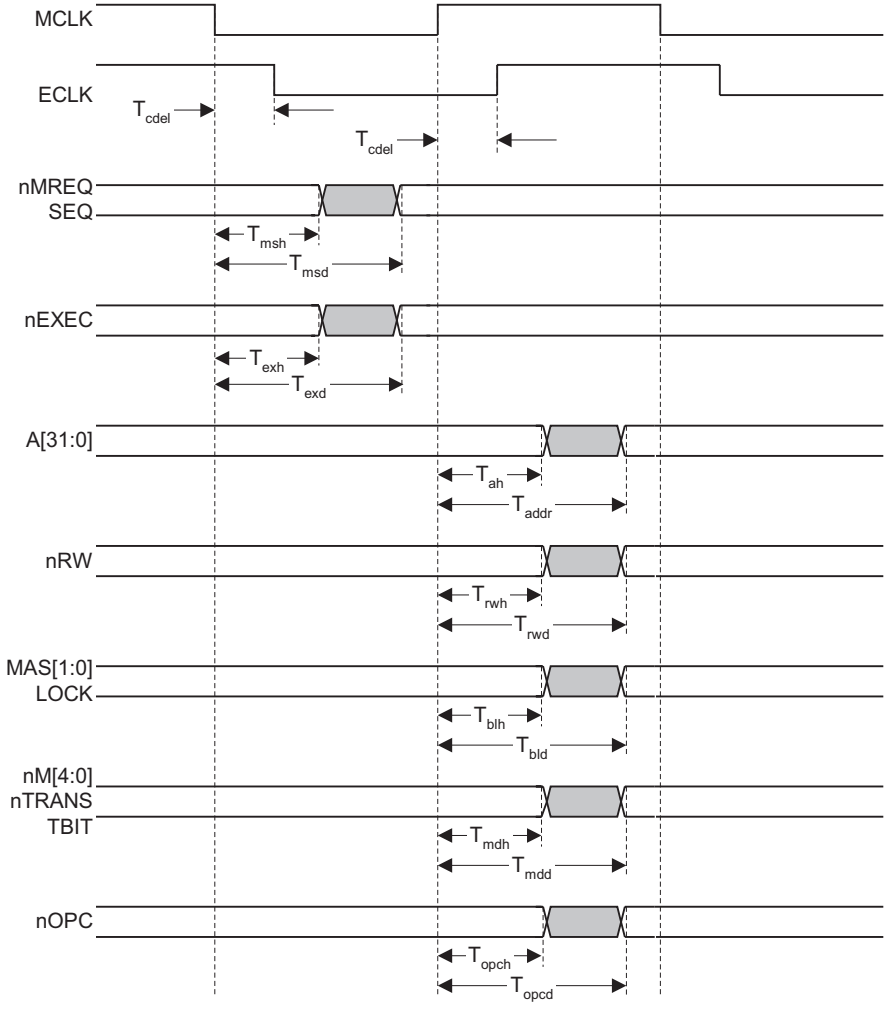

Figure 7-1 General timing

Note

In Figure 7-1 on page 7-4, **nWAIT**, **APE**, **ALE**, and **ABE** are all HIGH during the cycle shown. T_{cdel} is the delay, on either edge (whichever is greater), from the edge of **MCLK** to **ECLK**.

Table 7-1 General timing parameters

Symbol	Parameter	Parameter type
T_{addr}	**MCLK**r to address valid	Maximum
T_{ah}	Address hold time from **MCLK**r	Minimum
T_{bld}	**MCLK**r to **MAS[1:0**] and **LOCK**	Maximum
T_{blh}	**MAS[1:0]** and **LOCK** hold from **MCLK**r	Minimum
T_{cdel}	**MCLK** to **ECLK** delay	Maximum
T_{exd}	**MCLK**f to **nEXEC** valid	Maximum
T_{exh}	**nEXEC** hold time from **MCLK**f	Minimum
T_{mdd}	**MCLK**r to **nTRANS**, **nM[4:0**], and **TBIT** valid	Maximum
T_{mdh}	**nTRANS** and **nM[4:0]** hold time from **MCLK**r	Minimum
T_{msd}	**MCLK**f to **nMREQ** and **SEQ** valid	Maximum
T_{msh}	**nMREQ** and **SEQ** hold time from **MCLK**f	Minimum
T_{opcd}	**MCLK**r to **nOPC** valid	Maximum
T_{opch}	**nOPC** hold time from **MCLK**r	Minimum
T_{rwd}	**MCLK**r to **nRW** valid	Maximum
T_{rwh}	**nRW** hold time from **MCLK**r	Minimum

7.3 Address bus enable control

Figure 7-2 shows the ARM7TDMI ABE control timing. The timing parameters used in Figure 7-2 are listed in Table 7-2.

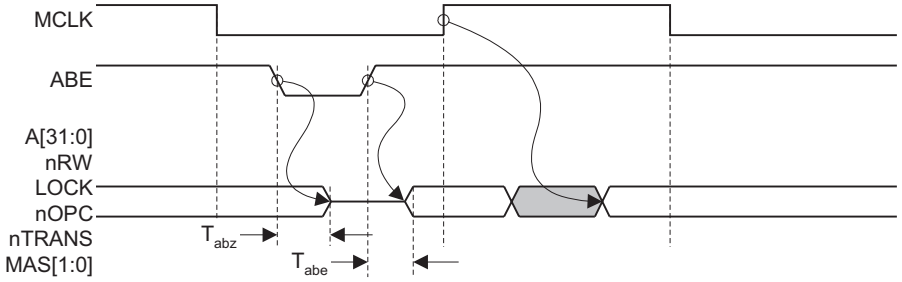

Figure 7-2 ABE control timing

Table 7-2 ABE control timing parameters

Symbol	Parameter	Parameter type
T_{abe}	Address bus enable time	Maximum
T_{abz}	Address bus disable time	Maximum

7.4 Bidirectional data write cycle

Figure 7-3 shows the ARM7TDMI processor bidirectional data write cycle timing. The timing parameters used in Figure 7-3 are listed in Table 7-3.

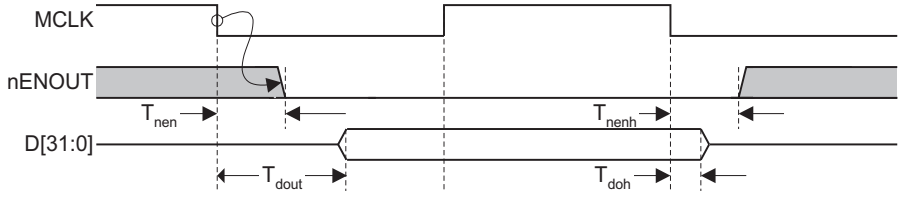

Figure 7-3 Bidirectional data write cycle timing

Note

In Figure 7-3 **DBE** is HIGH and **nENIN** is LOW during the cycle shown.

Table 7-3 Bidirectional data write cycle timing parameters

Symbol	Parameter	Parameter type
T_{doh}	**DOUT[31:0]** hold from **MCLK**f	Minimum
T_{dout}	**MCLK**f to **D[31:0]** valid	Maximum
T_{nen}	**MCLK**f to **nENOUT** valid	Maximum
T_{nenh}	**nENOUT** hold time from **MCLK**f	Minimum

7.5 Bidirectional data read cycle

Figure 7-4 shows the ARM7TDMI processor bidirectional data read cycle timing. The timing parameters used in Figure 7-4 are listed in Table 7-4.

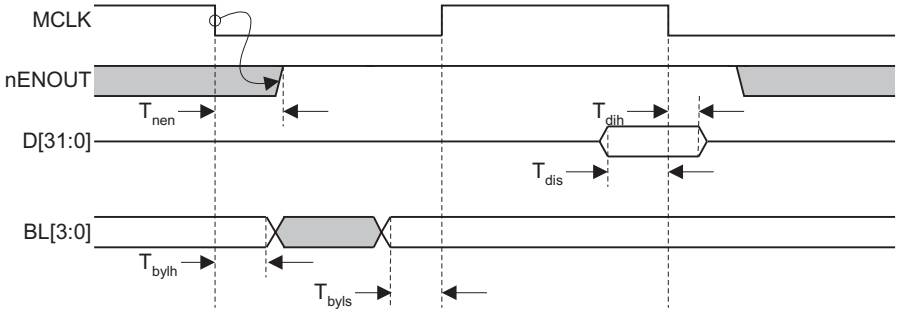

Figure 7-4 Bidirectional data read cycle timing

Note

In Figure 7-4**, DBE** is HIGH and **nENIN** is LOW during the cycle shown.

Table 7-4 Bidirectional data read cycle timing parameters

Symbol	Parameter	Parameter type
T_{bylh}	**BL[3:0]** hold time from **MCLK**f	Minimum
T_{byls}	**BL[3:0]** set up to from **MCLK**r	Minimum
T_{dih}	**DIN[31:0]** hold time from **MCLK**f	Minimum
T_{dis}	**DIN[31:0]** setup time to **MCLK**f	Minimum
T_{nen}	**MCLK**f to **nENOUT** valid	Maximum

7.6 Data bus control

Figure 7-5 shows the ARM7TDMI data bus control timing. The timing parameters used in Figure 7-5 are listed in Table 7-5.

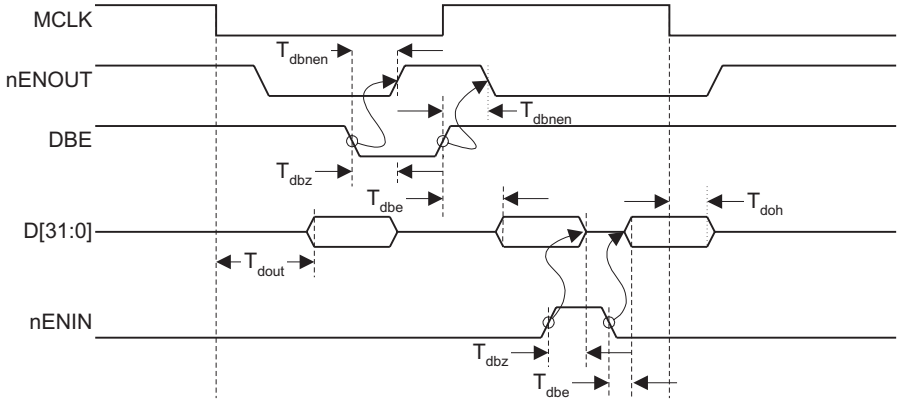

Figure 7-5 Data bus control timing

Note

The cycle shown in Figure 7-5 is a data write cycle because **nENOUT** was driven LOW during phase one. Here, **DBE** has first been used to modify the behavior of the data bus, and then **nENIN**.

Table 7-5 Data bus control timing parameters

Symbol	Parameter	Parameter type
T_{dbe}	Data bus enable time from **DBE**r	Maximum
T_{dbnen}	**DBE** to **nENOUT** valid	Maximum
T_{dbz}	Data bus disable time from **DBE**f	Maximum
T_{doh}	**DOUT[31:0]** hold from **MCLK**f	Minimum
T_{dout}	**MCLK**f to **D[31:0]** valid	Maximum

7.7 Output 3-state timing

Figure 7-6 shows the ARM7TDMI processor output 3-state timing. The timing parameters used in Figure 7-6 are listed in Table 7-6.

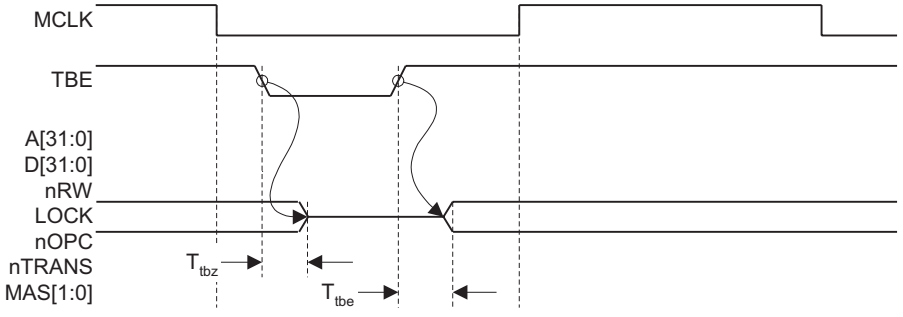

Figure 7-6 Output 3-state timing

Table 7-6 Output 3-state time timing parameters

Symbol	Parameter	Parameter type
T_{tbe}	Address and Data bus enable time from **TBE**r	Maximum
T_{tbz}	Address and Data bus disable time from **TBE**f	Maximum

7.8 Unidirectional data write cycle timing

Figure 7-7 shows the ARM7TDMI processor unidirectional data write cycle timing. The timing parameters used in Figure 7-6 are listed in Table 7-6.

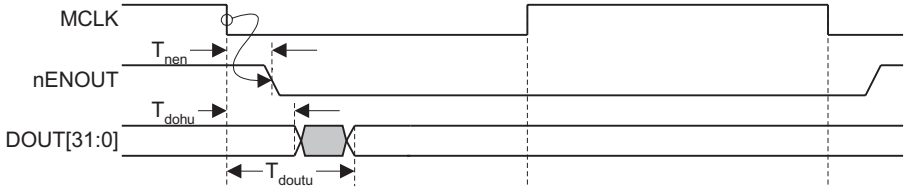

Figure 7-7 Unidirectional data write cycle timing

Table 7-7 Unidirectional data write cycle timing parameters

Symbol	Parameter	Parameter type
T_{dohu}	**DOUT[31:0**] hold time from **MCLK**f	Minimum
T_{doutu}	**MCLK**f to **DOUT[31:0]** valid	Maximum
T_{nen}	**MCLK**f to **nENOUT** valid	Maximum

7.9 Unidirectional data read cycle timing

Figure 7-8 shows the ARM7TDMI processor unidirectional data read cycle timing. The timing parameters used in Figure 7-7 are listed in Table 7-7.

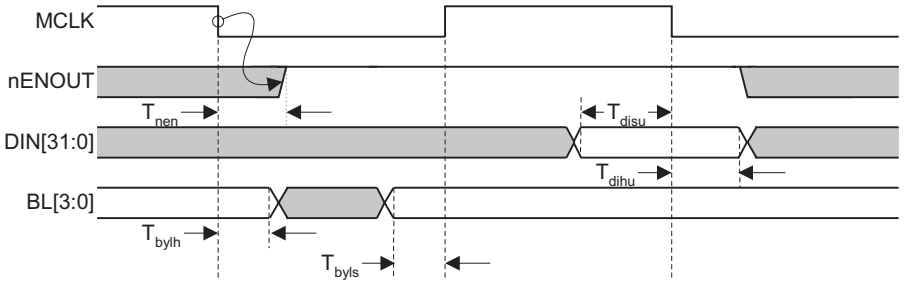

Figure 7-8 Unidirectional data read cycle timing

Table 7-8 Unidirectional data read cycle timing parameters

Symbol	Parameter	Parameter type
T_{bylh}	**BL[3:0]** hold time from **MCLK**f	Minimum
T_{byls}	**BL[3:0]** set up to from **MCLK**r	Minimum
T_{dihu}	**DIN[31:0]** hold time from **MCLK**f	Minimum
T_{disu}	**DIN[31:0]** set up time to **MCLK**f	Minimum
T_{nen}	**MCLK**f to **nENOUT** valid	Maximum

7.10 Configuration pin timing

Figure 7-9 shows the ARM7TDMI processor configuration pin timing. The timing parameters used in Figure 7-9 are listed in Table 7-9.

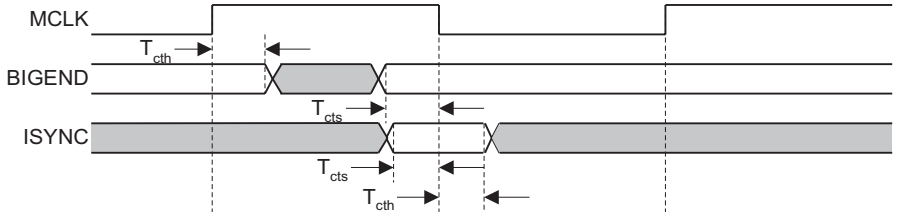

Figure 7-9 Configuration pin timing

Table 7-9 Configuration pin timing parameters

Symbol	Parameter	Parameter type
T_{cth}	Configurations hold time	Minimum
T_{cts}	Configuration setup time	Minimum

7.11 Coprocessor timing

Figure 7-10 shows the ARM7TDMI processor coprocessor timing. The timing parameters used in Figure 7-10 are listed in Table 7-10.

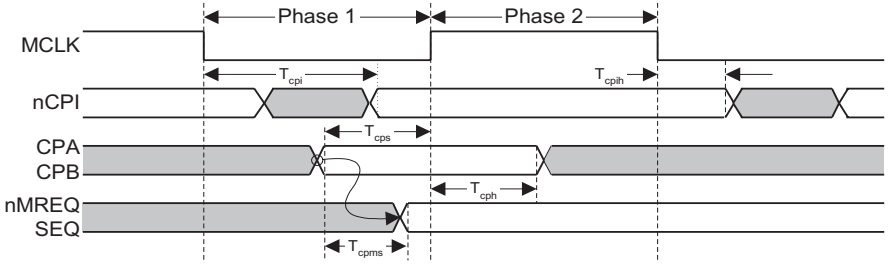

Figure 7-10 Coprocessor timing

Note

In Figure 7-10, usually **nMREQ** and **SEQ** become valid T_{msd} after the falling edge of **MCLK**. In this cycle the core has been busy-waiting for a coprocessor to complete the instruction. If **CPA** and **CPB** change during phase 1, the timing of **nMREQ** and **SEQ** depends on T_{cpms}. Most systems can generate **CPA** and **CPB** during the previous phase 2, and so the timing of **nMREQ** and **SEQ** is always T_{msd}.

