ARM7TDMI-S

(Rev 4)

Technical Reference Manual

ARM

ARM DDI 0234A

ARM7TDMI-S
Technical Reference Manual

Release Information

Change history

Date	Issue	Change
28 September 2001	A	First release of ARM7TDMI-S (Rev 4) processor

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-S Technical Reference Manual

List of Tables
ARM7TDMI-S Technical Reference Manual

List of Figures
ARM7TDMI-S Technical Reference Manual

Preface

This preface introduces the ARM7TDMI-S processor and its reference documentation. It contains the following sections:

About this document

This document is a reference manual for the ARM7TDMI-S processor.

Intended audience

This document has been written for experienced hardware and software engineers who might or might not have experience of ARM products.

Organization

This document is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to the ARM7TDMI-S processor.

Chapter 2 *Programmer's Model*

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

Chapter 3 *Memory Interface*

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

Chapter 4 *Coprocessor Interface*

Read this chapter for information about implementing specialized additional instructions for use with coprocessors.

Chapter 5 *Debugging Your System*

Read this chapter for a description of the ARM7TDMI-S processor hardware extensions for advanced debugging.

Chapter 6 *ETM Interface*

Read this chapter for information about connecting an ETM7 to an ARM7TDMI-S processor.

Chapter 7 *Instruction Cycle Timings*

Read this chapter for a description of the instruction cycle timings for the ARM7TDMI-S processor.

Chapter 8 *AC Parameters*

Read this chapter for the AC parameters timing diagrams and definitions.

Appendix A *Signal Descriptions*

Read this chapter for a list of all ARM7TDMI-S processor signals.

Appendix B *Differences Between the ARM7TDMI-S and the ARM7TDMI*

Read this chapter for a description of the differences between the ARM7TDMI-S processor and the ARM7TDMI hard macrocell with reference to interface signals, scan interface signals, timing parameters, and design considerations.

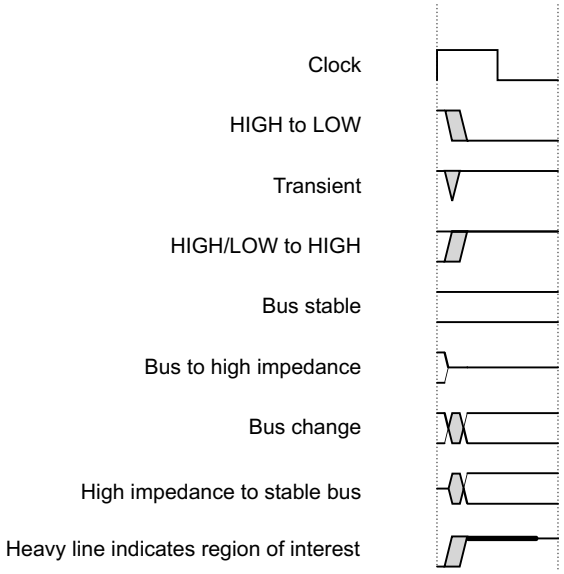
Typographical conventions

The following typographical conventions are used in this document:

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

Timing diagram conventions

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



Key to timing diagram conventions

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

Further reading

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

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

ARM publications

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

- *ARM Architecture Reference Manual* (ARM DDI 0100)
- *ARM7TDMI Technical Reference Manual* (ARM DDI 0029)
- *ETM7 (Rev 1) Technical Reference Manual* (ARM DDI 0158).

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-S processor, and on the documentation.

Feedback on this document

If you have any comments on 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.

Feedback on the ARM7TDMI-S processor

If you have any problems with the ARM7TDMI-S processor, please contact your supplier giving:

- the product name
- details of the platform you are running on, including the hardware platform, operating system type and version
- a small standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample code output illustrating the problem.

Chapter 1
Introduction

This chapter introduces the ARM7TDMI-S processor. It contains the following sections:

1.1 About the ARM7TDMI-S processor

The ARM7TDMI-S processor 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 gate count.

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.

1.1.1 The instruction pipeline

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

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

- Fetch
- Decode
- Execute.

The three-stage pipeline is shown in Figure 1-1.

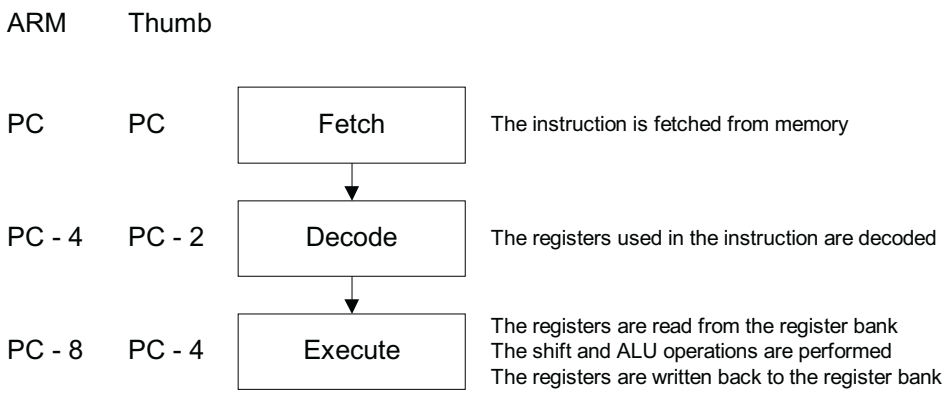


Figure 1-1 The instruction pipeline

Note

The *Program Counter* (PC) points to the instruction being fetched rather than to the instruction being executed.

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

1.1.2 Memory access

The ARM7TDMI-S processor has a Von Neumann architecture, with a single 32-bit data bus carrying both instructions and data. Only load, store, and swap instructions can access data from memory.

Data can be 8-bit bytes, 16-bit halfwords, or 32-bit words. Words must be aligned to 4-byte boundaries. Halfwords must be aligned to 2-byte boundaries.

1.1.3 Memory interface

The memory interface of the ARM7TDMI-S processor enables 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-S processor has four basic types of memory cycle:

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

1.2 ARM7TDMI-S architecture

The ARM7TDMI-S processor has two instruction sets:

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

The ARM7TDMI-S processor is an implementation of the ARM architecture v4T. For full details of both the ARM and Thumb instruction sets, see the *ARM Architecture Reference Manual*.

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, and greater than 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 the ARM code and provides 160% of the performance of ARM code when running on a processor connected to a 16-bit memory system. Thumb, therefore, makes the ARM7TDMI-S processor ideally suited to embedded applications with restricted memory bandwidth, where code density 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 and linked with Thumb code.

1.3 ARM7TDMI-S block, core and functional diagrams

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

- the ARM7TDMI-S block diagram is shown in Figure 1-2
- the ARM7TDMI-S core is shown in Figure 1-3 on page 1-7
- the ARM7TDMI-S functional diagram is shown in Figure 1-4 on page 1-8.

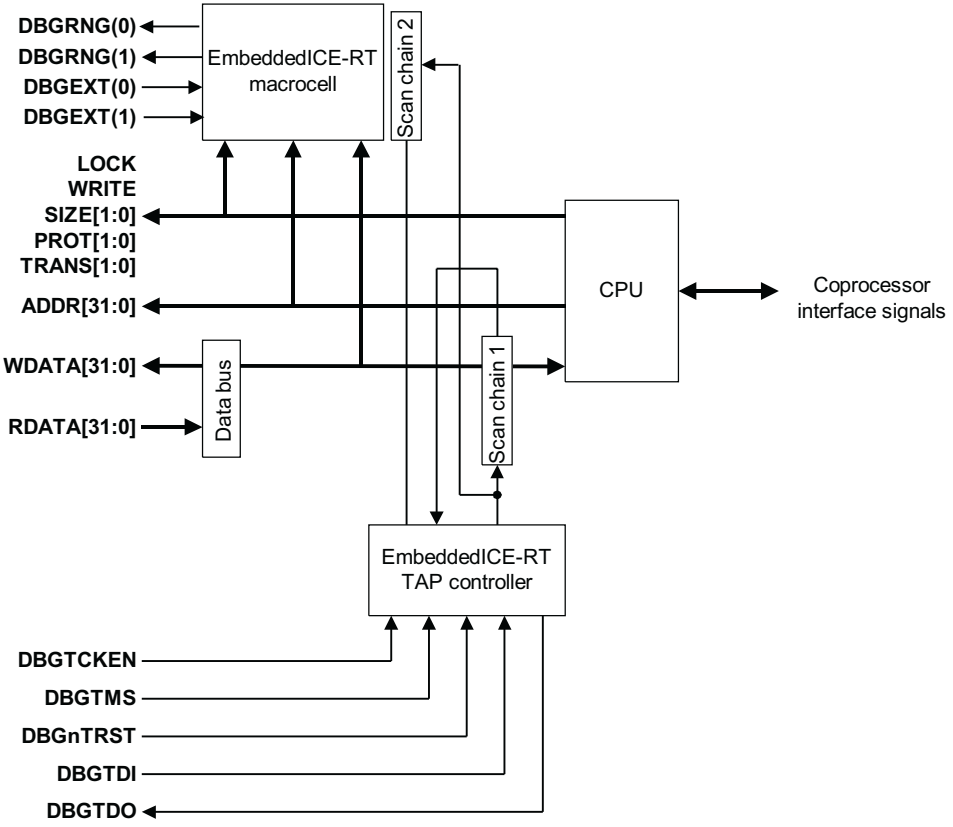


Figure 1-2 ARM7TDMI-S block diagram

Note

There are no bidirectional paths on the data bus. These are shown in Figure 1-2 for simplicity.