Table 7-10 Coprocessor timing parameters

Symbol	Parameter	Parameter type
T_{cph}	**CPA**,**CPB** hold time from **MCLK**r	Minimum
T_{cpi}	**MCLK**f to **nCPI** valid	Maximum
T_{cpih}	**nCPI** hold time from **MCLK**f	Minimum
T_{cpms}	**CPA**, **CPB** to **nMREQ**, **SEQ**	Maximum
T_{cps}	**CPA**, **CPB** setup to **MCLK**r	Minimum

7.12 Exception timing

Figure 7-11 shows the ARM7TDMI processor exception timing. The timing parameters used in Figure 7-11 are listed in Table 7-11.

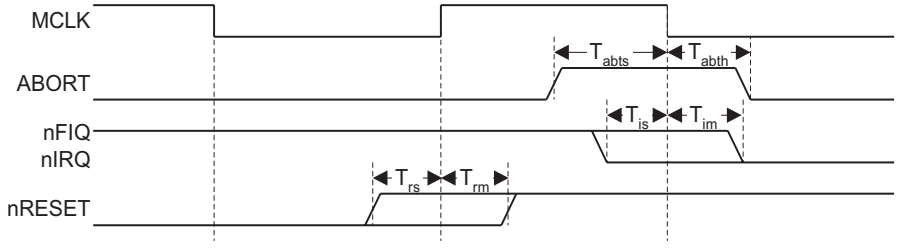

Figure 7-11 Exception timing

Note

In Figure 7-11, to guarantee recognition of the asynchronous interrupt (ISYNC=0) or reset source, the appropriate signals must be setup or held as follows:

- setup T_{is} and T_{rs} respectively before the corresponding clock edge
- hold T_{im} and T_{is} respectively after the corresponding clock edge.

These inputs can be applied fully asynchronously where the exact cycle of recognition is unimportant.

Table 7-11 Exception timing parameters

Symbol	Parameter	Parameter type
T_{abth}	**ABORT** hold time from **MCLK**f	Minimum
T_{abts}	**ABORT** set up time to **MCLK**f	Minimum
T_{im}	Asynchronous interrupt guaranteed nonrecognition time, with ISYNC=0	Maximum
T_{is}	Asynchronous interrupt set up time to **MCLK**f for guaranteed recognition, with ISYNC=0	Minimum
T_{rm}	Reset guaranteed nonrecognition time	Maximum
T_{rs}	Reset setup time to **MCLK**r for guaranteed recognition	Minimum

7.13 Synchronous interrupt timing

Figure 7-12 shows the ARM7TDMI processor synchronous interrupt timing. The timing parameters used in Figure 7-12 are listed in Table 7-12.

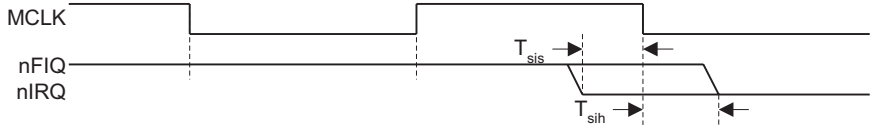

Figure 7-12 Synchronous interrupt timing

Table 7-12 Synchronous interrupt timing parameters

Symbol	Parameter	Parameter type
T_{sih}	Synchronous **nFIQ**, **nIRQ** hold from **MCLK**f with ISYNC=1	Minimum
T_{sis}	Synchronous **nFIQ**, **nIRQ** setup to **MCLK**f, with ISYNC=1	Minimum

7.14 Debug timing

Figure 7-13 shows the ARM7TDMI processor synchronous interrupt timing. The timing parameters used in Figure 7-13 are listed in Table 7-13.

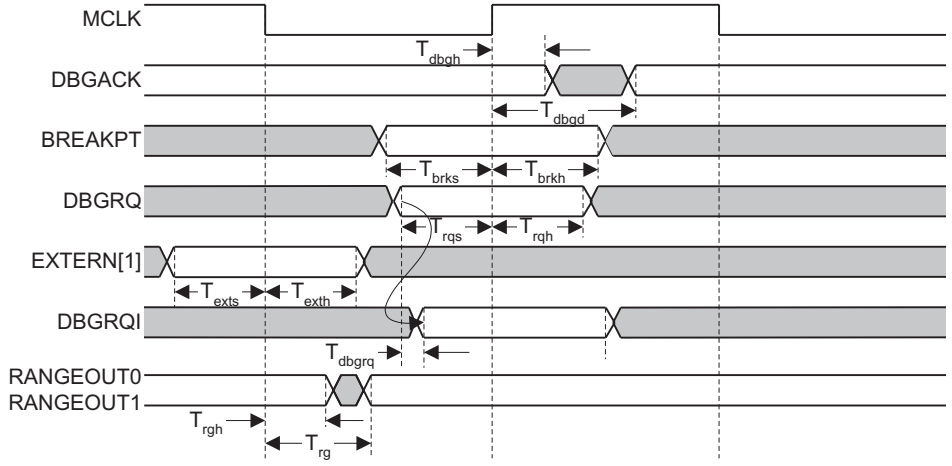

Figure 7-13 Debug timing

Table 7-13 Debug timing parameters

Symbol	Parameter	Parameter type
T_{brkh}	Hold time of **BREAKPT** from **MCLK**r	Minimum
T_{brks}	Set up time of **BREAKPT** to **MCLK**r	Minimum
T_{dbgd}	**MCLK**r to **DBGACK** valid	Maximum
T_{dbgh}	**DGBACK** hold time from **MCLK**r	Minimum
T_{dbgrq}	**DBGRQ** to **DBGRQI** valid	Maximum
T_{exth}	**EXTERN[1:0]** hold time from **MCLK**f	Minimum
T_{exts}	**EXTERN[1:0]** set up time to **MCLK**f	Minimum
T_{rg}	**MCLK**f to **RANGEOUT0**, **RANGEOUT1** valid	Maximum

Table 7-13 Debug timing parameters (continued)

Symbol	Parameter	Parameter type
T_{rgh}	**RANGEOUT0**, **RANGEOUT1** hold time from **MCLK**f	Minimum
T_{rqh}	**DBGRQ** guaranteed non-recognition time	Minimum
T_{rqs}	**DBGRQ** set up time to **MCLK**r for guaranteed recognition	Minimum

7.15 Debug communications channel output timing

Figure 7-14 shows the ARM7TDMI processor DCC output timing. The timing parameter used in Figure 7-14 is listed in Table 7-14.

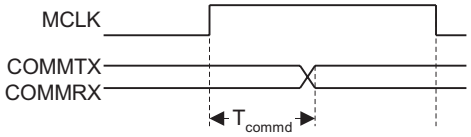

Figure 7-14 DCC output timing

Table 7-14 DCC output timing parameters

Symbol	Parameter	Parameter type
T_{commd}	**MCLK**r to **COMMRX**, **COMMTX** valid	Maximum

7.16 Breakpoint timing

Figure 7-15 shows the ARM7TDMI processor synchronous interrupt timing. The timing parameter used in Figure 7-12 is listed in Table 7-12.

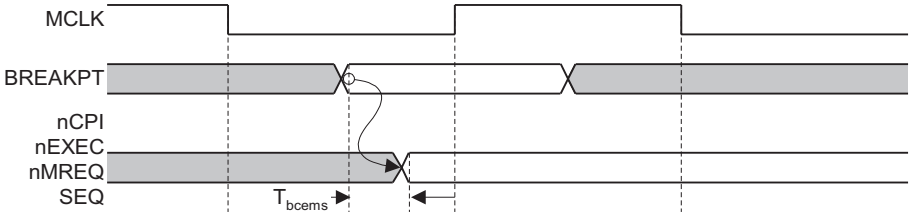

Figure 7-15 Breakpoint timing

Note

In Figure 7-15, **BREAKPT** changing in the LOW phase of **MCLK** (to signal a watchpointed store) affects **nCPI**, **nEXEC**, **nMREQ**, and **SEQ** in the same phase.

Table 7-15 Breakpoint timing parameters

Symbol	Parameter	Parameter type
T_{bcems}	**BREAKPT** to **nCPI**, **nEXEC**, **nMREQ**, **SEQ** delay	Maximum

7.17 Test clock and external clock timing

Figure 7-16 shows the ARM7TDMI processor test clock and external clock timing. The timing parameter used in Figure 7-16 is listed in Table 7-16.

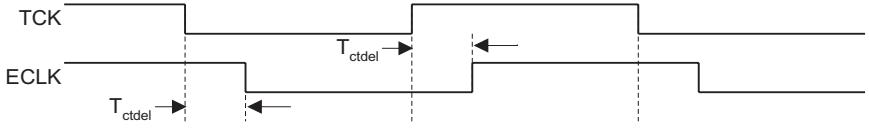

Figure 7-16 TCK and ECLK timing

Note

In Figure 7-16, T_{ctdel} is the delay, on either edge (whichever is greater), from the edge of **TCK** to **ECLK**.

Table 7-16 TCK and ECLK timing parameters

Symbol	Parameter	Parameter type
T_{ctdel}	**TCK** to **ECLK** delay	Maximum

7.18 Memory clock timing

Figure 7-17 shows the ARM7TDMI processor memory clock timing. The timing parameters used in Figure 7-17 are listed in Table 7-17.

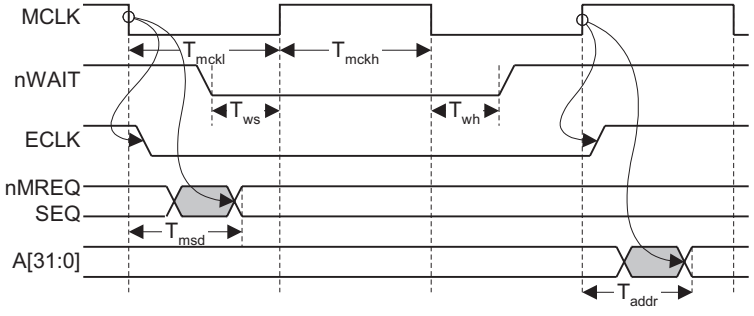

Figure 7-17 MCLK timing

Note

In Figure 7-17, the core is not clocked by the HIGH phase of **MCLK** when **nWAIT** is LOW. During the cycles shown, **nMREQ** and **SEQ** change once, during the first LOW phase of **MCLK**, and **A[31:0]** change once, during the second HIGH phase of **MCLK**. Phase 2 is shown for reference. This is the internal clock from which the core times all its activity. This signal is included to show how the HIGH phase of the external **MCLK** has been removed from the internal core clock.

Table 7-17 MCLK timing parameters

Symbol	Parameter	Parameter type
T_{addr}	**MCLK**r to address valid	Maximum
T_{mckh}	**MCLK** HIGH time	Minimum
T_{mckl}	**MCLK** LOW time	Minimum
T_{msd}	**MCLK**f to **nMREQ** and **SEQ** valid	Maximum
T_{wh}	**nWAIT** hold from **MCLK**f	Minimum
T_{ws}	**nWAIT** setup to **MCLK**r	Minimum

7.19 Boundary scan general timing

Figure 7-18 shows the ARM7TDMI processor boundary scan general timing. The timing parameters used in Figure 7-18 are listed in Table 7-18.

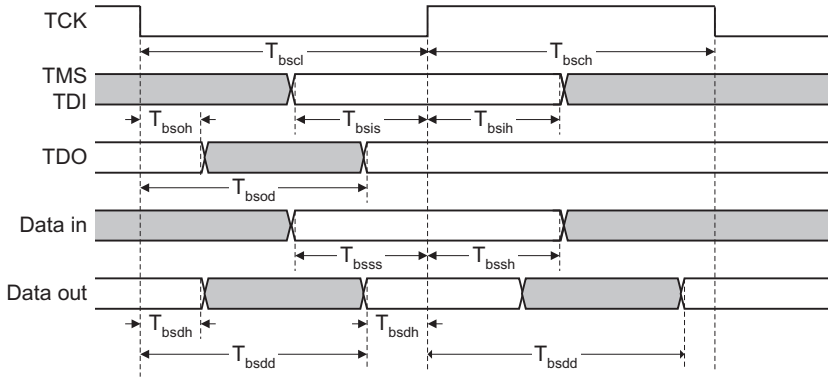

Figure 7-18 Boundary scan general timing

Table 7-18 Boundary scan general timing parameters

Symbol	Parameter	Parameter type
T_{bsch}	**TCK** high period	Minimum
T_{bscl}	**TCK** low period	Minimum
T_{bsdd}	**TCK** to data output valid	Maximum
T_{bsdh}	Data output hold time from **TCK**	Minimum
T_{bsih}	**TDI**, **TMS** hold from **TCK**r	Minimum
T_{bsis}	**TDI**, **TMS** setup to **TCK**r	Minimum
T_{bsod}	**TCK**f to **TDO** valid	Maximum
T_{bsoh}	**TDO** hold time from **TCK**f	Minimum
T_{bssh}	I/O signal setup from **TCK**r	Minimum
T_{bsss}	I/O signal setup to **TCK**r,	Minimum

7.20 Reset period timing

Figure 7-19 shows the ARM7TDMI reset period timing. The timing parameters used in Figure 7-19 are listed in Table 7-19.

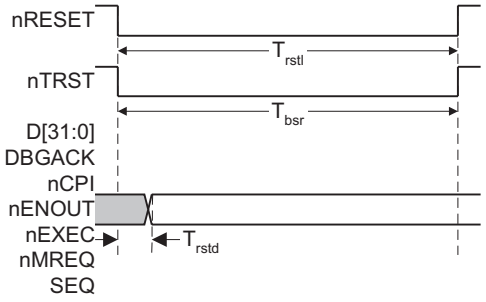

Figure 7-19 Reset period timing

Table 7-19 Reset period timing parameters

Symbol	Parameter	Parameter type
T_{bsr}	**nTRST** reset period	Minimum
T_{rstd}	**nRESET**f to **D[31:0]**, **DBGACK**, **nCPI**, **nENOUT**, **nEXEC**, **nMREQ**, **SEQ** valid	Maximum
T_{rstl}	**nRESET** LOW for guaranteed reset	Minimum

7.21 Output enable and disable times

Figure 7-20 shows the output enable and disable times due to a HIGHZ TAP instruction. Figure 7-21 shows the output enable and disable times due to data scanning.The timing parameters used in Figure 7-20 and Figure 7-21 are listed in Table 7-20.

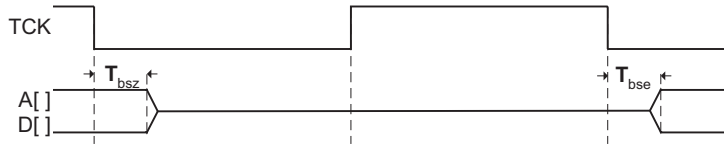

Figure 7-20 Output enable and disable times due to HIGHZ TAP instruction

Note

Figure 7-20 shows the T_{bse}, output enable time, parameter and T_{bsz}, output disable time, when the HIGHZ TAP instruction is loaded into the instruction register.

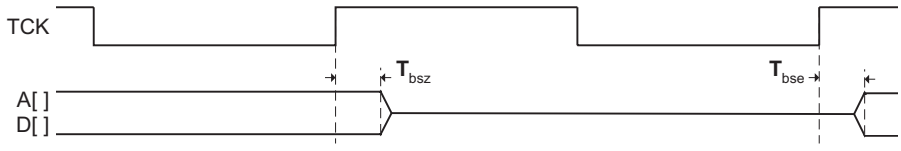

Figure 7-21 Output enable and disable times due to data scanning

Note

Figure 7-21 shows the T_{bse}, output enable time, parameter and T_{bsz}, output disable time when data scanning, due to different logic levels being scanned through the scan cells for **ABE** and **DBE**.

Table 7-20 Output enable and disable timing parameters

Symbol	Parameter	Parameter type
T_{bse}	Output enable time	Maximum
T_{bsz}	Output disable time	Maximum

7.22 Address latch enable control

Figure 7-22 shows the ARM7TDMI reset period timing. The timing parameters used in Figure 7-22 are listed in Table 7-21.

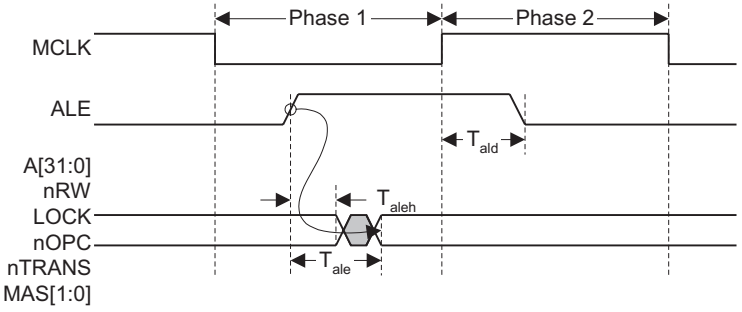

Figure 7-22 ALE control timing

Note

In Figure 7-22, T_{ald} is the time by which **ALE** must be driven LOW to latch the current address in phase 2. If **ALE** is driven LOW after T_{ald}, then a new address is latched. This is known as *address breakthrough*.

Table 7-21 ALE address control timing parameters

Symbol	Parameter	Parameter type
T_{ald}	Address group latch output time	Maximum
T_{ale}	Address group latch open output delay	Maximum
T_{aleh}	Address group latch output hold time	Minimum

7.23 Address pipeline control timing

Figure 7-23 shows the ARM7TDMI APE control timing. The timing parameters used in Figure 7-23 are listed in Table 7-22.

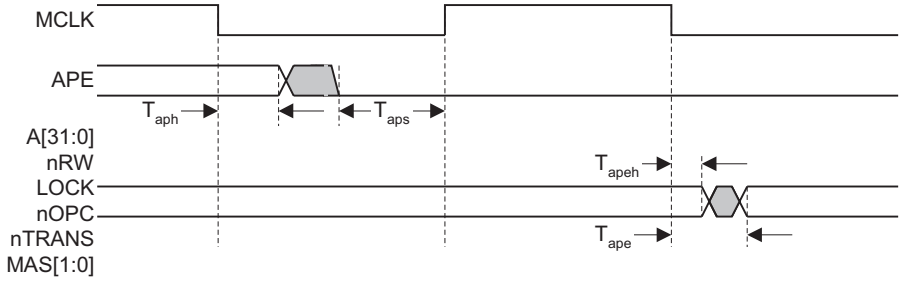

Figure 7-23 APE control timing

Table 7-22 APE control timing parameters

Symbol	Parameter	Parameter type
T_{ape}	**MCLK**f to address group valid	Maximum
T_{apeh}	Address group output hold time from **MCLK**f	Minimum
T_{aph}	**APE** hold time from **MCLK**f	Minimum
T_{aps}	**APE** set up time to **MCLK**r	Minimum

7.24 Notes on AC Parameters

Table 7-23 lists the AC timing parameters in alphabetical order.

Contact your supplier for AC timing parameter values.

In Table 7-23:

- the letter f at the end of a signal name indicates the falling edge
- the letter r at the end of a signal name indicates the rising edge.

Table 7-23 AC timing parameters used in this chapter

Symbol	Parameter	Parameter Type	Figure cross reference
T_{abe}	Address bus enable time	Maximum	Figure 7-2
T_{abth}	**ABORT** hold time from **MCLK**f	Minimum	Figure 7-11
T_{abts}	**ABORT** set up time to **MCLK**f	Minimum	Figure 7-11
T_{abz}	Address bus disable time	Maximum	Figure 7-2
T_{addr}	**MCLK**r to address valid	Maximum	Figure 7-1 Figure 7-17
T_{ah}	Address hold time from **MCLK**r	Minimum	Figure 7-1
T_{ald}	Address group latch time	Maximum	Figure 7-22
T_{ale}	Address group latch open output delay	Maximum	Figure 7-22
T_{aleh}	Address group latch output hold time	Minimum	Figure 7-22
T_{ape}	**MCLK**f to address group valid	Maximum	Figure 7-23
T_{apeh}	Address group output hold time from **MCLK**f	Minimum	Figure 7-23
T_{aph}	**APE** hold time from **MCLK**f	Minimum	Figure 7-23
T_{aps}	**APE** set up time to **MCLK**r	Minimum	Figure 7-23
T_{bcems}	**BREAKPT** to **nCPI**, **nEXEC**, **nMREQ**, **SEQ** delay	Maximum	Figure 7-13
T_{bld}	**MCLK**r to **MAS[1:0]** and **LOCK**	Maximum	Figure 7-1
T_{blh}	**MAS[1:0]** and **LOCK** hold from **MCLK**r	Minimum	Figure 7-1
T_{brkh}	Hold time of **BREAKPT** from **MCLK**r	Minimum	Figure 7-13

Table 7-23 AC timing parameters used in this chapter (continued)

Symbol	Parameter	Parameter Type	Figure cross reference
T_{brks}	Set up time of **BREAKPT** to **MCLK**r	Minimum	Figure 7-13
T_{bsch}	**TCK** high period	Minimum	Figure 7-18
T_{bscl}	**TCK** low period	Minimum	Figure 7-18
T_{bsdd}	**TCK** to data output valid	Maximum	Figure 7-18
T_{bsdh}	Data output hold time from **TCK**	Minimum	Figure 7-18
T_{bse}	Output enable time	Maximum	Figure 7-20 Figure 7-21
T_{bsih}	**TDI**, **TMS** hold from **TCK**r	Minimum	Figure 7-18
T_{bsis}	**TDI**, **TMS** setup to **TCK**r	Minimum	Figure 7-18
T_{bsod}	**TCK**f to **TDO** valid	Maximum	Figure 7-18
T_{bsoh}	**TDO** hold time from **TCK**f	Minimum	Figure 7-18
T_{bsr}	**nTRST** reset period	Minimum	Figure 7-19
T_{bssh}	I/O signal setup from **TCK**r	Minimum	Figure 7-18
T_{bsss}	I/O signal setup to **TCK**r,	Minimum	Figure 7-18
T_{bsz}	Output disable time	Maximum	Figure 7-20 Figure 7-21
T_{bylh}	**BL[3:0]** hold time from **MCLK**f	Minimum	Figure 7-4 Figure 7-8
T_{byls}	**BL[3:0]** set up to from **MCLK**r	Minimum	Figure 7-4 Figure 7-8
T_{cdel}	**MCLK** to **ECLK** delay	Maximum	Figure 7-1
T_{clkbs}	**TCK** to boundary scan clocks	Maximum	-
T_{commd}	**MCLK**r to **COMMRX**, **COMMTX** valid	Maximum	Figure 7-14
T_{cph}	**CPA**,**CPB** hold time from **MCLK**r	Minimum	Figure 7-10
T_{cpi}	**MCLK**f to **nCPI** valid	Maximum	Figure 7-10

Table 7-23 AC timing parameters used in this chapter (continued)

Symbol	Parameter	Parameter Type	Figure cross reference
T_{cpih}	**nCPI** hold time from **MCLK**f	Minimum	Figure 7-10
T_{cpms}	**CPA**, **CPB** to **nMREQ**, **SEQ**	Maximum	Figure 7-10
T_{cps}	**CPA**, **CPB** setup to **MCLK**r	Minimum	Figure 7-10
T_{ctdel}	**TCK** to **ECLK** delay	Maximum	Figure 7-16
T_{cth}	Config hold time	Minimum	Figure 7-9
T_{cts}	Config setup time	Minimum	Figure 7-9
T_{dbe}	Data bus enable time from **DBE**r	Maximum	Figure 7-5
T_{dbgd}	**MCLK**r to **DBGACK** valid	Maximum	Figure 7-13
T_{dbgh}	**DGBACK** hold time from **MCLK**r	Minimum	Figure 7-13
T_{dbgrq}	**DBGRQ** to **DBGRQI** valid	Maximum	Figure 7-13
T_{dbnen}	**DBE** to **nENOUT** valid	Maximum	Figure 7-5
T_{dbz}	Data bus disable time from **DBE**f	Maximum	Figure 7-5
T_{dckf}	**DCLK** induced, **TCK**f to various outputs valid	Maximum	-
T_{dckfh}	**DCLK** induced, various outputs hold from **TCK**f	Minimum	-
T_{dckr}	**DCLK** induced, **TCK**r to various outputs valid	Maximum	-
T_{dckrh}	**DCLK** induced, various outputs hold from **TCK**r	Minimum	-
T_{dih}	**DIN[31:0]** hold time from **MCLK**f	Minimum	Figure 7-4
T_{dihu}	**DIN[31:0]** hold time from **MCLK**f	Minimum	Figure 7-8
T_{dis}	**DIN[31:0]** setup time to **MCLKf**	Minimum	Figure 7-4
T_{disu}	**DIN[31:0]** set up time to **MCLK**f	Minimum	Figure 7-8
T_{doh}	**DOUT[31:0]** hold from **MCLK**f	Minimum	Figure 7-3 Figure 7-5
T_{dohu}	**DOUT[31:0**] hold time from **MCLK**f	Minimum	Figure 7-7

Table 7-23 AC timing parameters used in this chapter (continued)

Symbol	Parameter	Parameter Type	Figure cross reference
T_{dout}	**MCLK**f to **D[31:0]** valid	Maximum	Figure 7-3 Figure 7-5
T_{doutu}	**MCLK**f to **DOUT[31:0]** valid	Maximum	Figure 7-7
T_{ecapd}	**TCK** to **ECAPCLK** changing	Maximum	-
T_{exd}	**MCLK**f to **nEXEC** valid	Maximum	Figure 7-1
T_{exh}	**nEXEC** hold time from **MCLK**f	Minimum	Figure 7-1
T_{exth}	**EXTERN[1:0]** hold time from **MCLK**f	Minimum	Figure 7-13
T_{exts}	**EXTERN[1:0]** set up time to **MCLK**f	Minimum	Figure 7-13
T_{im}	Asynchronous interrupt guaranteed nonrecognition time, with **ISYNC**=0	Maximum	Figure 7-11
T_{is}	Asynchronous interrupt set up time to **MCLK**f for guaranteed recognition, with **ISYNC**=0	Minimum	Figure 7-11
T_{mckh}	**MCLK** HIGH time	Minimum	Figure 7-17
T_{mckl}	**MCLK** LOW time	Minimum	Figure 7-17
T_{mdd}	**MCLK**r to **nTRANS**, **nM[4:0**], and **TBIT** valid	Maximum	Figure 7-1
T_{mdh}	**nTRANS** and **nM[4:0]** hold time from **MCLK**r	Minimum	Figure 7-1
T_{msd}	**MCLK**f to **nMREQ** and **SEQ** valid	Maximum	Figure 7-1 Figure 7-17
T_{msh}	**nMREQ** and **SEQ** hold time from **MCLK**f	Minimum	Figure 7-1
T_{nen}	**MCLK**f to **nENOUT** valid	Maximum	Figure 7-3 Figure 7-4 Figure 7-7 Figure 7-8
T_{nenh}	**nENOUT** hold time from **MCLK**f	Minimum	Figure 7-3
T_{opcd}	**MCLK**r to **nOPC** valid	Maximum	Figure 7-1
T_{opch}	**nOPC** hold time from **MCLK**r	Minimum	Figure 7-1
T_{rg}	**MCLK**f to **RANGEOUT0**, **RANGEOUT1** valid	Maximum	Figure 7-13

Table 7-23 AC timing parameters used in this chapter (continued)

Symbol	Parameter	Parameter Type	Figure cross reference
T_{rgh}	**RANGEOUT0**, **RANGEOUT1** hold time from **MCLK**f	Minimum	Figure 7-13
T_{rm}	Reset guaranteed nonrecognition time	Maximum	Figure 7-11
T_{rqh}	**DBGRQ** guaranteed non-recognition time	Minimum	Figure 7-13
T_{rqs}	**DBGRQ** set up time to **MCLK**r for guaranteed recognition	Minimum	Figure 7-13
T_{rs}	Reset setup time to **MCLK**r for guaranteed recognition	Minimum	Figure 7-11
T_{rstd}	**nRESET**f to **D[31:0]**, **DBGACK**, **nCPI**, **nENOUT**, **nEXEC**, **nMREQ**, **SEQ** valid	Maximum	Figure 7-19
T_{rstl}	**nRESET** LOW for guaranteed reset	Minimum	Figure 7-19
T_{rwd}	**MCLK**r to **nRW** valid	Maximum	Figure 7-1
T_{rwh}	**nRW** hold time from **MCLK**r	Minimum	Figure 7-1
T_{sdtd}	**SDOUTBS** to **TDO** valid	Maximum	-
T_{shbsf}	**TCK** to **SHCLKBS**, **SHCLK2BS** falling	Maximum	-
T_{shbsr}	**TCK** to **SHCLKBS**, **SHCLK2BS** rising	Maximum	-
T_{sih}	Synchronous **nFIQ**, **nIRQ** hold from **MCLK**f with **ISYNC**=1	Minimum	Figure 7-12
T_{sis}	Synchronous **nFIQ**, **nIRQ** setup to **MCLK**f, with **ISYNC**=1	Minimum	Figure 7-12
T_{tbe}	Address and Data bus enable time from **TBE**r	Maximum	Figure 7-6
T_{tbz}	Address and Data bus disable time from **TBE**f	Maximum	Figure 7-6
T_{tckf}	**TCK** to **TCK1**, **TCK2** falling	Maximum	-
T_{tckr}	**TCK** to **TCK1**, **TCK2** rising	Maximum	-
T_{tdbgd}	**TCK** to **DBGACK**, **DBGRQI** changing	Maximum	-
T_{tpfd}	**TCK**f to TAP outputs	Maximum	-
T_{tpfh}	TAP outputs hold time from **TCK**f	Minimum	-
T_{tprd}	**TCK**r to TAP outputs	Maximum	-
T_{tprh}	TAP outputs hold time from **TCK**r	Minimum	-

Table 7-23 AC timing parameters used in this chapter (continued)

Symbol	Parameter	Parameter Type	Figure cross reference
T_{trstd}	**nTRST**f to every output valid	Maximum	-
T_{trstd}	**nTRST**f to TAP outputs valid	Maximum	-
T_{trsts}	**nTRST**r setup to **TCK**r	Maximum	-
T_{wh}	**nWAIT** hold from **MCLK**f	Minimum	Figure 7-17
T_{ws}	**nWAIT** setup to **MCLK**r	Minimum	Figure 7-17

7.25 DC parameters

Contact your supplier for information on:

- operating conditions
- maximum ratings.

Appendix A
Signal Description

This appendix lists and describes the signals for the ARM7TDMI processor. It contains the following section:

- *Signal description* on page A-2.

A.1 Signal description

This section describes all of the signals for the ARM7TDMI processor.

A.1.1 Transistor dimensions

Table A-1 on page A-2 lists the transistor sizes for a 0.18 µm ARM7TDMI processor.

Table A-1 Transistor sizes

Driver	Layer	Width	Length
INV4	P	p = 7.45µm	0.18µm
	N	N = 4.2µm	0.18µm
INV8	P	p = 14.9µm	0.18µm
	N	N = 8.1µm	0.18µm

A.1.2 Signal types

Table A-2 on page A-2 lists the signal types used in this appendix.

Table A-2 Signal types

Type	Description
IC	Input CMOS thresholds
P	Power
O4	Output with INV4 inverter
O8	Output with INV8 inverter

A.1.3 Signals

Table A-3 lists and describes all of the signals used for the ARM7TDMI processor.