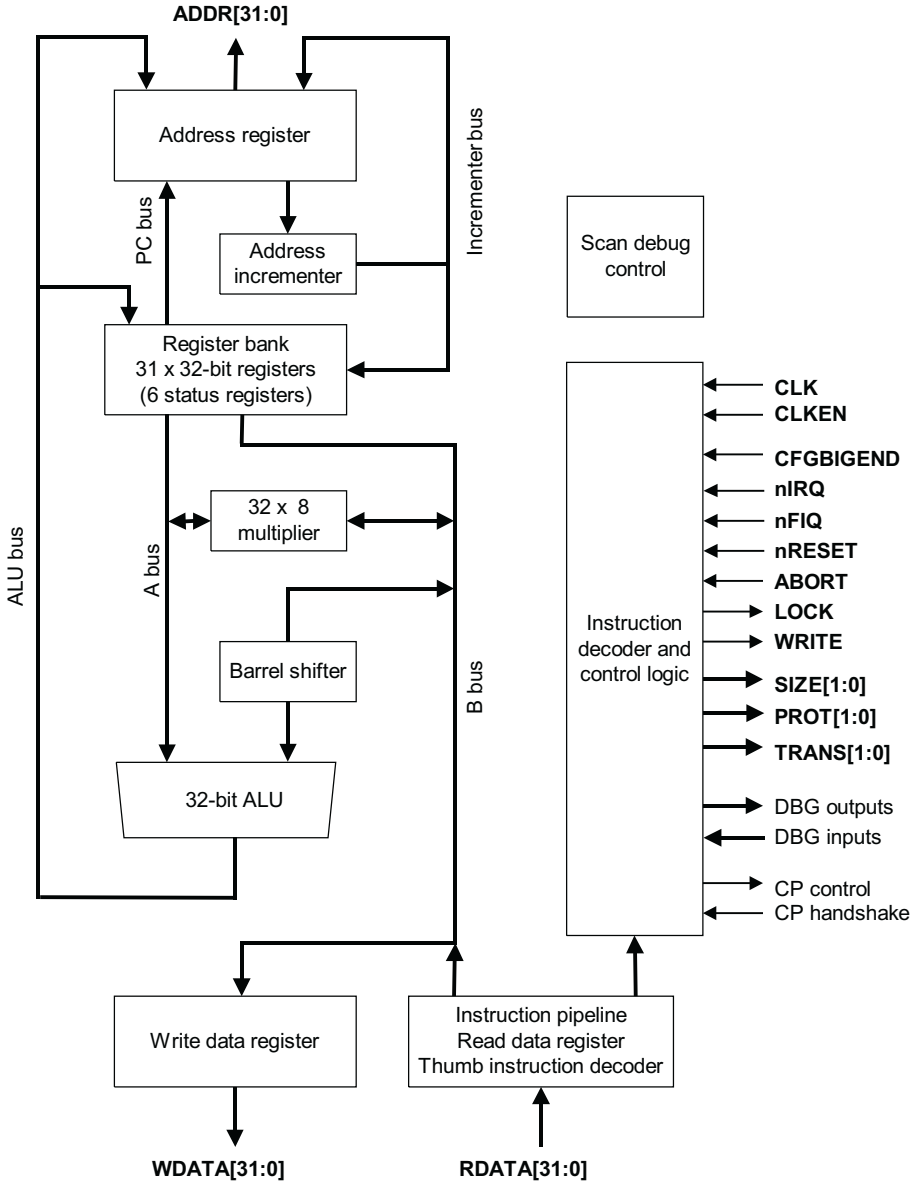


Figure 1-3 ARM7TDMI-S core

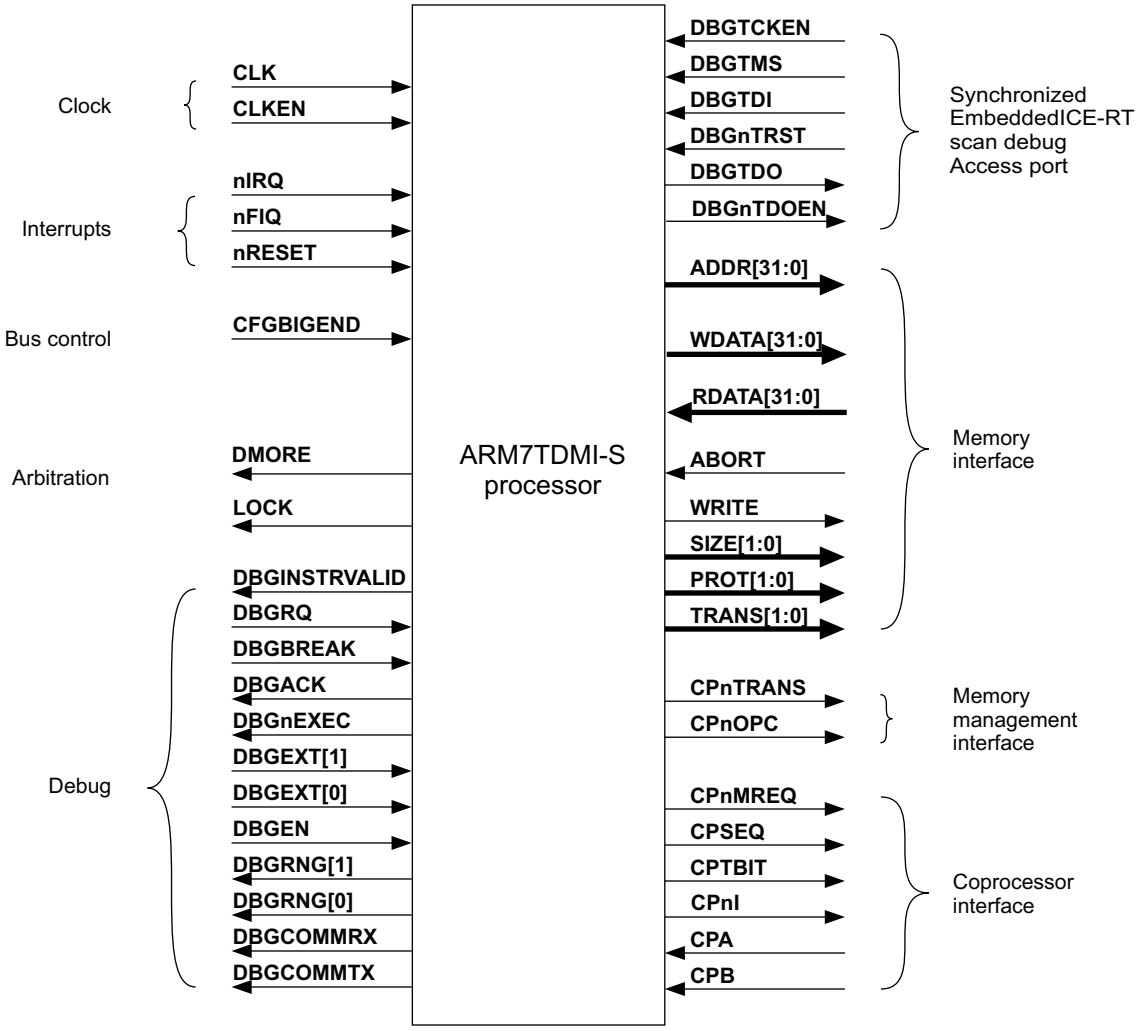


Figure 1-4 ARM7TDMI-S functional diagram

1.4 ARM7TDMI-S instruction set summary

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

- *ARM instruction summary* on page 1-10
- *Thumb instruction summary* on page 1-17.

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

The ARM7TDMI-S processor is an implementation of the ARMv4T architecture. For a complete description of both instruction sets, see the *ARM Architecture Reference Manual*.

Table 1-1 Key to tables

Instruction	Description
{cond}	See Table 1-11 on page 1-17.
<Oprnd2>	See Table 1-9 on page 1-16.
{field}	See Table 1-10 on page 1-16.
S	Sets condition codes (optional).
B	Byte operation (optional).
H	Halfword operation (optional).
T	Forces address translation. Cannot be used with pre-indexed addresses.
<a_mode2>	See Table 1-3 on page 1-13.
<a_mode2P>	See Table 1-4 on page 1-14.
<a_mode3>	See Table 1-5 on page 1-14.
<a_mode4L>	See Table 1-6 on page 1-15.
<a_mode4S>	See Table 1-7 on page 1-15.
<a_mode5>	See Table 1-8 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 }).

1.4.1 ARM instruction summary

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

Table 1-2 ARM instruction summary

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

Table 1-2 ARM instruction summary (continued)

Operation	Description	Assembler
	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>{^}
	Decrement after	LDM{cond}DA Rd{!}, <reglist>{^}
	Stack operations	LDM{cond}<a_mode4L> Rd{!}, <reglist>
	Stack operations and restore CPSR	LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^

Table 1-2 ARM instruction summary (continued)

Operation	Description	Assembler
	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 operations	STM{cond}<a_mode4S> Rd{!}, <reglist>
	User registers	STM{cond}<a_mode4S> Rd{!}, <reglist>^
Swap	Word	SWP{cond} Rd, Rm, [Rn]
	Byte	SWP{cond}B Rd, Rm, [Rn]
Coprocessors	Data operations	CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>
	Move to ARM 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 mode 2, <a_mode2>, is shown in Table 1-3.

Table 1-3 Addressing mode 2

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

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

Table 1-4 Addressing mode 2 (privileged)

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

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

Table 1-5 Addressing mode 3

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

Table 1-5 Addressing mode 3

Operation	Assembler
Register	`[Rn, +/-Rm]`
Pre-indexed	`[Rn, +/-Rm]!`
Post-indexed	`[Rn], +/-Rm`

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

Table 1-6 Addressing mode 4 (load)

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

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

Table 1-7 Addressing mode 4 (store)

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

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

Table 1-8 Addressing mode 5

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

Operand 2, <Oprnd2>, is shown in Table 1-9.

Table 1-9 Operand 2

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

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

Table 1-10 Fields

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

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

Table 1-11 Condition fields

Suffix	Description
EQ	Equal
NE	Not equal
CS	Unsigned higher, or same
CC	Unsigned lower
MI	Negative
PL	Positive, or zero
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower, or same
GE	Greater, or equal
LT	Less than
GT	Greater than
LE	Less than, or equal
AL	Always

1.4.2 Thumb instruction summary

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

Table 1-12 Thumb instruction summary

Operation		Assembler
Move	Immediate	MOV Rd, #8bit_Imm
	High to Low	MOV Rd, Hs
	Low to High	MOV Hd, Rs
	High to High	MOV Hd, Hs

Table 1-12 Thumb instruction summary (continued)

Operation		Assembler
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`

Table 1-12 Thumb instruction summary (continued)

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

Table 1-12 Thumb instruction summary (continued)

Operation			Assembler
		If Z clear and N or V set, or if Z clear, and N or V clear	`BGT label`
		If Z set, or N set and V clear, or N clear and V set	`BLE label`
		Unconditional	`B label`
		Long branch with link	`BL label`
		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>`

Table 1-12 Thumb instruction summary (continued)

Operation			Assembler
Store	With immediate offset		
		Word	`STR Rd, [Rb, #7bit_offset]`
		Halfword	`STRH Rd, [Rb, #6bit_offset]`
		Byte	`STRB Rd, [Rb, #5bit_offset]`
	With register offset		
		Word	`STR Rd, [Rb, Ro]`
		Halfword	`STRH Rd, [Rb, Ro]`
		Byte	`STRB Rd, [Rb, Ro]`
	SP-relative		`STR Rd, [SP, #10bit_offset]`
	Multiple		`STMIA Rb!, <reglist>`
Push/Pop		Push registers onto stack	`PUSH <reglist>`
		Push LR and registers onto stack	`PUSH <reglist, LR>`
		Pop registers from stack	`POP <reglist>`
		Pop registers and PC from stack	`POP <reglist, PC>`
Software Interrupt			`SWI 8bit_Imm`

1.5 Differences between Rev 3a and Rev 4

The changes incorporated in the ARM7TDMI-S (Rev 4) processor are summarized in the following sections:

- *Addition of EmbeddedICE-RT logic*
- *Improved Debug Communications Channel (DCC) bandwidth* on page 1-23
- *Access to DCC through JTAG* on page 1-23
- *TAP controller ID register* on page 1-23
- *More efficient multiple transfers* on page 1-24.

1.5.1 Addition of EmbeddedICE-RT logic

EmbeddedICE-RT is an enhanced implementation of the EmbeddedICE logic that was part of the ARM7TDMI-S (Rev 3) processor. EmbeddedICE-RT enables you to perform debugging in *monitor mode*. In monitor mode, the core takes an exception upon a breakpoint or watchpoint, rather than entering debug state as it does in *halt mode*.

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

For more details, see Chapter 5 *Debugging Your System*.

Power saving

When **DBGEN** is tied LOW, much of the EmbeddedICE-RT logic is disabled to keep power consumption to a minimum.

Changes to the programmer's model

The changes to the programmer's model are as follows:

Debug control register

Two new bits have been added:

Bit 4 Monitor mode enable. Use this to control how the device reacts on a breakpoint or watchpoint:

- When set, the core takes the instruction or data abort exception.
- When clear, the core enters debug state.

Bit 5 EmbeddedICE-RT disable. Use this when changing watchpoints and breakpoints:

- When set, this bit disables breakpoints and watchpoints, enabling the breakpoint or watchpoint registers to be programmed with new values.
- When clear, the new breakpoint or watchpoint values become operational.

For more information, see *Debug control register* on page 5-57.

Coprocessor register map

A new register (R2) in the coprocessor register map indicates whether the processor entered the Prefetch or Data Abort exception because of a real abort, or because of a breakpoint or watchpoint. For more details, see *Abort status register* on page 5-56.

1.5.2 Improved Debug Communications Channel (DCC) bandwidth

In the ARM7TDMI-S (Rev 3) processor, two accesses to scan chain 2 were required to read the DCC data. The first accessed the status bit, and the second accessed the data itself.

To increase DCC bandwidth, only one access is required to read both the data and the status bit in the ARM7TDMI-S (Rev 4) processor. The status bit is now included in the least significant bit of the address field that is read from the scan chain.

The status bit in the DCC control register is left unchanged to ensure backwards compatibility.

For more information, see *The debug communications channel* on page 5-20.

1.5.3 Access to DCC through JTAG

The DCC control register can be controlled from the JTAG interface in ARM7TDMI-S Rev 4. A processor write clears bit 0, the data read control bit.

For more information, see *The debug communications channel* on page 5-20.

1.5.4 TAP controller ID register

The TAP controller ID register value is now `0x7F1F0F0F`.

For more information, see *ARM7TDMI-S device identification (ID) code register* on page 5-31.

1.5.5 More efficient multiple transfers

The ARM7TDMI-S (Rev 4) core provides an extra output signal, **DMORE**. This signal improves the efficiency of `LDM` and `STM` instructions. **DMORE** is HIGH when the next data memory access is followed by a sequential data memory access.

For a full list of ARM7TDMI-S (Rev 4) signals, see Appendix A *Signal Descriptions.*

Chapter 2
Programmer's Model

This chapter describes the programmer's model for the ARM7TDMI-S processor. It contains the following sections:

2.1 About the programmer's model

The ARM7TDMI-S 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 fully in the *ARM Architecture Reference Manual*.

2.2 Processor operating states

The ARM7TDMI-S processor has two operating states:

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

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

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

You can switch the operating state of the ARM7TDMI-S core between ARM state and Thumb state using the `BX` instruction. This is described fully in the *ARM Architecture Reference Manual*.

All exception handling is performed 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.

2.3 Memory formats

The ARM7TDMI-S processor views memory as a linear collection of bytes numbered in ascending order from zero:

- bytes 0 to 3 hold the first stored word
- bytes 4 to 7 hold the second stored word
- bytes 8 to 11 hold the third stored word.

The ARM7TDMI-S processor can treat words in memory as being stored in one of:

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

2.3.1 Big-endian format

In big-endian format, the ARM7TDMI-S 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 byte 0 of the memory system connects to data lines 31 to 24. This is shown in Figure 2-1.