Table A-3 Signal Descriptions

Name	Type	Description
A[31:0] Addresses	O8	This is the 32-bit address bus. **ALE**, **ABE**, and **APE** are used to control when the address bus is valid.
ABE Address bus enable	IC	The address bus drivers are disabled when this is LOW, putting the address bus into a high impedance state. This also controls the LOCK, **MAS[1:0]**, **nRW**, **nOPC**, and **nTRANS** signals in the same way. **ABE** must be tied HIGH if there is no system requirement to disable the address drivers.
ABORT Memory abort	IC	The memory system uses this signal to tell the processor that a requested access is not allowed.
ALE Address latch enable	IC	This signal is provided for backwards compatibility with older ARM processors. For new designs, if address retiming is required, ARM Limited recommends the use of **APE**, and for **ALE** to be connected HIGH. The address bus, **LOCK**, **MAS[1:0]**, **nRW**, **nOPC**, and **nTRANS** signals are latched when this is held LOW. This allows these address signals to be held valid for the complete duration of a memory access cycle. For example, when interfacing to ROM, the address must be valid until after the data has been read.
APE Address pipeline enable	IC	Selects whether the address bus, **LOCK**, **MAS[1:0]**, **nRW**, **nTRANS**, and **nOPC** signals operate in pipelined (**APE** is HIGH) or depipelined mode (**APE** is LOW). Pipelined mode is particularly useful for DRAM systems, where it is desirable to provide the address to the memory as early as possible, to allow longer periods for address decoding and the generation of DRAM control signals. In this mode, the address bus does not remain valid to the end of the memory cycle. Depipelined mode can be useful for SRAM and ROM access. Here the address bus, **LOCK**, **MAS[1:0]**, **nRW**, **nTRANS**, and **nOPC** signals must be kept stable throughout the complete memory cycle. However, this does not provide optimum performance. See *Address timing* on page 3-14 for details of this timing.
BIGEND Big endian configuration	IC	Selects how the processor treats bytes in memory: • HIGH for big-endian format • LOW for little-endian format.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
BL[3:0] Byte latch control	IC	The values on the data bus are latched on the falling edge of **MCLK** when these signals are HIGH. For most designs these signals must be tied HIGH.
BREAKPT Breakpoint	IC	A conditional request for the processor to enter debug state is made by placing this signal HIGH. If the memory access at that time is an instruction fetch, the processor enters debug state only if the instruction reaches the execution stage of the pipeline. If the memory access is for data, the processor enters debug state after the current instruction completes execution. This allows extension of the internal breakpoints provided by the EmbeddedICE Logic. See *Behavior of the program counter during debug* on page B-29 for details on the use of this signal.
BUSDIS Bus disable	O4	When INTEST is selected on scan chain 0, 4, or 8 this is HIGH. It can be used to disable external logic driving onto the bidirectional data bus during scan testing. This signal changes after the falling edge of **TCK**.
BUSEN Data bus configuration	IC	A static configuration signal that selects whether the bidirectional data bus (**D[31:0]**) or the unidirectional data busses (**DIN[31:0]** and **DOUT[31:0]**) are used for transfer of data between the processor and memory. When **BUSEN** is LOW, **D[31:0]** is used; **DOUT[31:0]** is driven to a value of zero, and **DIN[31:0]** is ignored, and must be tied LOW. When **BUSEN** is HIGH, **DIN[31:0]** and **DOUT[31:0]** are used; **D[31:0]** is ignored and must be left unconnected. See Chapter 3 *Memory Interface* for details on the use of this signal.
COMMRX Communications channel receive	O4	When the communications channel receive buffer is full this is HIGH. This signal changes after the rising edge of **MCLK**. See *Debug Communications Channel* on page 5-16 for more information.
COMMTX Communications channel transmit	O4	When the communications channel transmit buffer is empty this is HIGH. This signal changes after the rising edge of **MCLK**. See *Debug Communications Channel* on page 5-16 for more information.
CPA Coprocessor absent	IC	Placed LOW by the coprocessor if it is capable of performing the operation requested by the processor.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
CPB Coprocessor busy	IC	Placed LOW by the coprocessor when it is ready to start the operation requested by the processor. It is sampled by the processor when **MCLK** goes HIGH in each cycle in which **nCPI** is LOW.
D[31:0] Data bus	IC O8	Used for data transfers between the processor and external memory. During read cycles input data must be valid on the falling edge of **MCLK**. During write cycles output data remains valid until after the falling edge of **MCLK**. This bus is always driven except during read cycles, irrespective of the value of **BUSEN**. Consequently it must be left unconnected if using the unidirectional data buses. See Chapter 3 *Memory Interface*.
DBE Data bus enable	IC	Must be HIGH for data to appear on either the bidirectional or unidirectional data output bus. When LOW the bidirectional data bus is placed into a high impedance state and data output is prevented on the unidirectional data output bus. It can be used for test purposes or in shared bus systems.
DBGACK Debug acknowledge	O4	When the processor is in a debug state this is HIGH.
DBGEN Debug enable	IC	A static configuration signal that disables the debug features of the processor when held LOW. This signal must be HIGH to allow the EmbeddedICE Logic to function.
DBGRQ Debug request	IC	This is a level-sensitive input, that when HIGH causes ARM7TDMI core to enter debug state after executing the current instruction. This allows external hardware to force the ARM7TDMI core into debug state, in addition to the debugging features provided by the EmbeddedICE Logic. See Appendix B *Debug in Depth*.
DBGRQI Internal debug request	O4	This is the logical OR of **DBGRQ** and bit 1 of the debug control register.
DIN[31:0] Data input bus	IC	Unidirectional bus used to transfer instructions and data from the memory to the processor. This bus is only used when **BUSEN** is HIGH. If unused then it must be tied LOW. This bus is sampled during read cycles on the falling edge of **MCLK**.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
DOUT[31:0] Data output bus	O8	Unidirectional bus used to transfer data from the processor to the memory system. This bus is only used when **BUSEN** is HIGH. Otherwise it is driven to a value of zero. During write cycles the output data becomes valid while **MCLK** is LOW, and remains valid until after the falling edge of **MCLK**.
DRIVEBS Boundary scan cell enable	O4	Controls the multiplexors in the scan cells of an external boundary-scan chain. This must be left unconnected, if an external boundary-scan chain is not connected.
ECAPCLK EXTEST capture clock	O4	Only used on the ARM7TDMI test chip, and must otherwise be left unconnected.
ECAPCLKBS EXTEST capture clock for boundary-scan	O4	Used to capture the device inputs of an external boundary-scan chain during EXTEST. When scan chain 3 is selected, the current instruction is EXTEST and the TAP controller state machine is in the CAPTURE- DR state, then this signal is a pulse equal in width to **TCK2**. This must be left unconnected, if an external boundary-scan chain is not connected.
ECLK External clock output	O4	In normal operation, this is simply **MCLK**, optionally stretched with **nWAIT**, exported from the core. When the core is being debugged, this is **DCLK**, which is generated internally from **TCK**.
EXTERN0 External input 0	IC	This is connected to the EmbeddedICE Logic and allows breakpoints and watchpoints to be dependent on an external condition.
EXTERN1 External input 1	IC	This is connected to the EmbeddedICE Logic and allows breakpoints and watchpoints to be dependent on an external condition.
HIGHZ High impedance	O4	When the HIGHZ instruction has been loaded into the TAP controller this signal is HIGH. See Appendix B *Debug in Depth* for details.
ICAPCLKBS INTEST capture clock	O4	This is used to capture the device outputs in an external boundary-scan chain during INTEST. This must be left unconnected, if an external boundary-scan chain is not connected.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
IR[3:0] TAP controller instruction register	O4	Reflects the current instruction loaded into the TAP controller instruction register. These bits change on the falling edge of **TCK** when the state machine is in the UPDATE-IR state. The instruction encoding is described in *Public instructions* on page B-9.
ISYNC Synchronous interrupts	IC	Set this HIGH if **nIRQ** and **nFIQ** are synchronous to the processor clock; LOW for asynchronous interrupts.
LOCK Locked operation	O8	When the processor is performing a locked memory access this is HIGH. This is used to prevent the memory controller allowing another device to access the memory. It is active only during the data swap (SWP) instruction. This is one of the signals controlled by **APE**, **ALE** and **ABE**.
MAS[1:0] Memory access size	O8	Used to indicate to the memory system the size of data transfer (byte, halfword or word) required for both read and write cycles, become valid before the falling edge of **MCLK** and remain valid until the rising edge of **MCLK** during the memory cycle. The binary values 00, 01, and 10 represent byte, halfword and word respectively (11 is reserved). This is one of the signals controlled by **APE**, **ALE**, and **ABE**.
MCLK Memory clock input	IC	This is the main clock for all memory accesses and processor operations. The clock speed can be reduced to allow access to slow peripherals or memory. Alternatively, the **nWAIT** can be used with a free-running **MCLK** to achieve the same effect.
nCPI Not coprocessor instruction	O4	LOW when a coprocessor instruction is processed. The processor then waits for a response from the coprocessor on the **CPA** and **CPB** lines. If **CPA** is HIGH when **MCLK** rises after a request has been initiated by the processor, then the coprocessor handshake is aborted, and the processor enters the undefined instruction trap. If **CPA** is LOW at this time, then the processor will enters a busy-wait period until **CPB** goes LOW before completing the coprocessor handshake.
nENIN NOT enable input	IC	This must be LOW for the data bus to be driven during write cycles. Can be used in conjunction with **nENOUT** to control the data bus during write cycles. See Chapter 3 *Memory Interface*.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
nENOUT Not enable output	O4	During a write cycle, this signal is driven LOW before the rising edge of **MCLK**, and remains LOW for the entire cycle. This can be used to aid arbitration in shared bus applications. See Chapter 3 *Memory Interface*.
nENOUTI Not enable output	O4	During a coprocessor register transfer C-cycle from the EmbeddedICE communications channel coprocessor to the ARM core, this signal goes LOW. This can be used to aid arbitration in shared bus systems.
nEXEC Not executed	O4	This is HIGH when the instruction in the execution unit is not being executed because, for example, it has failed its condition code check.
nFIQ Not fast interrupt request	IC	Taking this LOW causes the processor to be interrupted if the appropriate enable in the processor is active. The signal is level-sensitive and must be held LOW until a suitable response is received from the processor. **nFIQ** can be synchronous or asynchronous to **MCLK**, depending on the state of **ISYNC**.
nHIGHZ Not **HIGHZ**	O4	When the current instruction is **HIGHZ** this signal is LOW. This is used to place the scan cells of that scan chain in the high impedance state. This must be left unconnected, if an external boundary-scan chain is not connected.
nIRQ Not interrupt request	IC	As **nFIQ**, but with lower priority. Can be taken LOW to interrupt the processor when the appropriate enable is active. **nIRQ** can be synchronous or asynchronous, depending on the state of **ISYNC**.
nM[4:0] Not processor mode	O4	These are the inverse of the internal status bits indicating the current processor mode.
nMREQ Not memory request	O4	When the processor requires memory access during the following cycle this is LOW.
nOPC Not op-code fetch	O8	When the processor is fetching an instruction from memory this is LOW. This is one of the signals controlled by **APE**, **ALE**, and **ABE**.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
nRESET Not reset	IC	Used to start the processor from a known address. A LOW level causes the instruction being executed to terminate abnormally. This signal must be held LOW for at least two clock cycles, with **nWAIT** held HIGH. When LOW the processor performs internal cycles with the address incrementing from the point where reset was activated. The address overflows to zero if **nRESET** is held beyond the maximum address limit. When HIGH for at least one clock cycle, the processor restarts from address 0.
nRW Not read, write	O8	When the processor is performing a read cycle, this is LOW. This is one of the signals controlled by **APE**, **ALE**, and **ABE**.
nTDOEN Not **TDO** enable	O4	When serial data is being driven out on **TDO** this is LOW. Usually used as an output enable for a **TDO** pin in a packaged part.
nTRANS Not memory translate	O8	When the processor is in User mode, this is LOW. It can be used either to tell the memory management system when address translation is turned on, or as an indicator of non-User mode activity. This is one of the signals controlled by **APE**, **ALE**, and **ABE**.
nTRST Not test reset	IC	Reset signal for the boundary-scan logic. This pin must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset, **nRESET**. See Chapter 5 *Debug Interface*.
nWAIT Not wait	IC	When LOW the processor extends an access over a number of cycles of **MCLK**, which is useful for accessing slow memory or peripherals. Internally, **nWAIT** is logically ANDed with **MCLK** and must only change when **MCLK** is LOW. If **nWAIT** is not used it must be tied HIGH.
PCLKBS Boundary scan update clock	O4	This is used by an external boundary-scan chain as the update clock. This must be left unconnected, if an external boundary-scan chain is not connected.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
RANGEOUT0 EmbeddedICE RANGEOUT0	O4	When the EmbeddedICE watchpoint unit 0 has matched the conditions currently present on the address, data, and control busses, then this is HIGH. This signal is independent of the state of the watchpoint enable control bit. **RANGEOUT0** changes when **ECLK** is LOW.
RANGEOUT1 EmbeddedICE RANGEOUT1	O4	As **RANGEOUT0** but corresponds to the EmbeddedICE watchpoint unit 1.
RSTCLKBS Boundary scan Reset Clock	O4	When either the TAP controller state machine is in the RESET state or when **nTRST** is LOW, then this is HIGH. This can be used to reset external boundary-scan cells.
SCREG[3:0] Scan chain register	O4	These reflect the ID number of the scan chain currently selected by the TAP controller. These change on the falling edge of **TCK** when the TAP state machine is in the UPDATE-DR state.
SDINBS Boundary scan serial input data	O4	This provides the serial data for an external boundary-scan chain input. It changes from the rising edge of **TCK** and is valid at the falling edge of **TCK**.
SDOUTBS Boundary scan serial output data	IC	Accepts serial data from an external boundary-scan chain output, synchronized to the rising edge of **TCK**. This must be tied LOW, if an external boundary-scan chain is not connected.
SEQ Sequential address	O4	When the address of the next memory cycle is closely related to that of the last memory access, this is HIGH. In ARM state the new address can be for the same word or the next. In THUMB state, the same halfword or the next. It can be used, in combination with the low-order address lines, to indicate that the next cycle can use a fast memory mode (for example DRAM page mode) or to bypass the address translation system.
SHCLKBS Boundary scan shift clock, phase one	O4	Used to clock the master half of the external scan cells and follows **TCK1** when in the SHIFT-DR state of the state machine and scan chain 3 is selected. When not in the SHIFT-DR state or when scan chain 3 is not selected, this clock is LOW.
SHCLK2BS Boundary scan shift clock, phase two	O4	As **SHCLKBS** but follows **TCK2** instead of **TCK1**. This must be left unconnected, if an external boundary-scan chain is not connected.

Table A-3 Signal Descriptions (continued)

Name	Type	Description
TAPSM[3:0] TAP controller state machine	O4	These reflect the current state of the TAP controller state machine. These bits change on the rising edge of **TCK**. See Figure B-2 on page B-5.
TBE Test bus enable	IC	When LOW, **D[31:0]**, **A[31:0]**, **LOCK**, **MAS[1:0]**, **nRW**, **nTRANS**, and **nOPC** are set to high impedance. Similar in effect as if both **ABE** and **DBE** had been driven LOW. However**, TBE** does not have an associated scan cell and so allows external signals to be driven high impedance during scan testing. Under normal operating conditions **TBE** must be HIGH.
TBIT	O4	When the processor is executing the THUMB instruction set, this is HIGH. It is LOW when executing the ARM instruction set. This signal changes in phase two in the first execute cycle of a BX instruction.
TCK	IC	Clock signal for all test circuitry. When in debug state, this is used to generate **DCLK**, **TCK1**, and **TCK2**.
TCK1 **TCK**, phase one	O4	HIGH when **TCK** is HIGH (slight phase lag due to the internal clock non-overlap).
TCK2 **TCK**, phase two	O4	HIGH when **TCK** is LOW (slight phase lag due to the internal clock non-overlap). It is the non-overlapping complement of **TCK1**.
TDI	IC	Serial data for the scan chains.
TDO Test data output	O4	Serial data from the scan chains.
TMS	IC	Mode select for scan chains.
VDD Power supply	P	Provide power to the device.
VSS Ground	P	These connections are the ground reference for all signals.

Appendix B
Debug in Depth

This appendix describes the debug features of the ARM7TDMI core in further detail and includes additional information about the EmbeddedICE Logic. It contains the following sections:

B.1 Scan chains and JTAG interface

There are three JTAG-style scan chains within the ARM7TDMI core. These enable debugging and configuration of EmbeddedICE Logic.

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*.

In addition, support is provided for an optional fourth scan chain. This is intended to be used for an external boundary-scan chain around the pads of a packaged device. The control signals provided for this scan chain are described in *Scan chain 3* on page B-20.

Note

The scan cells are not fully JTAG compliant.

The following sections describe:

- *Scan chain implementation* on page B-3
- *TAP state machine* on page B-5.

B.1.1 Scan chain implementation

The three scan paths are referred to as:

1. *Scan chain 0* on page B-4
2. *Scan chain 1* on page B-4
3. *Scan chain 2* on page B-4.

The scan chains are shown in Figure B-1 on page B-4.

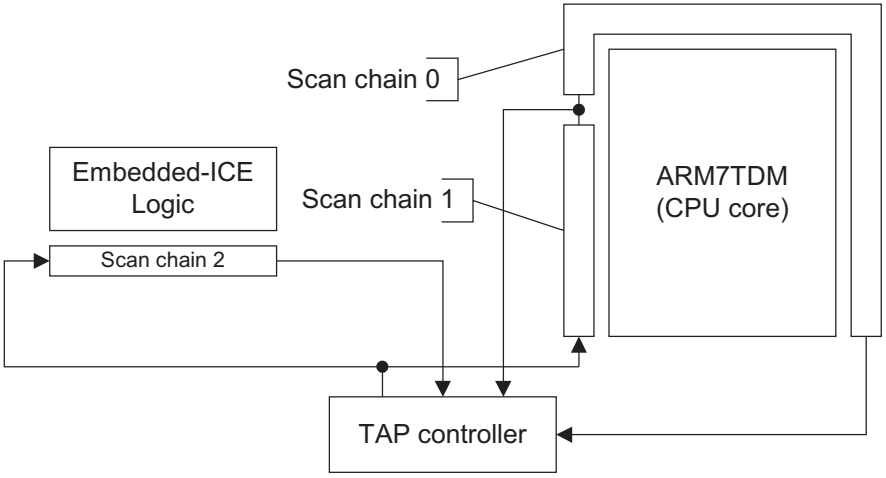

Figure B-1 ARM7TDMI core scan chain arrangements

Scan chain 0

Scan chain 0 enables access to the entire periphery of the ARM7TDMI core, including the data bus. The scan chain functions enable inter-device testing (EXTEST) and serial testing of the core (INTEST). The order of the scan chain, from search data in to out, is:

1. Data bus bits 0 to 31.
2. The control signals.
3. Address bus bits 31 to 0.

   A[0] is scanned out first.

Scan chain 1

Scan chain 1 is a subset of scan chain 0. It provides serial access to the core data bus **D[31:0]** and the **BREAKPT** signal.

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

1. Data bus bits 0 to 31.
2. The **BREAKPT** bit, the first to be shifted out.

Scan chain 2

Scan chain 2 enables access to the EmbeddedICE Logic registers. Refer to *Test data registers* on page B-14 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-5 shows the state transitions that occur in the TAP controller.