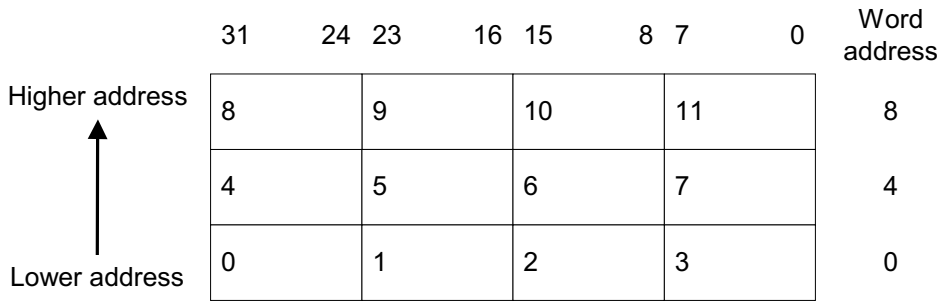


Figure 2-1 Big-endian addresses of bytes within words

2.3.2 Little-endian format

In little-endian format, the lowest-numbered byte in a word is considered the least-significant byte of the word, and the highest-numbered byte is the most significant. So byte 0 of the memory system connects to data lines 7 to 0. This is shown in Figure 2-2 on page 2-5.

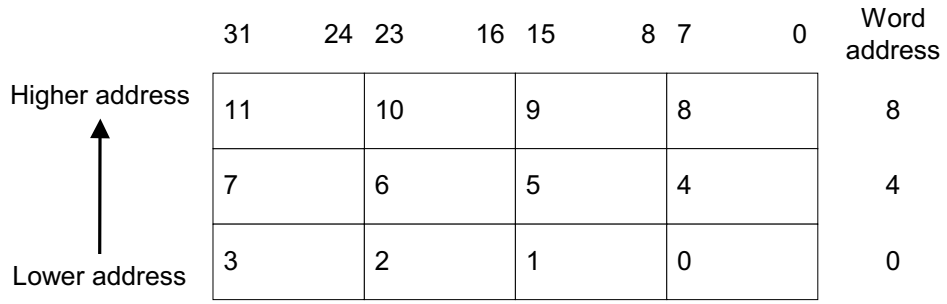


Figure 2-2 Little-endian addresses of bytes within words

2.4 Instruction length

Instructions are either:

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

2.5 Data types

The ARM7TDMI-S processor supports the following data types:

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

You must align these as follows:

- word quantities must be aligned to four-byte boundaries
- halfword quantities must be aligned to two-byte boundaries
- byte quantities can be placed on any byte boundary.

2.6 Operating modes

The ARM7TDMI-S processor has seven operating modes:

- 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.
- 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, exceptions, or access protected resources.

2.7 Registers

The ARM7TDMI-S 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.

2.7.1 The ARM state register set

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

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

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

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 *Program Counter* (PC).

In ARM state, bits [1:0] of r15 are zero. Bits [31:2] contain the PC. In Thumb state, bit [0] is zero. Bits [31:1] contain the PC.

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

See *The program status registers* on page 2-16 for a description of the program status registers.

Banked registers have a mode identifier that shows to which User mode register they are mapped. These mode identifiers are shown 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

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

In ARM state, most of the 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 stack pointer and LR for each mode

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

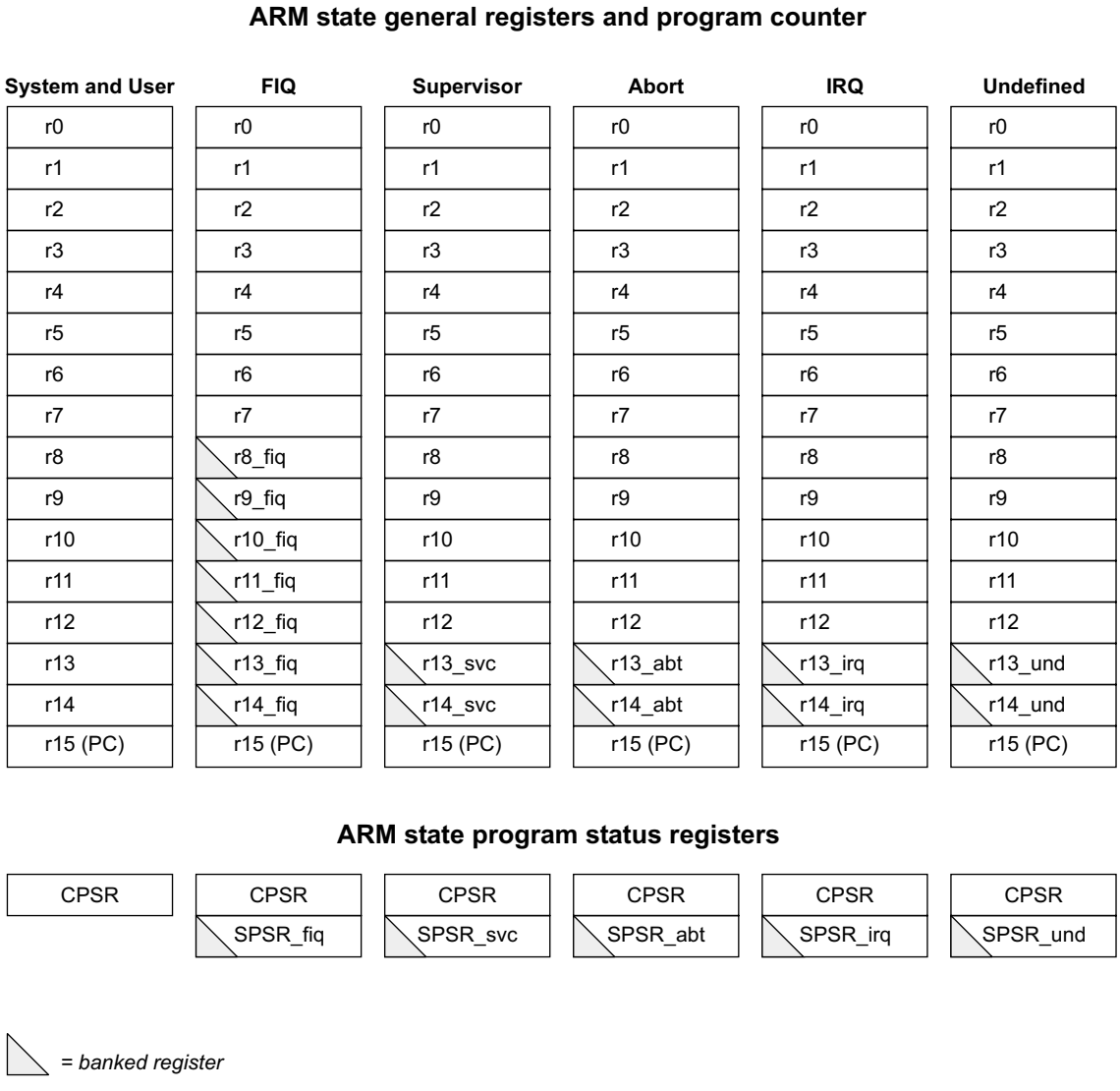


Figure 2-3 Register organization in ARM state

2.7.2 The Thumb state register set

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

- eight general registers, r0–r7
- the PC
- a Stack Pointer (SP)
- a Link Register (LR)
- the CPSR.

There are banked SPs, LRs, and SPSRs for each privileged mode. This register set is shown in Figure 2-4 on page 2-13.

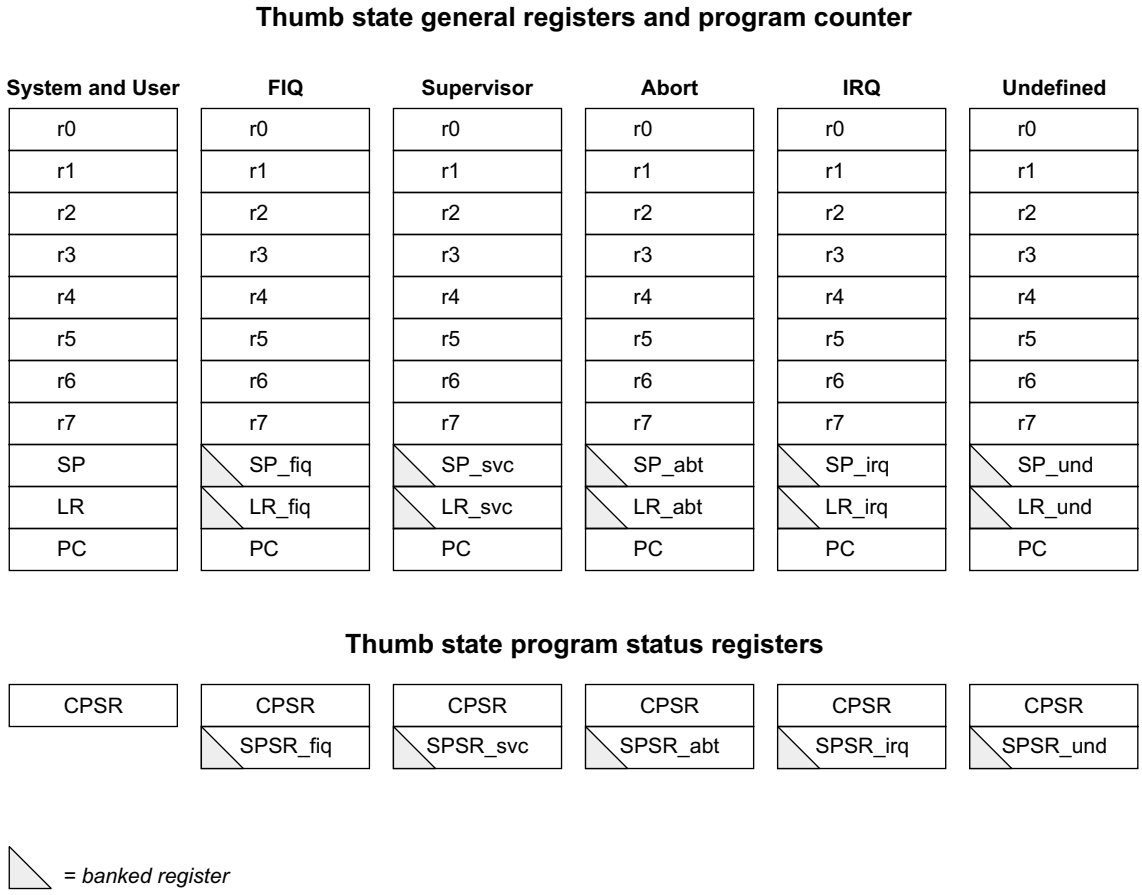


Figure 2-4 Register organization in Thumb state

2.7.3 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 ARM state r13
- Thumb state LR maps onto ARM state r14
- The Thumb state PC maps onto the ARM state PC (r15).

These relationships are shown in Figure 2-5 on page 2-14.

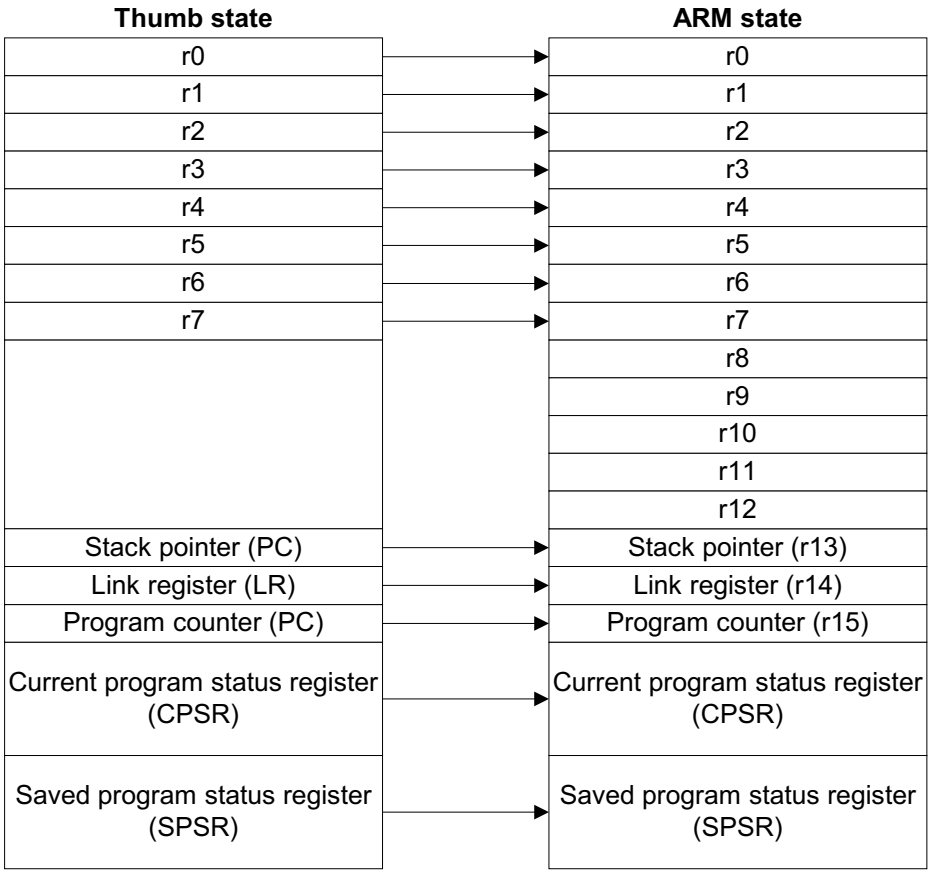


Figure 2-5 Mapping of Thumb state registers onto ARM state registers

Note

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

2.7.4 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 enables you to compare high register values with low register values. The ADD instruction enables you to add high register values to low register values. For more details, see the *ARM Architecture Reference Manual.*

2.8 The program status registers

The ARM7TDMI-S core contains a CPSR and five SPSRs for exception handlers to use. The program status registers:

- hold the condition code flags
- control the enabling and disabling of interrupts
- set the processor operating mode.

The arrangement of bits is shown in Figure 2-6.

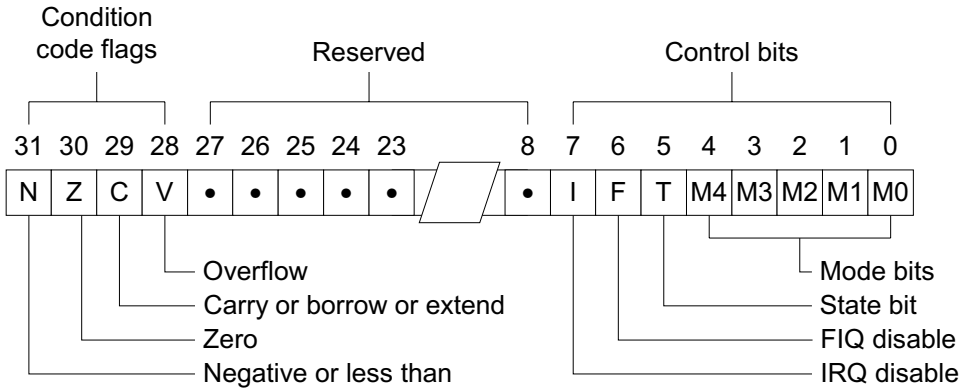


Figure 2-6 Program status register format

Note

To maintain compatibility with future ARM processors, and as good practice, you are strongly advised to use a read-write-modify strategy when changing the CPSR.

2.8.1 The condition code flags

The N, Z, C, and V bits are the condition code flags, You can set these bits by arithmetic and logical operations. The flags can also be set by MSR and LDM instructions. The ARM7TDMI-S 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, see the *ARM Architecture Reference Manual*.

2.8.2 The control bits

The bottom eight bits of a PSR are known collectively as the *control bits*. They are the:

- *Interrupt disable bits*
- *T bit*
- *Mode bits*.

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 by the **CPTBIT** external signal.

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

The M4, M3, M2, M1, and M0 bits (M[4:0]) are the mode bits. These bits 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

Table 2-2 PSR mode bit values (continued)

M[4:0]	Mode	Visible Thumb state registers	Visible ARM state registers
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

Note

If you program an illegal value into M[4:0], the processor enters an unrecoverable state.

2.8.3 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 one or zero.

2.9 Exceptions

Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example to service an interrupt from a peripheral. Before attempting to handle an exception, the ARM7TDMI-S core preserves the current processor state so that the original program can resume when the handler routine has finished.

If two or more exceptions arise simultaneously, the exceptions are dealt with in the fixed order given in *Exception priorities* on page 2-24.

This section provides details of the exception handling on the ARM7TDMI-S processor:

- *Exception entry/exit summary*
- *Entering an exception* on page 2-20
- *Leaving an exception* on page 2-21.

2.9.1 Exception entry/exit summary

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

Table 2-3 Exception entry and exit

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

Table 2-3 Exception entry and exit (continued)

Exception or entry	Return instruction	Previous state		Notes
		ARM r14_x	Thumb r14_x	
FIQ	`SUBS PC, R14_fiq, #4`	PC + 4	PC + 4	Where the PC is the address of the instruction that was not executed because the FIQ or IRQ took priority.
IRQ	`SUBS PC, R14_irq, #4`	PC + 4	PC + 4	
Data Abort	`SUBS PC, R14_abt, #8`	PC + 8	PC + 8	Where the PC is the address of the Load or Store instruction that generated the Data Abort.
RESET	Not applicable	-	-	The value saved in r14_svc on reset is `UNPREDICTABLE`.