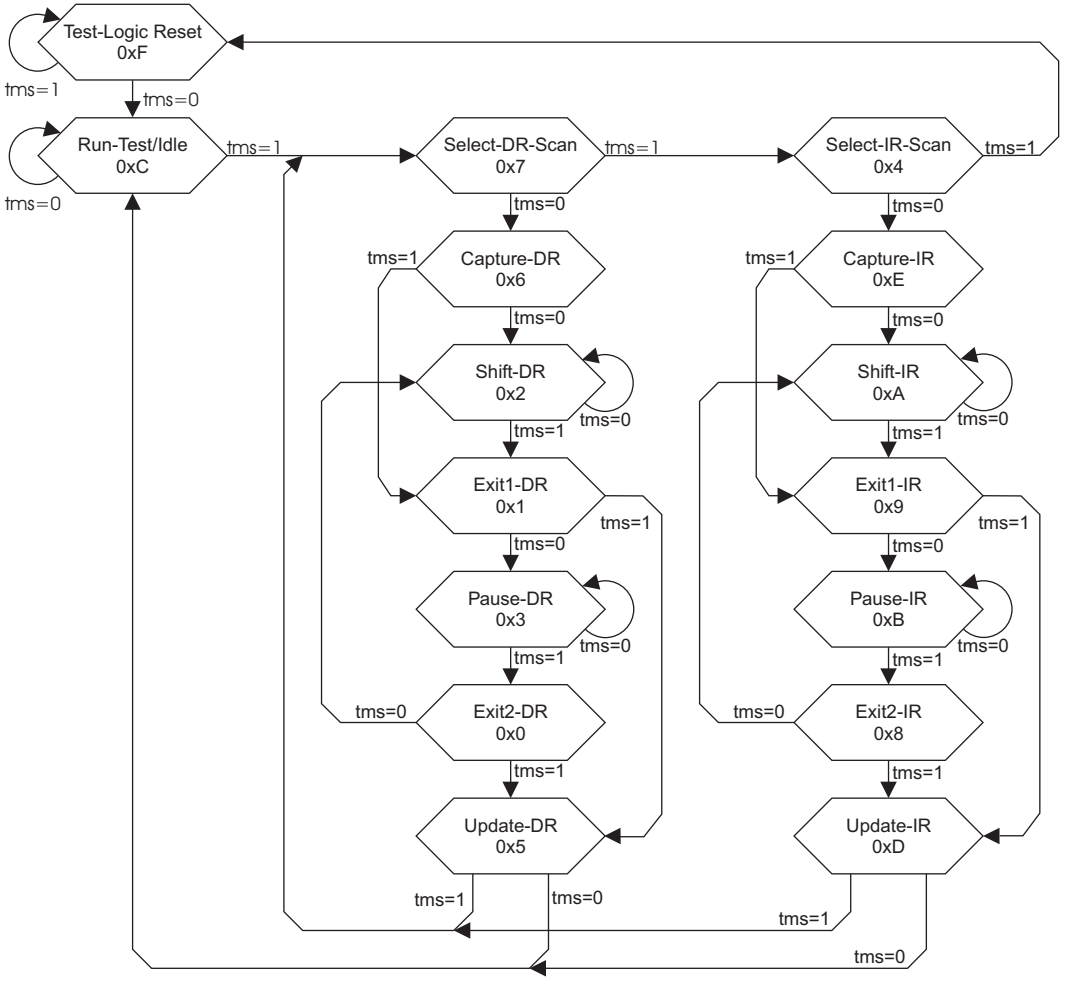

Figure B-2 Test access port controller state transitions

From IEEE Std 1149.1-1990. Copyright 1994-2001 IEEE. All rights reserved.

B.2 Resetting the TAP controller

The boundary-scan (JTAG) interface includes a state machine controller named the TAP controller. To force the TAP controller into the correct state after power-up, you must apply a reset pulse to the **nTRST** signal:

- When the boundary-scan interface or EmbeddedICE is to be used, **nTRST** must be driven LOW and then HIGH again.
- When the boundary-scan interface or EmbeddedICE is not to be used, the **nTRST** input can be tied permanently LOW.

Note

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

The **nTRST** signal:

1. Selects system mode. This means that the boundary-scan cells do not intercept any of the signals passing between the external system and the core.
2. Selects the IDCODE instruction.

   When the TAP controller is put into the SHIFT-DR state and **TCK** is pulsed, the contents of the ID register are clocked out of **TDO**.
3. Sets the TAP controller state machine to the TEST-LOGIC RESET state.
4. Sets the scan chain select register to `0x3`, which selects the external boundary-scan chain, if present.

Note

You must use **nTRST** to reset the boundary-scan interface at least once after power up. After this the TAP controller state machine can be put into the TEST-LOGIC RESET state to subsequently reset the boundary-scan interface.

B.3 Pullup resistors

The IEEE 1149.1 standard implies that **nTRST**, **TDI**, and **TMS** must have internal pullup resistors. To minimize static current draw, these resistors are not fitted to the ARM7TDMI core. Accordingly, the four inputs to the test interface, the **nTRST**, **TDI,** and **TMS** signal plus **TCK**, must all be driven to good logic levels to achieve normal circuit operation.

B.4 Instruction register

The instruction register is 4 bits in length.

There is no parity bit.

The fixed value 0001 is loaded into the instruction register during the CAPTURE-IR controller state.

The least significant bit of the instruction register is scanned in and scanned out first.

B.5 Public instructions

Table B-1 lists the public instructions.

Table B-1 Public instructions

Instruction	Binary	Hexadecimal
EXTEST	0000	0x0
SCAN_N	0010	0x2
SAMPLE/PRELOAD	0011	0x3
RESTART	0100	0x4
CLAMP	0101	0x5
HIGHZ	0111	0x7
CLAMPZ	1001	0x9
INTEST	1100	0xC
IDCODE	1110	0xE
BYPASS	1111	0xF

In the following instruction descriptions, **TDI** and **TMS** are sampled on the rising edge of **TCK** and all output transitions on **TDO** occur as a result of the falling edge of **TCK**. The following sections describe:

- *EXTEST (0000)* on page B-9
- *SCAN_N (0010)* on page B-10
- *SAMPLE/PRELOAD (0011)* on page B-10
- *RESTART (0100)* on page B-10
- *CLAMP (0101)* on page B-11
- *HIGHZ (0111)* on page B-11
- *CLAMPZ (1001)* on page B-11
- *INTEST (1100)* on page B-12
- *IDCODE (1110)* on page B-12
- *BYPASS (1111)* on page B-12.

B.5.1 EXTEST (0000)

The selected scan chain is placed in test mode by the EXTEST instruction.

The EXTEST instruction connects the selected scan chain between **TDI** and **TDO**.

When the instruction register is loaded with the EXTEST instruction, all of the scan cells are placed in their test mode of operation:

- In the CAPTURE-DR state, inputs from the system logic and outputs from the output scan cells to the system are captured by the scan cells.
- In the SHIFT-DR state, the previously captured test data is shifted out of the scan chain using **TDO**, while new test data is shifted in using the **TDI** input. This data is applied immediately to the system logic and system pins.

B.5.2 SCAN_N (0010)

The SCAN_N instruction connects the scan path select register between **TDI** and **TDO**:

- 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 **TDI** and **TDO** and remains connected until a subsequent SCAN_N instruction is issued.
- On reset, scan chain 3 is selected by default.

The scan path select register is 4 bits long in this implementation, although no finite length is specified. The least significant bit of the scan path select register is shifted in/out first.

B.5.3 SAMPLE/PRELOAD (0011)

This instruction is included for production test only and must never be used on the scan chains provided by the ARM7TDMI core. It can be used on user-added scan chains such as boundary-scan chains.

B.5.4 RESTART (0100)

The RESTART instruction restarts the processor on exit from debug state. The RESTART instruction connects the bypass register between **TDI** and **TDO**. The TAP controller behaves as if the BYPASS instruction had been loaded.

The processor exits debug state when the RUN-TEST-IDLE state is entered.

B.5.5 CLAMP (0101)

This instruction connects a 1 bit shift register, the BYPASS register, between **TDI** and **TDO**. When the CLAMP instruction is loaded into the instruction register, the state of all the scan cell output signals is defined by the values previously loaded into the currently loaded scan chain. This instruction must only be used when scan chain 0 is the currently selected scan chain:

- In the CAPTURE-DR state, a logic 0 is captured by the bypass register.
- In the SHIFT-DR state, test data is shifted into the bypass register using **TDI** and out using **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a zero.
- In the UPDATE-DR state the bypass register is not affected.

B.5.6 HIGHZ (0111)

This instruction connects a 1 bit shift register, the BYPASS register, between **TDI** and **TDO**. When the HIGHZ instruction is loaded into the instruction register, the Address bus, **A[31:0]**, the data bus, **D[31:0]**, **nRW**, **nOPC**, **LOCK**, **MAS[1:0]**, and **nTRANS** are all driven to the high impedance state and the external **HIGHZ** signal is driven HIGH. This is as if the signal **TBE** had been driven LOW:

- In the CAPTURE-DR state, a logic 0 is captured by the bypass register.
- In the SHIFT-DR state, test data is shifted into the bypass register using **TDI** and out using **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a zero.
- In the UPDATE-DR state, the bypass register is not affected.

B.5.7 CLAMPZ (1001)

This instruction connects a 1 bit shift register, the BYPASS register, between **TDI** and **TDO**.

When the CLAMPZ instruction is loaded into the instruction register, all the 3-state outputs are placed in their inactive state, but the data supplied to the scan cell outputs is derived from the scan cells. The purpose of this instruction is to ensure that, during production test, each output can be disabled when its data value is either a logic 0 or a logic 1:

- In the CAPTURE-DR state, a logic 0 is captured by the bypass register.
- In the SHIFT-DR state, test data is shifted into the bypass register using **TDI** and out using **TDO** after a delay of one **TCK** cycle. The first bit shifted out is a zero.
- In the UPDATE-DR state, the bypass register is not affected.

B.5.8 INTEST (1100)

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

- The INTEST instruction connects the selected scan chain between **TDI** and **TDO.**
- When the INTEST instruction is loaded into the instruction register, all the scan cells are placed in their test mode of operation.
- In the CAPTURE-DR state, the value of the data applied from the core logic to the output scan cells and the value of the data applied from the system logic to the input scan cells is captured.
- In the SHIFT-DR state, the previously-captured test data is shifted out of the scan chain through the **TDO** pin, while new test data is shifted in through the **TDI** pin.

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

B.5.9 IDCODE (1110)

The IDCODE instruction connects the device identification code register or ID register between **TDI** and **TDO**. The register is a 32-bit register that enables the manufacturer, part number, and version of a component to be read through the TAP. See *ARM7TDMI core device IDentification (ID) code register* on page B-14 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 **TDO** pin, while data is shifted into the ID register through the **TDI** pin.
- In the UPDATE-DR state, the ID register is unaffected.

B.5.10 BYPASS (1111)

The BYPASS instruction connects a 1-bit shift register, the bypass register, between **TDI** and **TDO**.

When the BYPASS instruction is loaded into the instruction register, all the scan cells assume their normal system mode of operation. The BYPASS instruction has no effect on the system pins:

- In the CAPTURE-DR state, a logic 0 is captured the bypass register.
- In the SHIFT-DR state, test data is shifted into the bypass register through **TDI** and shifted out through **TDO** after a delay of one **TCK** cycle. The first bit to shift out is a zero.
- In the UPDATE-DR state, the bypass register is not affected.

All unused instruction codes default to the BYPASS instruction.

Note

BYPASS does not enable the processor to exit debug state or synchronize to **MCLK** for a system speed access while in debug state. You must use RESTART to achieve this.

B.6 Test data registers

There are seven test data registers that can connect between **TDI** and **TDO**:

- *Bypass register* on page B-14
- *ARM7TDMI core device IDentification (ID) code register* on page B-14
- *Instruction register* on page B-15
- *Scan path select register* on page B-15
- *Scan chains 0, 1, 2, and 3* on page B-16.

In the following test data register descriptions, data is shifted during every **TCK** cycle.

B.6.1 Bypass register

Purpose Bypasses the device during scan testing by providing a path between **TDI** and **TDO**.

Length 1 bit.

Operating mode When the BYPASS instruction is the current instruction in the instruction register, serial data is transferred from **TDI** to **TDO** in the SHIFT-DR state with a delay of one **TCK** cycle. There is no parallel output from the bypass register.

A logic 0 is loaded from the parallel input of the bypass register in the CAPTURE-DR state.

B.6.2 ARM7TDMI core device IDentification (ID) code register

Purpose Reads the 32-bit device identification code. No programmable supplementary identification code is provided.

Length 32 bits. The format of the register is as shown in Figure B-3.

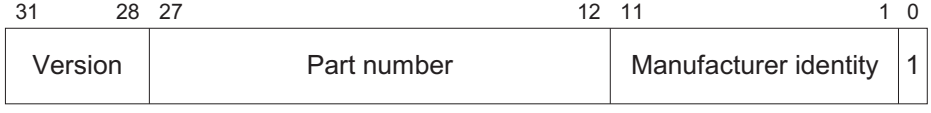

Figure B-3 ID code register format

Contact your supplier for the correct device identification code.

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.6.3 Instruction register

Purpose Changes the current TAP instruction.

Length 4 bits.

Operating mode In the SHIFT-IR state, the instruction register is selected as the serial path between **TDI** and **TDO**.

During the UPDATE-IR state, the value in the instruction register becomes the current instruction.

During the CAPTURE-IR state, the binary value 0001 is loaded into this register. This value is shifted out during SHIFT-IR. On reset, IDCODE becomes the current instruction.

The least significant bit of the register is scanned in or out first.

B.6.4 Scan path select register

Purpose Changes the current active scan chain.

Length 4 bits.

Operating mode SCAN_N as the current instruction in the SHIFT-DR state selects the scan path select register as the serial path between **TDI** and **TDO**.

During the CAPTURE-DR state, the value 1000 binary is loaded into this register. This value is loaded out during SHIFT-DR, while a new value is loaded in.

During the UPDATE-DR state, the value in the 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 0 is selected as the active scan chain.

The least significant bit of the register is scanned in or out first.

The number of the currently selected scan chain is reflected on the **SCREG[3:0]** outputs. The TAP controller can be used to drive external scan chains in addition to those within the ARM7TDMI macrocell. The external scan chain must be assigned a number and control signals for it can be derived from **SCREG[3:0]**, **IR[3:0]**,

TAPSM[3:0], **TCK1**, and **TCK2**.The list of scan chain numbers allocated by ARM are shown in Table B-2 on page B-16. An external scan chain can take any other number.The serial data stream to be applied to the external scan chain is made present on **SDINBS**, the serial data back from the scan chain must be presented to the TAP controller on the **SDOUTBS** input. The scan chain present between **SDINBS** and **SDOUTBS** is connected between **TDI** and **TDO** whenever scan chain 3 is selected, or when any of the unassigned scan chain numbers 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 **SDOUTBS**. The multiplexor can be controlled by decoding **SCREG[3:0]**.

Table B-2 lists the scan chain number allocation.

Table B-2 Scan chain number allocation

Scan chain number	Function
0	Macrocell scan test
1	Debug
2	EmbeddedICE Logic programming
3[a]	External boundary-scan
4	Reserved
8	Reserved

a. To be implemented by ASIC designer.

B.6.5 Scan chains 0, 1, 2, and 3

These enable serial access to the core logic and to EmbeddedICE Logic for programming purposes. They are described in detail below.

Scan chain 0 and 1

Purpose Enables access to the processor core for test and debug.

Length Scan chain 0: 105 bits. Scan chain 1: 33 bits

Each scan chain cell is fairly simple and consists of a serial register and a multiplexor as shown in Figure B-4 on page B-17. The scan cells perform two basic functions:

- CAPTURE

- SHIFT.

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 serial register and this is controlled by the multiplexor.

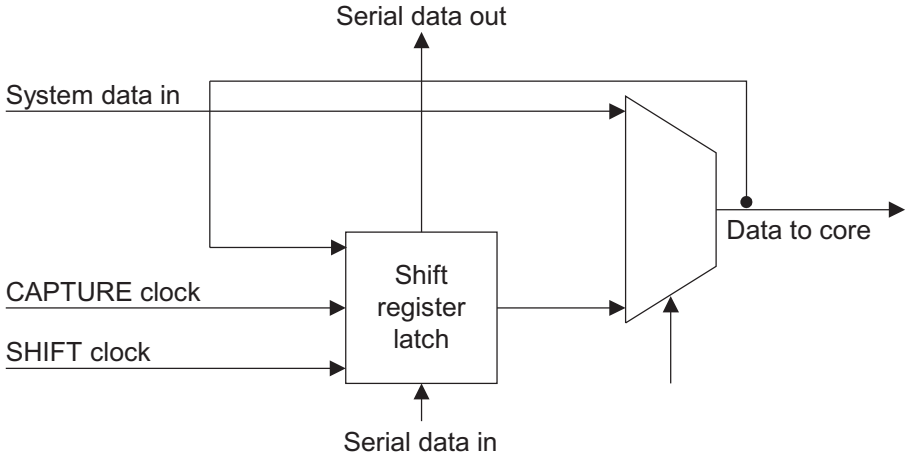

Figure B-4 Input scan cell

For output cells, capture involves placing the output value of a core 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 of the control signals for the scan cells are generated internally by the TAP controller. The action of the TAP controller is determined by the current instruction and the state of the TAP state machine.

There are three basic modes of operation of the scan chains, INTEST, EXTEST, and SYSTEM that are selected by the various TAP controller instructions:

- In INTEST mode, the core is internally tested. The data serially scanned in is applied to the core and the resulting outputs are captured in the output cells and scanned out.
- In EXTEST mode, data is scanned onto the outputs of the core and applied to the external system. System input data is captured in the input cells and then shifted out.

- In SYSTEM mode, the scan cells are idle. System data is applied to inputs and core outputs are applied to the system.

Note

The scan cells are not fully JTAG-compliant in that they do not have an *update* stage. Therefore, while data is being moved around the scan chain, the contents of the scan cell are not isolated from the output. From these operations, the output from the scan cell to the core or to the external system can change on every scan clock. This does not affect the ARM7TDMI core because its internal state does not change until it is clocked. However, the rest of the system must be aware that every output can change asynchronously as data is moved around the scan chain. External logic must ensure that this does not harm the rest of the system.

Scan chain 0

Scan chain 0 is intended primarily for inter-device testing, EXTEST, and testing the core, INTEST. Scan chain 0 is selected using the SCAN_N instruction as described at *SCAN_N (0010)* on page B-10.