2.9.2 Entering an exception

When handling an exception the ARM7TDMI-S core:

1. Preserves the address of the next instruction in the appropriate LR. When the exception entry is from:
   - ARM state, the ARM7TDMI-S copies the address of the next instruction into the LR (current PC + 4, or PC + 8 depending on the exception)
   - Thumb state, the ARM7TDMI-S writes the value of the PC into the LR, offset by a value (current PC + 4, or PC + 8 depending on the exception).

   The exception handler does not have to determine the state when entering an exception. For example, in the case of a SWI, `MOVS PC, r14_svc` always returns to the next instruction regardless of whether the SWI was executed in ARM or Thumb state.
2. Copies the CPSR into the appropriate SPSR.
3. Forces the CPSR mode bits to a value which depends on the exception.
4. Forces the PC to fetch the next instruction from the relevant exception vector.

The ARM7TDMI-S core also sets the interrupt disable flags on interrupt exceptions to prevent otherwise unmanageable nestings of exceptions.

Note

Exceptions are always handled 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.

2.9.3 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 2-19.
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 restores the T, F, and I bits to whatever value they held immediately prior to the exception.

2.9.4 Fast interrupt request

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

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

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

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

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

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

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

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

2.9.6 Abort

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

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-S core 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 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`, `STR`) write back modified base registers. The abort handler must be aware of this.
- The swap instruction (`SWP`) aborts as though it had not been executed. (The abort must occur on the read access of the SWP instruction.)
- Block data transfer instructions (`LDM`, `STM`) complete. When write-back is set, the base is updated. If the instruction would have overwritten the base with data (when it has the base register in the transfer list), the ARM7TDMI-S prevents the overwriting.

The ARM7TDMI-S core prevents all register overwriting after an abort is indicated. This means that the ARM7TDMI-S core always preserves r15 (always the last register to be transferred) in an aborted `LDM` instruction.

The abort mechanism enables 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, make the requested data available, and retry the aborted instruction. The application program does not have to know the amount of memory available to it, nor is its state in any way affected by the abort.

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.

2.9.7 Software interrupt instruction

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

2.9.8 Undefined instruction

When the ARM7TDMI-S processor encounters an instruction that neither it nor any coprocessor in the system can handle, the ARM7TDMI-S core takes the undefined instruction trap. Software can use this mechanism to extend the ARM instruction set by emulating undefined coprocessor instructions.

Note

The ARM7TDMI-S processor is fully compliant with the ARM architecture v4T, and traps all instruction bit patterns that are classified as undefined.

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*.

2.9.9 Exception vectors

Table 2-4 shows the exception vector addresses. In the table, I and F represent the previous value.

Table 2-4 Exception vectors

Address	Exception	Mode on entry	I state on entry	F state on entry
0x00000000	Reset	Supervisor	Disabled	Disabled
0x00000004	Undefined instruction	Undefined	I	F
0x00000008	Software interrupt	Supervisor	Disabled	F
0x0000000C	Abort (Prefetch)	Abort	I	F
0x00000010	Abort (Data)	Abort	I	F
0x00000014	Reserved	Reserved	-	-
0x00000018	IRQ	IRQ	Disabled	F
0x0000001C	FIQ	FIQ	Disabled	Disabled

2.9.10 Exception priorities

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

1. Reset (highest priority).
2. Data Abort.
3. FIQ.
4. IRQ.
5. Prefetch Abort.
6. Undefined instruction.
7. SWI (lowest priority).

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-S core 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.

2.10 Interrupt latencies

Interrupt latencies are described in:

- *Maximum interrupt latencies*
- *Minimum interrupt latencies*.

2.10.1 Maximum interrupt latencies

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

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

The total latency is therefore 27 processor cycles, slightly less than 0.7 microseconds in a system that uses a continuous 40MHz processor clock. At the end of this time, the ARM7TDMI-S executes the instruction at `0x1c`.

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

2.10.2 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} (four processor cycles).

2.11 Reset

When the **nRESET** signal goes LOW, the ARM7TDMI-S processor abandons the executing instruction.

When **nRESET** goes HIGH again the ARM7TDMI-S processor:

1. Forces M[4:0] to b10011 (Supervisor mode).
2. Sets the I and F bits in the CPSR.
3. Clears the CPSR T bit.
4. Forces the PC to fetch the next instruction from address `0x00`.
5. Reverts to ARM state and resumes execution.

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

Chapter 3
Memory Interface

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

3.1 About the memory interface

The ARM7TDMI-S processor has a Von Neumann architecture, with a single 32-bit data bus carrying both instructions and data. Only load, store, and swap instructions can access data from memory.

The ARM7TDMI-S processor supports four basic types of memory cycle:

- nonsequential
- sequential
- internal
- coprocessor register transfer.

3.2 Bus interface signals

The signals in the ARM7TDMI-S processor bus interface can be grouped into four categories:

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

The clocking and clock control signals are:

- **CLK**
- **CLKEN**
- **nRESET**.

The address class signals are:

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

The memory request signals are:

- **TRANS[1:0]**.

The data timed signals are:

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

Each of these signal groups shares a common timing relationship to the bus interface cycle. All signals in the ARM7TDMI-S processor bus interface are generated from or sampled by the rising edge of **CLK**.

Bus cycles can be extended using the **CLKEN** signal. This signal is introduced in *Using CLKEN to control bus cycles* on page 3-17. All other sections of this chapter describe a simple system in which **CLKEN** is permanently HIGH.

3.3 Bus cycle types

The ARM7TDMI-S processor bus interface is pipelined, and so the address class signals, and the memory request signals are broadcast in the bus cycle ahead of the bus cycle to which they refer. This gives the maximum time for a memory cycle to decode the address, and respond to the access request.

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

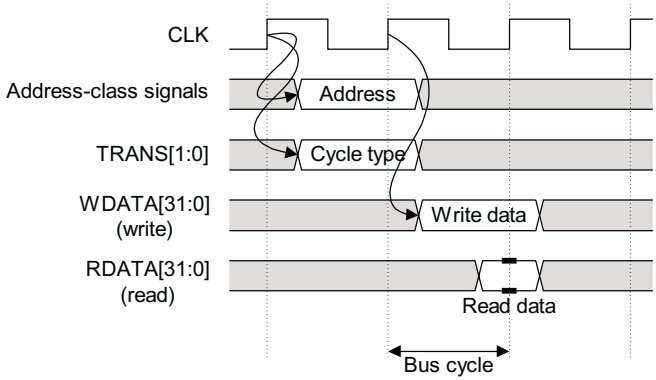


Figure 3-1 Simple memory cycle

The ARM7TDMI-S processor bus interface can perform four different types of memory cycle. These are indicated by the state of the **TRANS[1:0]** signals. Memory cycle types are encoded on the **TRANS[1:0]** signals as shown in Table 3-1.

Table 3-1 Cycle types

TRANS[1:0]	Cycle type	Description
00	I cycle	Internal cycle
01	C cycle	Coprocessor register transfer cycle
10	N cycle	Nonsequential cycle
11	S cycle	Sequential cycle

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

The ARM7TDMI-S processor has four basic types of memory cycle:

Nonsequential cycle

During this cycle, the ARM7TDMI-S core requests a transfer to, or from an address which is unrelated to the address used in the preceding cycle.

Sequential cycle During this cycle, the ARM7TDMI-S core requests a transfer to or from an address that is either one word or one halfword greater than the address used in the preceding cycle.

Internal cycle During this cycle, the ARM7TDMI-S core does not require a transfer because it is performing an internal function and no useful prefetching can be performed at the same time.

Coprocessor register transfer cycle

During this cycle, the ARM7TDMI-S core uses the data bus to communicate with a coprocessor but does not require any action by the memory system.

3.3.1 Nonsequential cycles

A nonsequential cycle is the simplest form of an ARM7TDMI-S processor bus cycle, and occurs when the ARM7TDMI-S processor requests a transfer to or from an address that is unrelated to the address used in the preceding cycle. The memory controller must initiate a memory access to satisfy this request.

The address class signals, and the **TRANS[1:0]** = 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. This is illustrated in Figure 3-2 on page 3-6.

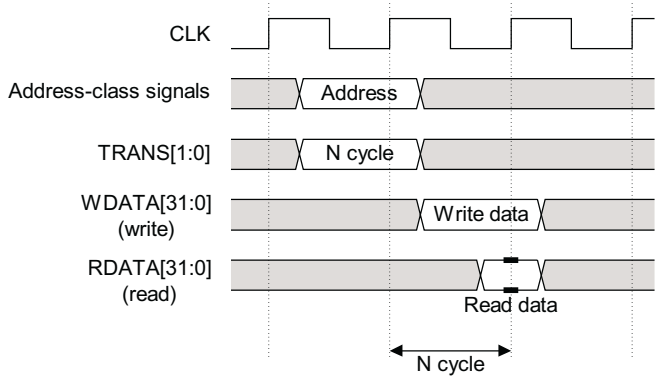


Figure 3-2 Nonsequential memory cycle

The ARM7TDMI-S processor can perform back to back nonsequential memory cycles. This happens, for example, when an STR instruction is executed, as shown in Figure 3-3. If you are designing a memory controller for the ARM7TDMI-S processor, and your memory system is unable to cope with this case, you must use the **CLKEN** signal to extend the bus cycle to allow sufficient cycles for the memory system. See *Using CLKEN to control bus cycles* on page 3-17.

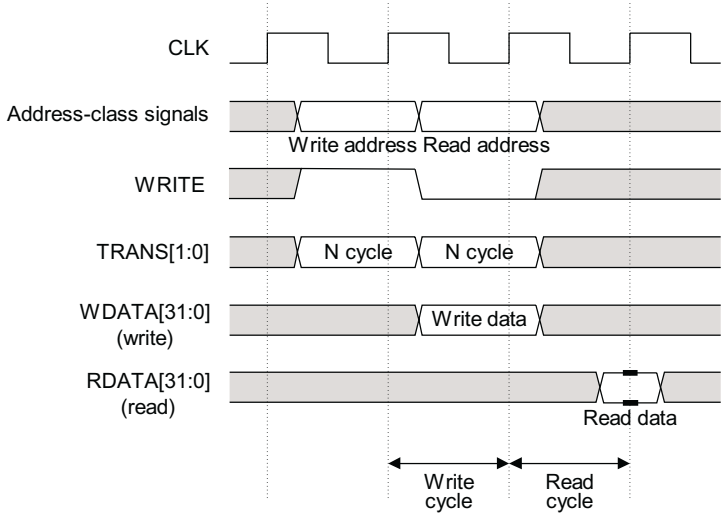


Figure 3-3 Back to back memory cycles

3.3.2 Sequential cycles

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

During a sequential cycle, the ARM7TDMI-S processor requests a memory location that is part of a sequential burst. If this is the first cycle in the burst, the address can be the same as the previous internal cycle. Otherwise the address is incremented from the previous cycle:

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

Bursts of byte accesses are not possible.

A burst always starts with an N cycle or a merged I-S cycle (see *Merged I-S cycles* on page 3-8), and continues with S cycles. A burst comprises transfers of the same type. The **ADDR[31:0]** signal increments during the burst. The other address class signals remain the same throughout the burst.

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

Table 3-2 Burst types

Burst type	Address increment	Cause
Word read	4 bytes	ARM7TDMI-S code fetches, or LDM instruction
Word write	4 bytes	STM instruction
Halfword read	2 bytes	Thumb code fetches

All accesses in a burst are of the same width, direction, and protection type. For more details, see *Addressing signals* on page 3-10.

An example of a burst access is shown in Figure 3-4 on page 3-8.

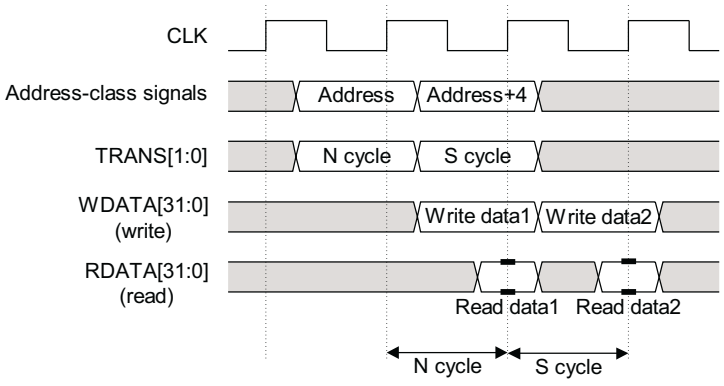


Figure 3-4 Sequential access cycles

3.3.3 Internal cycles

During an internal cycle, the ARM7TDMI-S 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-S processor broadcasts the address for the next access, so that decode can start, but the memory controller must not commit to a memory access. This is described in *Merged I-S cycles*.

3.3.4 Merged I-S cycles

Where possible, the ARM7TDMI-S processor performs an optimization on the bus to allow extra time for memory decode. When this happens, the address of the next memory cycle is broadcast during an internal cycle on this bus. This enables the memory controller to decode the address, but it must not initiate a memory access during this cycle. In a merged I-S cycle, the next cycle is a sequential cycle to the same memory location. This commits to the access, and the memory controller must initiate the memory access. This is shown in Figure 3-5 on page 3-9.

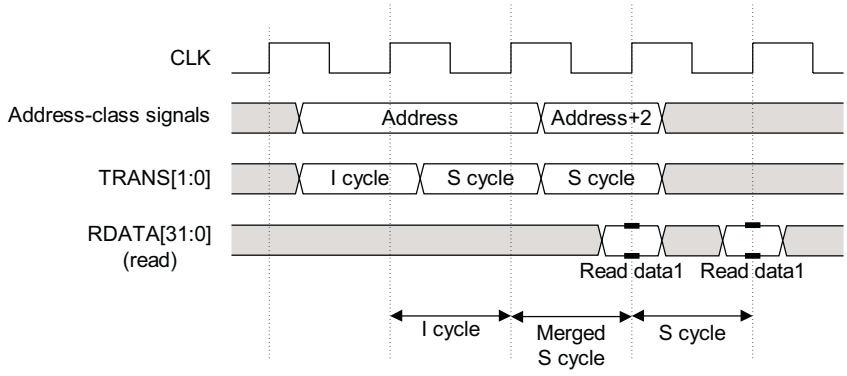


Figure 3-5 Merged I-S cycle

Note

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

3.3.5 Coprocessor register transfer cycles

During a coprocessor register transfer cycle, the ARM7TDMI-S 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 coprocessor interface is described in Chapter 4 *Coprocessor Interface*.

3.4 Addressing signals

The address class signals are described in the following sections:

- *ADDR[31:0]*]
- *WRITE*
- *SIZE[1:0]*
- *PROT[1:0]* on page 3-11
- *LOCK* on page 3-12
- *CPTBIT* on page 3-12.

3.4.1 ADDR[31:0]

ADDR[31:0] is the 32-bit address bus which 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 must ignore the bottom two bits, **ADDR[1:0]**, and when a halfword access is signaled the memory system must ignore the bottom bit, **ADDR[0]**.

3.4.2 WRITE

WRITE specifies the direction of the transfer. **WRITE** indicates an ARM7TDMI-S core write cycle when HIGH, and an ARM7TDMI-S core read cycle when LOW. A burst of S cycles is always either a read burst or a write burst. The direction cannot be changed in the middle of a burst.

3.4.3 SIZE[1:0]

The **SIZE[1:0]** bus encodes the size of the transfer. The ARM7TDMI-S processor can transfer word, halfword, and byte quantities. This is encoded on **SIZE[1:0]** as shown in Table 3-3 on page 3-11.

Table 3-3 Transfer widths

SIZE[1:0]	Transfer width
00	Byte
01	Halfword
10	Word
11	Reserved

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

Note

A writable memory system for the ARM7TDMI-S processor must have individual byte write enables. Both the C Compiler and the ARM debug tool chain (for example, Multi-ICE) assume that arbitrary bytes in the memory can be written. If individual byte write capability is not provided, it might not be possible to use either of these tools.

3.4.4 PROT[1:0]

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

Table 3-4 PROT[1:0] encoding

PROT[1:0]	Mode	Opcode or data
00	User	Opcode
01	User	Data
10	Privileged	Opcode
11	Privileged	Data

3.4.5 LOCK

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

3.4.6 CPTBIT

CPTBIT indicates the operating state of the ARM7TDMI-S processor:

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

3.5 Data timed signals

The data timed signals are described in the following sections:

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

3.5.1 WDATA[31:0]

WDATA[31:0] is the write data bus. All data written out from the ARM7TDMI-S processor is broadcast on this bus. Data transfers from the ARM7TDMI-S core to a coprocessor also use this bus during C-cycles. In normal circumstances, a memory system must sample the **WDATA[31:0]** bus on the rising edge of **CLK** at the end of a write bus cycle. The **WDATA[31:0]** value is valid only during write cycles.

3.5.2 RDATA[31:0]

RDATA[31:0] is the read data bus, and is used by the ARM7TDMI-S core to fetch both opcodes and data. The **RDATA[31:0]** signal is sampled on the rising edge of **CLK** at the end of the bus cycle. **RDATA[31:0]** is also used during C-cycles to transfer data from a coprocessor to the ARM7TDMI-S core.

3.5.3 ABORT

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

If **ABORT** is asserted on a data access, it causes the ARM7TDMI-S processor to take the Data Abort trap. If it is asserted on an opcode fetch, the abort is tracked down the pipeline, and the Prefetch Abort trap is taken if the instruction is executed.