INTEST enables serial testing of the core. The TAP controller must be placed in INTEST mode after scan chain 0 has been selected:

- During CAPTURE-DR, the current outputs from the core logic are captured in the output cells.
- During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, therefore applying known stimuli to the inputs.
- During RUN-TEST-IDLE, the core is clocked. Usually, the TAP controller only spends one cycle in RUN-TEST-IDLE. The whole operation can then be repeated.

For a description of the core clocks during test and debug, see *The ARM7TDMI core clocks* on page B-22.

EXTEST enables inter-device testing, useful for verifying the connections between devices on a circuit board. The TAP controller must be placed in EXTEST mode after scan chain 0 has been selected:

- During CAPTURE-DR, the current inputs to the core logic from the system are captured in the input cells.
- During SHIFT-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known values on the outputs of the core.

- During UPDATE-DR, the value shifted into the data bus **D[31:0]** scan cells appears on the outputs. For all other outputs, the value appears as the data is shifted round.

Note

During RUN-TEST-IDLE, the core is not clocked.

The operation can then be repeated.

Scan chain 1

The primary use for scan chain 1 is for debugging, although it can be used for EXTEST on the data bus. Scan chain 1 is selected using the SCAN_N TAP controller instruction. Debugging is similar to INTEST and the procedure described above for scan chain 0 must be followed.

Scan chain 1 is 33 bits long, 32 bits for the data value, plus the scan cell on the **BREAKPT** core input. This 33rd bit serves four purposes:

1. Under normal INTEST test conditions, it enables a known value to be scanned into the **BREAKPT** input.
2. During EXTEST test conditions, the value applied to the **BREAKPT** input from the system can be captured.
3. While debugging, the value placed in the 33rd bit determines if the ARM7TDMI core synchronizes back to system speed before executing the instruction. See *System speed access* on page B-31 for further details.
4. After the ARM7TDMI core has entered debug state, the first time this bit is captured and scanned out, its value tells the debugger if the core entered debug state due to a breakpoint, bit 33 clear, or a watchpoint, bit 33 set.

Scan chain 2

Purpose Enables the EmbeddedICE macrocell registers to be accessed. The order of the scan chain, from **TDI** to **TDO** is:

1. Read/write, register address bits 4 to 0.
2. Data value bits 31 to 0.

See *EmbeddedICE block diagram* on page B-41.

Length 38 bits.

To access this serial register, scan chain 2 must first be selected using the SCAN_N TAP controller instruction. The TAP controller must then be placed in INTEST mode.

- During CAPTURE-DR, no action is taken.
- During SHIFT-DR, a data value is shifted into the serial register. Bits 32 to 36 specify the address of the EmbeddedICE Logic register to be accessed.
- During UPDATE-DR, this register is either read or written depending on the value of bit 37, with 0=read).

Scan chain 3

Purpose Enables the ARM7TDMI core to control an external boundary-scan chain.

Length User defined.

Scan chain 3 control signals are provided so that an optional external boundary-scan chain can be controlled through the ARM7TDMI core. Typically, this is used for a scan chain around the pad ring of a packaged device.

The following control signals are provided which are generated only when scan chain 3 has been selected. These outputs are inactive at all other times:

DRIVEBS This is used to switch the scan cells from system mode to test mode. This signal is asserted whenever either the INTEST, EXTEST, CLAMP, or CLAMPZ instruction is selected.

PCLKBS This is an update clock, generated in the UPDATE-DR state. Typically the value scanned into a chain is transferred to the cell output on the rising edge of this signal.

ICAPCLKBS, ECAPCLKBS

These are capture clocks used to sample data into the scan cells during INTEST and EXTEST respectively. These clocks are generated in the CAPTURE-DR state.

SHCLKBS, SHCLK2BS

These are non-overlapping clocks generated in the SHIFT-DR state used to clock the master and slave element of the scan cells respectively. When the state machine is not in the SHIFT-DR state, both these clocks are LOW.

The following scan chain control signals can also be used for scan chain 3:

nHIGHZ This signal can be used to drive the outputs of the scan cells to the HIGH impedance state. This signal is driven LOW when the HIGHZ instruction is loaded into the instruction register and HIGH at all other times.

RSTCLKBS This signal is active when the TAP controller state machine is in the RESET-TEST LOGIC state. It can be used to reset any additional scan cells.

In addition to these control outputs, **SDINBS** output and **SDOUTBS** input are also provided. When an external scan chain is in use, **SDOUTBS** must be connected to the serial data output of the external scan chain and **SDINBS** must be connected to the serial data input of the scan chain.

B.7 The ARM7TDMI core clocks

The ARM7TDMI core has two clocks:

- the memory clock, **MCLK**
- an internally **TCK** generated clock, **DCLK**.

During normal operation, the core is clocked by **MCLK** and internal logic holds **DCLK** LOW. When the ARM7TDMI core is in the debug state, the core is clocked by **DCLK** under control of the TAP state machine and **MCLK** can free-run. The selected clock is output on the signal **ECLK** for use by the external system.

Note

When the CPU core is being debugged and is running from **DCLK**, **nWAIT** has no effect.

B.7.1 Clock switch during debug

When the ARM7TDMI core enters debug state, it must switch from **MCLK** to **DCLK**. This is handled automatically by logic in the ARM7TDMI core. On entry to debug state, the core asserts **DBGACK** in the HIGH phase of **MCLK**. The switch between the two clocks occurs on the next falling edge of **MCLK**. This is shown in Figure B-5.

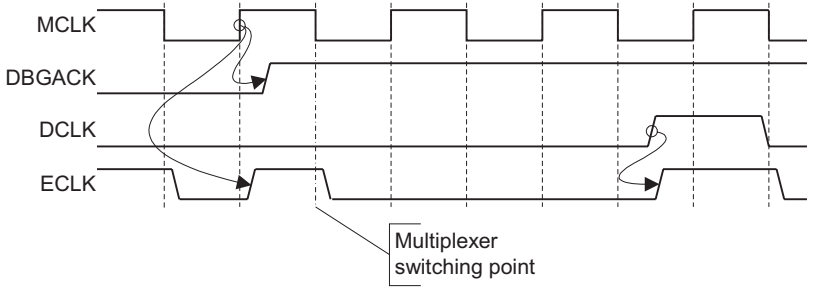

Figure B-5 Clock switching on entry to debug state

The ARM7TDMI core is forced to use **DCLK** as the primary clock until debugging is complete. On exit from debug, the core must be enabled to synchronize back to **MCLK**. This must be done in the following sequence:

1. The final instruction of the debug sequence must be shifted into the data bus scan chain and clocked in by asserting **DCLK**.

2. At this point, RESTART must be clocked into the TAP instruction register.
3. The ARM7TDMI core now automatically resynchronizes back to **MCLK** and starts fetching instructions from memory at **MCLK** speed.

Refer to *Exit from debug state* on page B-26.

B.7.2 Clock switch during test

When under serial test conditions, that is when test patterns are being applied to the ARM7TM core through the JTAG interface, the ARM7TDMI core must be clocked using **DCLK**. Entry into test is less automatic than debug and some care must be taken. On the way into test, **MCLK** must be held LOW. The TAP controller can now be used to serially test the ARM7TDMI core. If scan chain 0 and INTEST are selected, **DCLK** is generated while the state machine is in the RUN-TEST-IDLE state. During EXTEST, **DCLK** is not generated.

On exit from test, RESTART must be selected as the TAP controller instruction. When this is done, **MCLK** can be enabled to resume.

Note

After INTEST testing, care must be taken to ensure that the core is in a sensible state before switching back. The safest way to do this is to either select RESTART and then cause a system reset, or to insert `MOV PC, #0` into the instruction pipeline before switching back.

B.8 Determining the core and system state

When the ARM7TDMI core is in debug state, you examine the core and system state by forcing the load and store multiples into the instruction pipeline.

Before you can examine the core and system state, the debugger must determine if the processor entered debug from Thumb state or ARM state, by examining bit 4 of the EmbeddedICE debug status register. When bit 4 is HIGH, the core has entered debug from Thumb state, when bit 4 is LOW, the core has entered debug entered from ARM state.

B.8.1 Determining the core state

When the processor has entered debug state from Thumb state, the simplest course of action is for the debugger to force the core back into ARM state. The debugger can then execute the same sequence of instructions to determine the processor state.

To force the processor into ARM state while in debug, execute the following sequence of Thumb instructions on the core:

```
STR R0, [R0]; Save R0 before use
MOV R0, PC  ; Copy PC into R0
STR R0, [R0]; Now save the PC in R0
BX  PC      ; Jump into ARM state
MOV R8, R8  ; NOP
MOV R8, R8  ; NOP
```

Note

Because all Thumb instructions are only 16 bits long, the simplest course of action, 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.

You can use the sequences of ARM instructions in Example B-1 and Example B-2 on page B-25 to determine the state of the processor.

With the processor in the ARM state, the instruction to execute is shown in Example B-1.

Example B-1 Instruction to determine core state

```
STM R0, {R0-R15}
```

The instruction in Example B-1 on page B-24 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 current bank of registers, you might want to access the banked registers. To do this, you must change mode. Typically, a mode change can occur only if the core is already in a privileged mode. However, while in debug state, a mode change from one mode into any other mode can occur. The debugger must restore the original mode before exiting debug state.

For example, if the debugger has been requested to return the state of the User mode registers and FIQ mode registers and debug state was entered in Supervisor mode, the instruction sequence can be as listed in Example B-2.

Example B-2 Determining state of User and FIQ mode registers

```
STM R0, {R0-R15};   Save current registers
MRS R0, CPSR
STR R0, R0;         Save CPSR to determine current mode
BIC R0, 0x1F;       Clear mode bits
ORR R0, 0x10;       Select user mode
MSR CPSR, R0;       Enter USER mode
STM R0, {R13,R14};  Save register not previously visible
ORR R0, 0x01;       Select FIQ mode
MSR CPSR, R0;       Enter FIQ mode
STM R0, {R8-R14};   Save banked FIQ registers
```

All these instructions execute at debug speed. Debug speed is much slower than system speed. This is because between each core clock, 33 clocks occur in order to shift in an instruction, or shift out data. Executing instructions this slowly is acceptable for accessing the core state because the ARM7TDMI core is fully static. However, you cannot use this method for determining the state of the rest of the system.

While in debug state, only the following instructions can be scanned into the instruction pipeline for execution:

- all data processing operations
- all load, store, load multiple, and store multiple instructions
- MSR and MRS.

B.8.2 Determining system state

To meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously to it. The ARM7TDMI core must be forced to synchronize back to system speed. This is controlled by the 33rd bit of scan chain 1.

Any instruction can be placed in scan chain 1 with bit 33, the **BREAKPT** bit, clear. This instruction is then executed at debug speed. To execute an instruction at system speed, the instruction prior to it must be scanned into scan chain 1 with bit 33 set.

After the system speed instruction has been scanned into the data bus and clocked into the pipeline, the RESTART instruction must be loaded into the TAP controller. This causes the ARM7TDMI core to automatically synchronize back to **MCLK**, the system clock, execute the instruction at system speed, and then re-enter debug state and switch itself back to the internally generated **DCLK**. When the instruction has completed, **DBGACK** is HIGH and the core is switched back to **DCLK**. At this point, INTEST can be selected in the TAP controller and debugging can resume.

To determine that a system speed instruction has completed, the debugger must look at both **DBGACK** and **nMREQ**. To access memory, the ARM7TDMI core drives **nMREQ** LOW, after it has synchronized back to system speed. This transition is used by the memory controller to arbitrate if the ARM7TDMI core can have the bus in the next cycle. If the bus is not available, the core can have its clock stalled indefinitely. Therefore, the only way to tell that the memory access has completed, is to examine the state of both **nMREQ** and **DBGACK**. When both are HIGH, the access has completed. Usually, the debugger uses the EmbeddedICE macrocell to control debugging and by reading the EmbeddedICE macrocell status register, the state of **nMREQ** and **DBGACK** can be determined.

By using system speed load multiples and debug speed store multiples, the system memory state can be fed back to the debug host.

There are restrictions on which instructions can have the 33rd bit set. The only valid instructions on which to set this bit are loads, stores, load multiple, and store multiple. See also *Exit from debug state* on page B-26. When the core returns to debug state after a system speed access, bit 33 of scan chain 1 is set HIGH. This gives the debugger information about why the core entered debug state the first time this scan chain is read.

B.8.3 Exit from debug state

Leaving debug state involves restoring the internal state of the ARM7TDMI core, causing a branch to the next instruction to be executed and synchronizing back to **MCLK**. After restoring internal state, a branch instruction must be loaded into the pipeline. See *Behavior of the program counter during debug* on page B-29 for a description of how to calculate the branch.

Bit 33 of scan chain 1 is used to force the ARM7TDMI core to resynchronize back to **MCLK**. The penultimate instruction of the debug sequence is scanned in with bit 33 set HIGH. The final instruction of the debug sequence is the branch and this is scanned in with bit 33 LOW. The core is then clocked to load the branch into the pipeline. Now, 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 **MCLK** occurs in the core. The ARM7TDMI core then resumes normal operation, fetching instructions from memory. The delay, until the state machine is in the RUN-TEST-IDLE state, enables conditions to be set up in other devices in a multiprocessor system without taking immediate effect. Then, when the RUN-TEST-IDLE state is entered, all processors resume operation simultaneously.

The function of **DBGACK** is to tell the rest of the system when the core is in debug state. This can be used to inhibit peripherals such as watchdog timers that have real time characteristics. Also, **DBGACK** can be used to mask out memory accesses which are caused by the debugging process. For example, when the core 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. Therefore, on exit from debug state, the pipeline must be refilled to its previous state. Because of the debugging process, more memory accesses occur than is normally expected. Any system peripheral that is sensitive to the number of memory accesses can be inhibited by using **DBGACK**.

For example, imagine a fictitious peripheral that simply counts the number of memory cycles. This device must return the same answer after a program has been run both with and without debugging. Figure B-6 on page B-28 shows the behavior of the core on exit from the debug state.

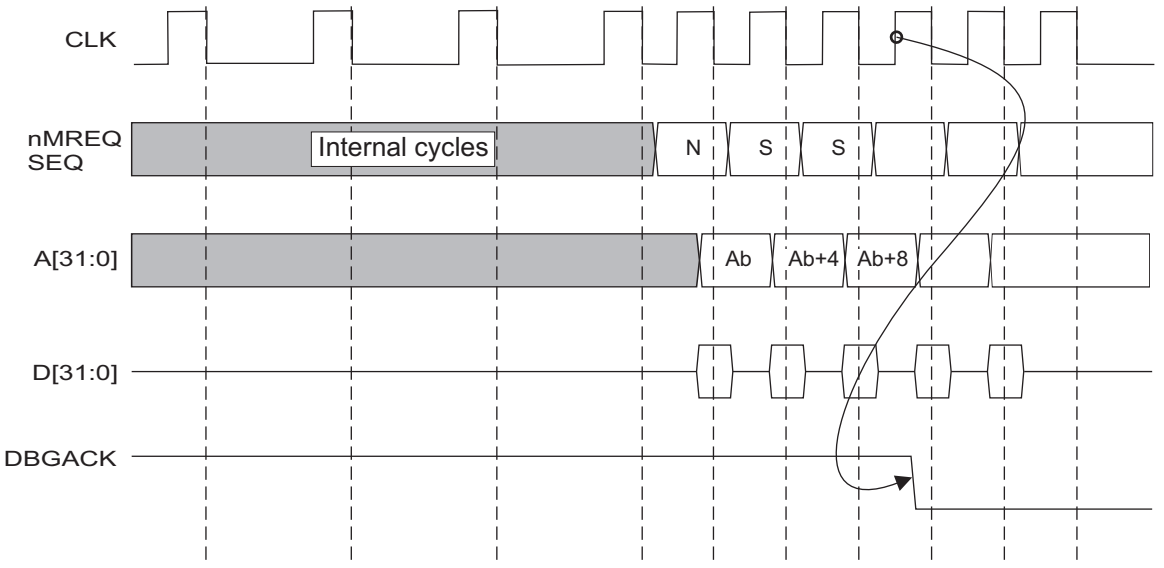

Figure B-6 Debug exit sequence

You can see from Figure 5-3 on page 5-7 that the final memory access occurs in the cycle after **DBGACK** goes HIGH, this is the point at which the cycle counter must be disabled. Figure B-6 shows that the first memory access that the cycle counter has not seen before occurs in the cycle after **DBGACK** goes LOW and so this is when the counter must be re-enabled.

Note

When a system speed access from debug state occurs, the core temporarily drops out of debug state and so **DBGACK** can go LOW. If there are peripherals that are sensitive to the number of memory accesses, they must be led to believe that the core is still in debug state. By programming the EmbeddedICE macrocell control register, the value on **DBGACK** can be forced to be HIGH.

B.9 Behavior of the program counter during debug

The debugger must keep track of what happens to the PC, so that the ARM7TDMI core can be forced to branch back to the place at which program flow was interrupted by debug. Program flow can be interrupted by any of the following:

- *Breakpoints* on page B-29
- *Watchpoints* on page B-29
- *Watchpoint with another exception* on page B-30
- *Debug request* on page B-30
- *System speed access* on page B-31.

B.9.1 Breakpoints

Entry into debug state from a breakpoint advances the PC by four addresses or 16 bytes. Each instruction executed in debug state advances the PC by one address or four bytes.

The usual way to exit from debug state after a breakpoint is to remove the breakpoint and branch back to the previously-breakpointed address.

For example, if the ARM7TDMI core entered debug state from a breakpoint set on a given address and two debug speed instructions were executed, a branch of minus seven addresses must occur:

- four for debug entry
- two for the instructions
- one for the final branch.

The following sequence shows the data scanned into scan chain 1, most significant bit first. The value of the first digit goes to the **BREAKPT** bit and then the instruction data into the remainder of scan chain 1:

```
0 E0802000; ADD R2, R0, R0
1 E1826001; ORR R6, R2, R1
0 EAFFFFF9; B-7, two's complement, seven instructions backwards
```

After the ARM7TDMI core 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.9.2 Watchpoints

The return to program execution after entry to debug state from a watchpoint is done in the same way as the procedure described in *Breakpoints* on page B-29.

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.

B.9.3 Watchpoint with another exception

If a watchpointed access simultaneously causes a Data Abort, the ARM7TDMI core enters debug state in abort mode. Entry into debug is held off until the core changes into abort mode and has fetched the instruction from the abort vector.

A similar sequence follows when an interrupt, or any other exception, occurs during a watchpointed memory access. The ARM7TDMI core 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 give the user the choice of servicing the exception before debugging.

Entry to debug state when an exception has occurred causes the PC to be incremented by three instructions rather than four and this must be considered in return branch calculation when exiting debug state. For example, suppose that an abort occurs on a watchpointed access and ten instructions have been executed to determine this eventuality. You can use the following sequence to return to program execution:

```
0 E1A00000; MOV R0, R0
1 E1A00000; MOV R0, R0
0 EAFFFFF0; B -16
```

This code forces a branch back to the abort vector, causing the instruction at that location to be refetched and executed.

Note

After the abort service routine, the instruction that caused the abort and watchpoint is refetched and executed. This triggers the watchpoint again and the ARM7TDMI core re-enters debug state.

B.9.4 Debug request

Entry into debug state through a debug request is similar to a breakpoint. However, unlike a breakpoint, the last instruction has completed execution and so must not be refetched on exit from debug state. You can assume that entry to debug state adds three addresses to the PC and every instruction executed in debug state adds one address.

For example, suppose that you have invoked a debug request and decided to return to program execution straight away. You can use the following sequence:


```
0 E1A00000; MOV R0, R0
1 E1A00000; MOV R0, R0
0 EAFFFFFA; B -6
```

This code restores the PC and restarts the program from the next instruction.

B.9.5 System speed access

When a system speed access is performed during debug state, the value of the PC increases by three addresses. System speed instructions access the memory system and so it is possible for aborts to take place. If an abort occurs during a system speed memory access, the ARM7TDMI core enters abort mode before returning to debug state.

This is similar to an aborted watchpoint, but the problem is much harder to fix because the abort was not caused by an instruction in the main program and so the PC does not point to the instruction that caused the abort. An abort handler usually looks at the PC to determine the instruction that caused the abort and also the abort address. In this case, the value of the PC is invalid, but because the debugger can determine which location was being accessed, the debugger can be written to help the abort handler fix the memory system.

B.9.6 Summary of return address calculations

The calculation of the branch return address is as follows:

- for normal breakpoint and watchpoint, the branch is:

  `- (4+N+3S)`

- for entry through debug request, **DBGRQ,** or watchpoint with exception, the branch is:

  `- (3+N+3S)`

where N is the number of debug speed instructions executed, including the final branch, and S is the number of system speed instructions executed.

B.10 Priorities and exceptions

When a breakpoint, or a debug request occurs, the normal flow of the program is interrupted. Debug can be treated 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-29. This section covers the following priorities:

- *Breakpoint with Prefetch Abort* on page B-32
- *Interrupts* on page B-32
- *Data Aborts* on page B-32.

B.10.1 Breakpoint with Prefetch Abort

When a breakpointed instruction fetch causes a Prefetch Abort, the abort is taken and the breakpoint is disregarded. Usually, Prefetch Aborts occur when, for example, an access is made to a virtual address that does not physically exist and the returned data is therefore invalid. In such a case, the normal action of the operating system is to swap in the page of memory and to return to the previously-invalid address. This time, when the instruction is fetched and providing the breakpoint is activated, it can be data-dependent, the ARM7TDMI core enters debug state.

The Prefetch Abort, therefore, takes higher priority than the breakpoint.

B.10.2 Interrupts

When the ARM7TDMI core enters debug state, interrupts are automatically disabled.

If an interrupt is pending during the instruction prior to entering debug state, the ARM7TDMI core enters debug state in the mode of the interrupt. On entry to debug state, the debugger cannot assume that the ARM7TDMI core is in the mode expected by the user program. The ARM7TDMI core must check the PC, the CPSR, and the SPSR to accurately determine the reason for the exception.

Debug, therefore, takes higher priority than the interrupt, but the ARM7TDMI core does remember that an interrupt has occurred.

B.10.3 Data Aborts

When a Data Abort occurs on a watchpointed access, the ARM7TDMI core enters debug state in abort mode. The watchpoint, therefore, has higher priority than the abort, but the ARM7TDMI core remembers that the abort happened.

B.11 Scan chain cell data

This section provides data for:

- *Scan chain 0 cells* on page B-33
- *Scan chain 1 cells* on page B-37.

B.11.1 Scan chain 0 cells

The ARM7TDMI core provides data for scan chain 0 cells as listed in Table B-3.

Table B-3 Scan chain 0 cells

Number	Signal	Type
1	**D[0]**	Input/output
2	**D[1]**	Input/output
3	**D[2]**	Input/output
4	**D[3]**	Input/output
5	**D[4]**	Input/output
6	**D[5]**	Input/output
7	**D[6]**	Input/output
8	**D[7]**	Input/output
9	**D[8]**	Input/output
10	**D[9]**	Input/output
11	**D[10]**	Input/output
12	**D[11]**	Input/output
13	**D[12]**	Input/output
14	**D[13]**	Input/output
15	**D[14]**	Input/output
16	**D[15]**	Input/output
17	**D[16]**	Input/output
18	**D[17]**	Input/output
19	**D[18]**	Input/output

Table B-3 Scan chain 0 cells (continued)

Number	Signal	Type
20	**D[19]**	Input/output
21	**D[20]**	Input/output
22	**D[21]**	Input/output
23	**D[22]**	Input/output
24	**D[23]**	Input/output
25	**D[24]**	Input/output
26	**D[25]**	Input/output
27	**D[26]**	Input/output
28	**D[27]**	Input/output
29	**D[28]**	Input/output
30	**D[29]**	Input/output
31	**D[30]**	Input/output
32	**D[31]**	Input/output
33	**BREAKPT**	Input
34	**NENIN**	Input
35	**NENOUT**	Output
36	**LOCK**	Output
37	**BIGEND**	Input
38	**DBE**	Input
39	**MAS[0]**	Output
40	**MAS[1]**	Output
41	**BL[0]**	Input
42	**BL[1]**	Input
43	**BL[2]**	Input
44	**BL[3**]	Input

Table B-3 Scan chain 0 cells (continued)

Number	Signal	Type
45	**DCTL**[a]	Output
46	**nRW**	Output
47	**DBGACK**	Output
48	**CGENDBGACK**	Output
49	**nFIQ**	Input
50	**nIRQ**	Input
51	**nRESET**	Input
52	**ISYNC**	Input
53	**DBGRQ**	Input
54	**ABORT**	Input
55	**CPA**	Input
56	**nOPC**	Output
57	**IFEN**	Input
58	**nCPI**	Output
59	**nMREQ**	Output
60	**SEQ**	Output
61	**nTRANS**	Output
62	**CPB**	Input
63	**nM[4]**	Output
64	**nM[3]**	Output
65	**nM[2]**	Output
66	**nM[1]**	Output
67	**nM[0]**	Output
68	**nEXEC**	Output
69	**ALE**	Input

Table B-3 Scan chain 0 cells (continued)

Number	Signal	Type
70	**ABE**	Input
71	**APE**	Input
72	**TBIT**	Output
73	**nWAIT**	Input
74	**A[31]**	Output
75	**A[30]**	Output
76	**A[29]**	Output
77	**A[28]**	Output
78	**A[27]**	Output
79	**A[26]**	Output
80	**A[25]**	Output
81	**A[24]**	Output
82	**A[23]**	Output
83	**A[22]**	Output
84	**A[21]**	Output
85	**A[20]**	Output
86	**A[19]**	Output
87	**A[18]**	Output
88	**A[17]**	Output
89	**A[16]**	Output
90	**A[15]**	Output
91	**A[14]**	Output
92	**A[13]**	Output
93	**A[12]**	Output
94	**A[11]**	Output

Table B-3 Scan chain 0 cells (continued)

Number	Signal	Type
95	**A[10]**	Output
96	**A[9]**	Output
97	**A[8]**	Output
98	**A[7]**	Output
99	**A[6]**	Output
100	**A[5]**	Output
101	**A[4]**	Output
102	**A[3]**	Output
103	**A[2]**	Output
104	**A[1]**	Output
105	**A[0]**	Output

a. **DCTL** is an output from the processor used to control the unidirectional data out latch, **DOUT[31:0]**. The signal is not visible from the periphery of the ARM7TDMI core. **DCTL** is not described further in this document.

B.11.2 Scan chain 1 cells

The ARM7TDMI core provides data for scan chain 1 cells as listed in Table B-4.

Table B-4 Scan chain 1 cells

Number	Signal	Type
1	**D[0]**	Input/output
2	**D[1]**	Input/output
3	**D[2]**	Input/output
4	**D[3]**	Input/output
5	**D[4]**	Input/output

Table B-4 Scan chain 1 cells (continued)

Number	Signal	Type
6	**D[5]**	Input/output
7	**D[6]**	Input/output
8	**D[7]**	Input/output
9	**D[8]**	Input/output
10	**D[9]**	Input/output
11	**D[10]**	Input/output
12	**D[11]**	Input/output
13	**D[12]**	Input/output
14	**D[13]**	Input/output
15	**D[14]**	Input/output
16	**D[15]**	Input/output
17	**D[16]**	Input/output
18	**D[17]**	Input/output
19	**D[18]**	Input/output
20	**D[19]**	Input/output
21	**D[20]**	Input/output
22	**D[21]**	Input/output
23	**D[22]**	Input/output
24	**D[23]**	Input/output
25	**D[24]**	Input/output
26	**D[25]**	Input/output
27	**D[26]**	Input/output
28	**D[27]**	Input/output
29	**D[28]**	Input/output
30	**D[29]**	Input/output

Table B-4 Scan chain 1 cells (continued)

Number	Signal	Type
31	**D[30]**	Input/output
32	**D[31]**	Input/output
33	**BREAKPT**	Input

B.12 The watchpoint registers

The two watchpoint units, known as Watchpoint 0 and Watchpoint 1, each contain three pairs of registers:

- address value and address mask
- data value and data mask
- control value and control mask.

Each register is independently programmable and has a unique address. The function and mapping of the resisters is shown in Table B-5.

Table B-5 Function and mapping of EmbeddedICE registers

Address	Width	Function
00000	3	Debug control
00001	5	Debug status
00100	6	Debug comms control register
00101	32	Debug comms data register
01000	32	Watchpoint 0 address value
01001	32	Watchpoint 0 address mask
01010	32	Watchpoint 0 data value
01011	32	Watchpoint 0 data mask
01100	9	Watchpoint 0 control value
01101	8	Watchpoint 0 control mask
10000	32	Watchpoint 1 address value
10001	32	Watchpoint 1 address mask
10010	32	Watchpoint 1 data value
10011	32	Watchpoint 1 data mask
10100	9	Watchpoint 1 control value
10101	8	Watchpoint 1 control mask

B.12.1 Programming and reading watchpoint registers

A watchpoint register is programmed by shifting data into the EmbeddedICE scan chain, scan chain 2. The scan chain is a 38-bit shift register comprising:

- a 32-bit data field
- a 5-bit address field
- a read/write bit.

This setup is shown in Figure B-7 on page B-41.

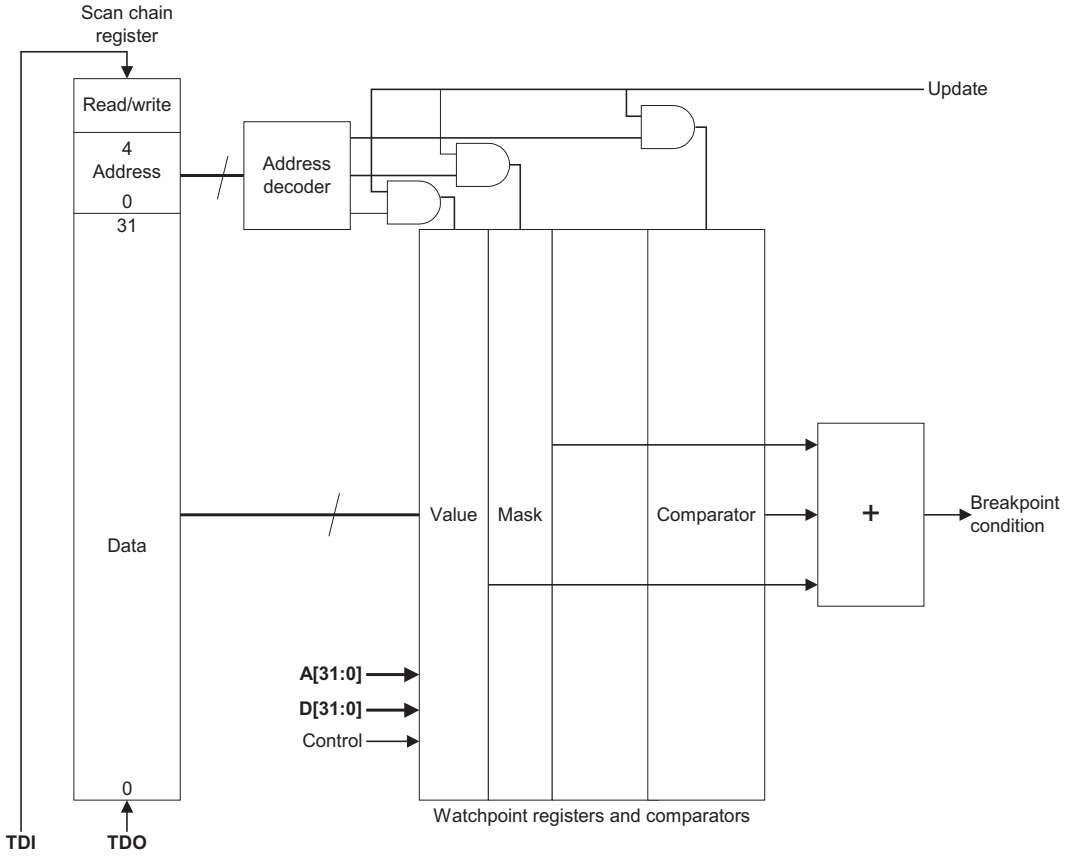

Figure B-7 EmbeddedICE block diagram

The data to be written is shifted into the 32-bit data field. The address of the register is shifted into the 5-bit address field. A 1 is shifted into the read/write bit.

A register is read by shifting its address into the address field and by shifting a 0 into the read/write bit. The 32-bit data field is ignored.

The register addresses are shown in Table B-5 on page B-40.

Note

A read or write actually takes place when the TAP controller enters the UPDATE-DR state.

B.12.2 Using the mask registers

For each value register in a register pair, there is a mask register of the same format. Setting a bit to 1 in the mask register has the effect of making the corresponding bit in the value register disregarded in the comparison.

For example, when a watchpoint is required on a particular memory location, but the data value is irrelevant, the data mask register can be programmed to `0xFFFFFFFF`, all bits set to 1, to ignore the entire data bus field.

Note

The mask is an XNOR mask rather than a conventional AND mask. When a mask bit is set to 1, the comparator for that bit position always matches, irrespective of the value register or the input value.

Setting the mask bit to 0 means that the comparator matches only if the input value matches the value programmed into the value register.

B.12.3 The control registers

The control value and control mask registers are mapped identically in the lower eight bits, as shown in Figure B-8 on page B-42.

8	7	6	5	4	3	2	1	0
ENABLE	**RANGE**	**CHAIN**	**EXTERN**	**nTRANS**	**nOPC**	**MAS[1]**	**MAS[0]**	**nRW**

Figure B-8 Watchpoint control value and mask format

Bit 8 of the control value register is the **ENABLE** bit and cannot be masked.

The bits have the following functions:

nRW Compares against the write signal from the core to detect the direction of bus activity. **nRW** is 0 for a read cycle and 1 for a write cycle.

MAS[1:0] Compares against the **MAS[1:0]** signal from the core to detect the size of bus activity.

The encoding is listed in Table B-6 on page B-43.

Table B-6 MAS[1:0] signal encoding

bit 1	bit 0	Data size
0	0	Byte
0	1	Halfword
1	0	Word
1	1	Reserved

nOPC Detects if the current cycle is an instruction fetch, with **nOPC**=0, or a data access, with **nOPC**=1.

nTRANS Compares against the not translate signal from the core to distinguish between User Mode, with **nTRANS**=0, and non-user mode, with **nTRANS**=1, accesses.

EXTERN[1:0] Is an external input to EmbeddedICE that enables the watchpoint to be dependent upon some external condition. The **EXTERN** input for Watchpoint 0 is labeled **EXTERN[0]**. The **EXTERN** input for Watchpoint 1 is labeled **EXTERN[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 ARM7TDMI core EmbeddedICE 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 Can be connected to another watchpoint unit.

In the ARM7TDMI core EmbeddedICE 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 **BREAKPT** 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.13 Programming breakpoints

Breakpoints are classified as hardware breakpoints or software breakpoints:

Hardware breakpoints on page B-45

Typically monitor the address value and can be set in any code, even in code that is in ROM or code that is self-modifying.

Software breakpoints on page B-46

Monitor a particular bit pattern being fetched from any address. One EmbeddedICE watchpoint can therefore be used to support any number of software breakpoints. Software breakpoints can normally be set only in RAM because a special bit pattern chosen to cause a software breakpoint has to replace the instruction.

B.13.1 Hardware breakpoints

To make a watchpoint unit cause hardware breakpoints on instruction fetches:

1. Program its address value register with the address of the instruction to be breakpointed.
2. For an ARM-state breakpoint, program bits [1:0] of the address mask register to 11. For a breakpoint in Thumb state, program bits [1:0] of the address mask register to 01.
3. Program the data value register only when you require a data-dependent breakpoint, that is only when you need to match the actual instruction code fetched as well as the address. If the data value is not required, program the data mask register to `0xFFFFFFFF`, all bits to 1. Otherwise program it to `0x00000000`.
4. Program the control value register with **nOPC** = 0.
5. Program the control mask register with **nOPC** = 0.
6. When you need to make the distinction between User and non-User mode instruction fetches, program the **nTRANS** value and mask bits appropriately.
7. If required, program the **EXTERN**, **RANGE,** and **CHAIN** bits in the same way.
8. Program the mask bits for all unused control values to 1.

B.13.2 Software breakpoints

To make a watchpoint unit cause software breakpoints on instruction fetches of a particular bit pattern:

1. Program its address mask register to `0xFFFFFFFF`, all bits set to 1, so that the address is disregarded.
2. Program the data value register with the particular bit pattern that has been chosen to represent a software breakpoint.

   If you are programming a Thumb software breakpoint, repeat the 16-bit pattern in both halves of the data value register. For example, if the bit pattern is `0xdeee`, program `0xDEEEDEEE`. When a 16-bit instruction is fetched, EmbeddedICE compares only the valid half of the data bus against the contents of the data value register. In this way, you can use a single watchpoint register to catch software breakpoints on both the upper and lower halves of the data bus.
3. Program the data mask register to `0x00000000`.
4. Program the control value register with **nOPC** = 0.
5. Program the control mask register with **nOPC** = 0 and all other bits to 1.
6. If you wish to make the distinction between User and non-User mode instruction fetches, program the **nTRANS** bit in the control value and control mask registers accordingly.
7. If required, program the **EXTERN**, **RANGE**, and **CHAIN** bits in the same way.

Note

You do not have to program the address value register.

Setting the breakpoint

To set the software breakpoint:

1. Read the instruction at the desired address and store it away.
2. Write the special bit pattern representing a software breakpoint at the address.

Clearing the breakpoint

To clear the software breakpoint, restore the instruction to the address.

B.14 Programming watchpoints

To make a watchpoint unit cause watchpoints on data accesses:

1. Program its address value register with the address of the data access to be watchpointed.
2. Program the address mask register to `0x00000000`.
3. Program the data value register only if you require a data-dependent watchpoint, that is, only if you need to match the actual data value read or written as well as the address. If the data value is irrelevant, program the data mask register to `0xFFFFFFFF`, all bits set to 1. Otherwise program the data mask register to `0x00000000`.
4. Program the control value register with **nOPC** = 1, **nRW** = 0 for a read, or **nRW** = 1 for a write, **MAS[1:0]** with the value corresponding to the appropriate data size.
5. Program the control mask register with **nOPC** = 0, **nRW** = 0, **MAS[1:0]** = 0 and all other bits to 1. You can set **nRW**, or **MAS[1:0]** to 1 when both reads and writes, or data size accesses are to be watchpointed respectively.
6. If you wish to make the distinction between User and non-User mode data accesses, program the **nTRANS** bit in the control value and control mask registers accordingly.
7. If required, program the **EXTERN**, **RANGE**, and **CHAIN** bits in the same way.

Note

The above are examples of how to program the watchpoint register to generate breakpoints and watchpoints. Many other ways of programming the registers are possible. For instance, you can provide simple range breakpoints by setting one or more of the address mask bits.

B.15 The debug control register

The debug control register is 3 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-9 on page B-48 shows the function of each bit in this register.

2	1	0
INTDIS	**DBGRQ**	**DBGACK**

Figure B-9 Debug control register format

If Bit 2, **INTDIS**, is asserted, the interrupt enable signal, **IFEN** of the core is forced LOW. Therefore. all interrupts, IRQ and FIQ, are disabled during debugging, **DBGACK** is HIGH, or if the **INTDIS** bit is asserted. The **IFEN** signal is driven as listed in Table B-7 on page B-48.

Table B-7 Interrupt signal control

DBGACK	INTDIS	IFEN	Interrupts
LOW	LOW	HIGH	Permitted
HIGH	x	LOW	Inhibited
x	HIGH	LOW	Inhibited

Bits 1 and 0 enable the values on **DBGRQ** and **DBGACK** to be forced.

Figure B-11 on page B-51 shows that the value stored in bit 1 of the control register is synchronized and then ORed with the external **DBGRQ** before being applied to the processor. The output of this OR gate is the signal **DBGRQI** which is brought out externally from the macrocell.

The synchronization between control bit 1 and **DBGRQI** is to assist in multiprocessor environments. The synchronization latch only opens when the TAP controller state machine is in the RUN-TEST-IDLE state. This enables an enter debug condition to be set up in all the processors in the system while they are still running. When the condition is set up in all the processors, it can then be applied to them simultaneously by entering the RUN-TEST-IDLE state.

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 ARM7TDMI core. This enables the debug system to signal to the rest of the system that the core is still being debugged even when system-speed accesses are being performed, in which case the internal **DBGACK** signal from the core is LOW.

B.16 The debug status register

The debug status register is 5 bits wide. If it is accessed for a write, with the read/write bit set, the status bits are written. If it is accessed for a read, with the read/write bit clear, the status bits are read. The format of the debug status register is shown in Figure B-10.

4	3	2	1	0
TBIT	**cgenL**	**IFEN**	**DBGRQ**	**DBGACK**

Figure B-10 Debug status register format

The function of each bit in this register is as follows:

Bit 4 Enables **TBIT** to be read. This enables the debugger to determine the processor state and therefore which instructions to execute.

Bit 3 Enables the debugger to determine if a memory access from the debug state has completed.

Bit 2 Enables the state of the core interrupt enable signal, **IFEN,** to be read. Enables the state of the **NMREQ** signal from the core, synchronized to **TCK,** to be read. This enables the debugger to determine that a memory access from the debug state has completed.

Bits 1:0 Enable the values on the synchronized versions of **DBGRQ** and **DBGACK** to be read.

The structure of the debug control and status registers is shown in Figure B-11 on page B-51.

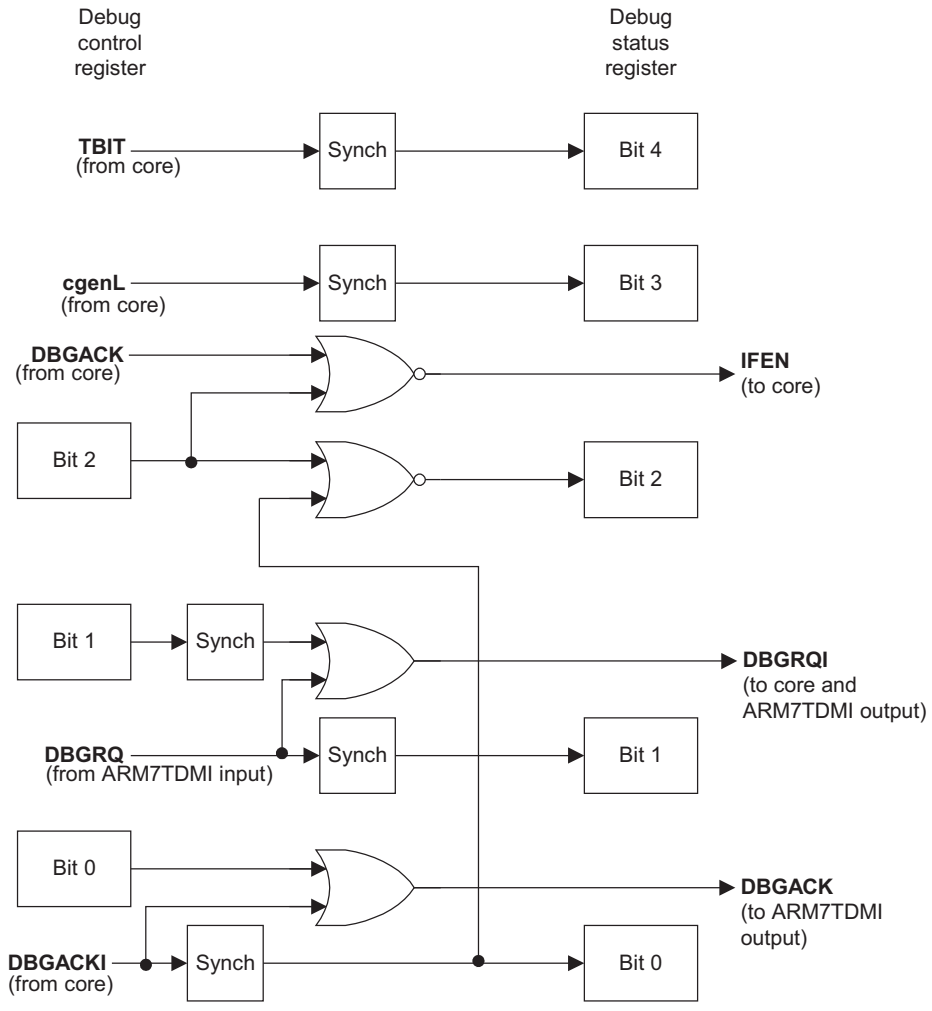

Figure B-11 Debug control and status register structure

B.17 Coupling breakpoints and watchpoints

You can couple watchpoint units 1 and 0 together using the **CHAIN** and **RANGE** inputs. Using **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.17.1 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 ARM7TDMI core
`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 ARM7TDMI core
`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 ARM7TDMI core, other watchpoint registers and the **EXTERN** 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[7:5]} XNOR {D[31:0],C[7:5]}) OR {Dm[31:0],Cm[7:5]}) == 0x7FFFFFFF)
```

The **CHAINOUT** output of watchpoint register 1 provides the **CHAIN** input to Watchpoint 0. This **CHAIN** input enables 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 multi process 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.17.2 RANGEOUT signal

The **RANGEOUT** signal is derived as follows:

```
RANGEOUT = ((({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]}) == 0x7FFFFFFF)
```

The **RANGEOUT** 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 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 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.18 EmbeddedICE timing

EmbeddedICE samples the **EXTERN[1]** and **EXTERN[0]** inputs on the falling edge of **ECLK**. Sufficient set-up and hold time must therefore be enabled for these signals.

Refer to Chapter 7 *AC and DC Parameters* for details of the required setup and hold times for these signals.

B.19 Programming Restriction

The EmbeddedICE Logic watchpoint units must only be programmed when the clock to the core is stopped. This can be achieved by putting the core into the debug state.

The reason for this restriction is that if the core continues to run at **ECLK** rates when EmbeddedICE Logic is being programmed at **TCK** rates, it is possible for the **BREAKPT** signal to be asserted asynchronously to the core.

This restriction does not apply if **MCLK** and **TCK** are driven from the same clock, or if it is known that the breakpoint or watchpoint condition can only occur some time after EmbeddedICE Logic has been programmed.

Note

This restriction does not apply in any event to the debug control or status registers.

Glossary

This glossary describes some of the terms used in this manual. Where terms can have several meanings, the meaning presented here is intended.

Abort Is caused by an illegal memory access. Abort can be caused by the external memory system, an external MMU or the EmbeddedICE Logic.

Addressing modes A procedure shared by many different instructions, for generating values used by the instructions. For four of the ARM addressing modes, the values generated are memory addresses (which is the traditional role of an addressing mode). A fifth addressing mode generates values to be used as operands by data-processing instructions.

Arithmetic Logic Unit The part of a computer that performs all arithmetic computations, such as addition and multiplication, and all comparison operations.

ALU *See* Arithmetic Logic Unit.

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

Big-endian Memory organization where the least significant byte of a word is at a higher address than the most significant byte.

Banked registers Register numbers whose physical register is defined by the current processor mode. The banked registers are registers R8 to R14.

Breakpoint A location in the image. If execution reaches this location, the debugger halts execution of the image.

See also Watchpoint.

CISC *See* Complex Instruction Set Computer.

Complex Instruction Set Computer
A microprocessor that recognizes a large number od instructions.

See also Reduced Instruction Set Computer.

CPSR *See* Program Status Register.

Control bits The bottom eight bits of a program status register. The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Current Program Status Register

See Program Status Register

Debug state A condition that allows the monitoring and control of the execution of a processor. Usually used to find errors in the application program flow.

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

EmbeddedICE The EmbeddedICE Logic is controlled via the JTAG test access port, using a protocol converter such as MultiICE: an extra piece of hardware that allows software tools to debug code running on a target processor. See also ICE and JTAG

Exception modes Privileged modes that are entered when specific exceptions occur.

Exception Handles an event. For example, an exception could handle an external interrupt or an undefined instruction.

External abort An abort that is generated by the external memory system.

FIQ Fast interrupt.

ICE *See* In-circuit emulator.

Idempotent A mathematical quantity that when applied to itself under a given binary operation equals itself.

In-circuit emulator An *In-Circuit Emulator* (ICE), is a device that aids the debugging of hardware and software. Debuggable ARM processors such as the ARM7TDMI have extra hardware called EmbeddedICE to assist this process.

See also EmbeddedICE.

IRQ Interrupt request.

Joint Test Action Group
The name of the organization that developed standard IEEE 1149.1. This standard defines a boundary-scan architecture used for in-circuit testing of integrated circuit devices.

JTAG See *Joint Test Action Group*.

Link register This register holds the address of the next instruction after a branch with link instruction.

Little-endian memory
Memory organization where the most significant byte of a word is at a higher address than the least significant byte.

LR *See* Link register

Macrocell A complex logic block with a defined interface and behavior. A typical VLSI system will comprise several macrocells (such as an ARM7TDMI, an ETM7, and a memory block) plus application-specific logic.

Memory Management Unit
Allows control of a memory system. Most of the control is provided through translation tables held in memory. The ARM7TDMI processor does not include a memory management unit, but you can add one if required.

MMU *See* Memory Management Unit

PC *See* Program Counter.

Privileged mode Any processor mode other than User mode. Memory systems typically check memory accesses from privileged modes against supervisor access permissions rather than the more restrictive user access permissions. The use of some instructions is also restricted to privileged modes.

Processor Status Register

See Program Status Register

Program Counter Register 15, the Program Counter, is used in most instructions as a pointer to the instruction that is two instructions after the current instruction.

Program Status Register
Contains some information about the current program and some information about the current processor. Also referred to as Processor Status Register.

Also referred to as *Current PSR* (CPSR), to emphasize the distinction between it and the *Saved PSR* (SPSR). The SPSR holds the value the PSR had when the current function was called, and which will be restored when control is returned.

PSR *See* Program Status Register.

Reduced Instruction Set Computer
A type of microprocessor that recognizes a lower number of instructions in comparison with a Complex Instruction Set Computer. The advantages of RISC architectures are:

- they can execute their instructions very fast because the instructions are so simple
- they require fewer transistors, this makes them cheaper to produce and more power efficient.

See also Complex Instruction Set Computer.

RISC *See* Reduced Instruction Set Computer

Saved Program Status Register
The Saved Program Status Register which is associated with the current processor mode and is undefined if there is no such Saved Program Status Register, as in User mode or System mode.

See also Program Status Register.

SBO *See* Should Be One fields.

SBZ *See* Should Be Zero fields.

Should Be One fields
Should be written as one (or all ones for bit fields) by software. Values other than one produces unpredictable results.

See also Should Be Zero fields.

Should Be Zero fields
Should be written as zero (or all 0s for bit fields) by software. Values other than zero produce unpredictable results.

See also Should Be One fields.

Software Interrupt Instruction
This instruction enters Supervisor mode to request a particular operating system function.

SPSR *See* Saved Program Status Register.

Stack pointer A register or variable pointing to the top of a stack. If the stack is full stack the SP points to the most recently pushed item, else if the stack is empty, the SP points to the first empty location, where the next item will be pushed.

Status registers *See* Program Status Register.

SP *See* Stack pointer

SWI *See* Software Interrupt Instruction.

TAP See *Test access port*.

Test Access Port The collection of four mandatory and one optional terminals that form the input/output and control interface to a JTAG boundary-scan architecture. The mandatory terminals are **TDI**, **TDO**, **TMS**, and **TCK**. The optional terminal is **nTRST**.

Thumb instruction A halfword which specifies an operation for an ARM processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

Thumb state A processor that is executing Thumb (16-bit) instructions is operating in Thumb state.

UND *See* Undefined.

Undefined Indicates an instruction that generates an undefined instruction trap.

UNP *See* Unpredictable

Unpredictable Means the result of an instruction cannot be relied upon. Unpredictable instructions must not halt or hang the processor, or any parts of the system.

Unpredictable fields Do not contain valid data, and a value can vary from moment to moment, instruction to instruction, and implementation to implementation.

Watchpoint A location in the image that is monitored. If the value stored there changes, the debugger halts execution of the image.

See also Breakpoint.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

U

W