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

For more details about aborts, see *Abort* on page 2-22.

3.5.4 Byte and halfword accesses

The ARM7TDMI-S processor indicates the size of a transfer using the **SIZE[1:0]** signals. These are encoded as shown in Table 3-5.

Table 3-5 Transfer size encoding

SIZE[1:0]	Transfer width
00	Byte
01	Halfword
10	Word
11	Reserved

All writable memory in an ARM7TDMI-S processor-based system supports the writing of individual bytes to allow the use of the C Compiler and the ARM debug tool chain (for example, Multi-ICE).

The address produced by the ARM7TDMI-S processor is always a byte address. However, the memory system ignores the insignificant bits of the address. The significant address bits are shown in Table 3-6.

Table 3-6 Significant address bits

SIZE[1:0]	Width	Significant address bits
00	Byte	**ADDR[31:0]**
01	Halfword	**ADDR[31:1]**
10	Word	**ADDR[31:2]**

When a halfword or byte read is performed, a 32-bit memory system can return the complete 32-bit word, and the ARM7TDMI-S processor extracts the valid halfword or byte field from it. The fields extracted depend on the state of the **CFGBIGEND** signal, which determines the endianness of the system (see *Memory formats* on page 2-4).

The fields extracted by the ARM7TDMI-S processor are shown in Table 3-7.

Table 3-7 Word accesses

SIZE[1:0]	ADDR[1:0]	Little-endian CFGBIGEND = 0	Big-endian CFGBIGEND = 1
10	XX	**RDATA[31:0]**	**RDATA[31:0]**

When connecting 8-bit to 16-bit memory systems to the ARM7TDMI-S processor, make sure that the data is presented to the correct byte lanes on the ARM7TDMI-S processor as shown in Table 3-8 and Table 3-9.

Table 3-8 Halfword accesses

SIZE[1:0]	ADDR[1:0]	Little-endian CFGBIGEND = 0	Big-endian CFGBIGEND = 1
01	0X	**RDATA[15:0]**	**RDATA[31:16]**
01	1X	**RDATA[31:16]**	**RDATA[15:0]**

Table 3-9 Byte accesses

SIZE[1:0]	ADDR[1:0]	Little-endian CFGBIGEND = 0	Big-endian CFGBIGEND = 1
00	00	**RDATA[7:0]**	**RDATA[31:24]**
00	01	**RDATA[15:8]**	**RDATA[23:16]**
00	10	**RDATA[23:16]**	**RDATA[15:8]**
00	11	**RDATA[31:24]**	**RDATA[7:0]**

Writes

When the ARM7TDMI-S processor performs a byte or halfword write, the data being written is replicated across the bus, as illustrated in Figure 3-6 on page 3-16. The memory system can use the most convenient copy of the data. A writable memory system must be capable of performing a write to any single byte in the memory system. This capability is required by the ARM C Compiler and the Debug tool chain.

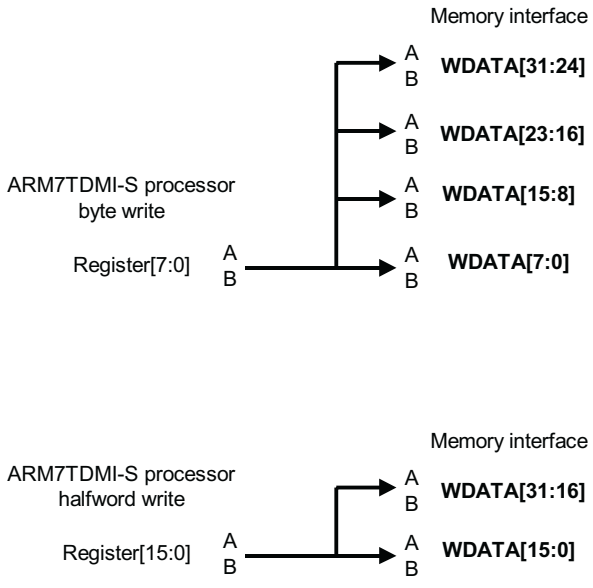


Figure 3-6 Data replication

3.6 Using CLKEN to control bus cycles

The pipelined nature of the ARM7TDMI-S processor bus interface means that there is a distinction between *clock* cycles and *bus* cycles. **CLKEN** can be used to stretch a *bus* cycle, so that it lasts for many *clock* cycles. The **CLKEN** input extends the timing of bus cycles in increments of complete **CLK** cycles:

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

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

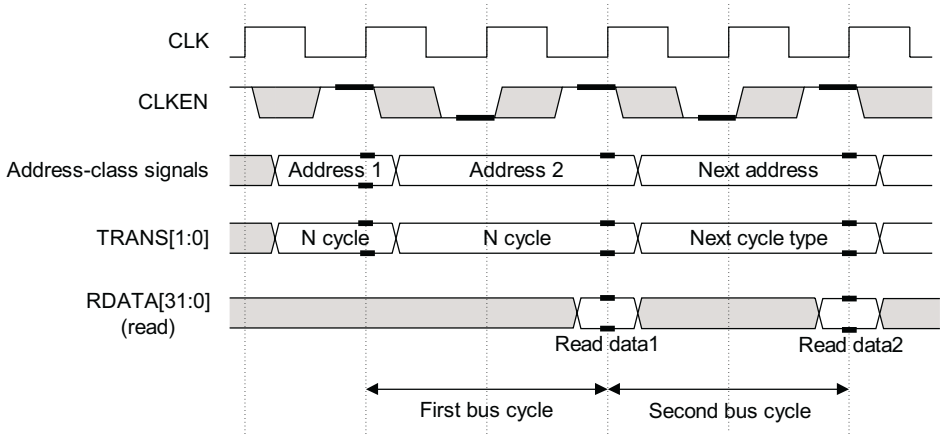


Figure 3-7 Use of CLKEN

Note

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

Chapter 4
Coprocessor Interface

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

4.1 About coprocessors

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

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

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

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

The ARM7TDMI-S core:

1. Evaluates the condition codes to determine whether the instruction must be executed by the coprocessor, then signals this to any coprocessors in the system (using **CPnI**).
2. Generates any addresses that are required by the instruction, including prefetching the next instruction to refill the pipeline.
3. Takes the undefined instruction trap if no coprocessor accepts the instruction.

The coprocessor:

1. Decodes instructions to determine whether it can accept the instruction.
2. Indicates whether it can accept the instruction (by signaling on **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

You can connect up to 16 coprocessors into a system, each with a unique coprocessor ID number to identify it. The ARM7TDMI-S processor contains two internal coprocessors:

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

Therefore, you cannot assign external coprocessors to coprocessor numbers 14 and 15. Other coprocessor numbers have also been reserved by ARM. Coprocessor availability is shown in Table 4-1.

Table 4-1 Coprocessor availability

Coprocessor number	Allocation
15	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 E-mail with `coprocessor` in the subject line to `info@arm.com` for up to date information on coprocessor numbers that have already been allocated.

4.2 Coprocessor interface signals

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

The clock and clock control signals are:

- **CLK**
- **CLKEN**
- **nRESET**.

The pipeline-following signals are:

- **CPnMREQ**
- **CPSEQ**
- **CPnTRANS**
- **CPnOPC**
- **CPTBIT**.

The handshake signals are:

- **CPnI**
- **CPA**
- **CPB**.

The data signals are:

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

These signals and their use are described in:

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

4.3 Pipeline-following signals

Every coprocessor in the system must contain a pipeline follower to track the instructions executing in the ARM7TDMI-S core pipeline. The coprocessors connect to the ARM7TDMI-S processor input data bus, **RDATA[31:0]**, over which instructions are fetched, and to **CLK** and **CLKEN**.

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 (**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 **CLKEN** is HIGH (except for reset).
- An instruction must be loaded into the pipeline on the rising edge of **CLK**, and only when **CPnOPC**, **CPnMREQ**, and **CPTBIT** were *all* LOW in the previous bus cycle.

  These conditions indicate that this cycle is an ARM state opcode Fetch, so the new opcode must be sampled into the pipeline.
- The pipeline must be advanced on the rising edge of **CLK** when **CPnOPC**, **CPnMREQ**, and **CPTBIT** are *all* LOW in the current bus cycle.

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

Any instructions that are flushed from the ARM7TDMI-S processor pipeline never signal on **CPnI** that they have entered Execute, and so they are automatically flushed from the coprocessor pipeline by the prefetches required to refill the pipeline.

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

4.4 Coprocessor interface handshaking

The ARM7TDMI-S 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
CPnI	ARM7TDMI-S to coprocessor	Not coprocessor instruction
CPA	Coprocessor to ARM7TDMI-S	Coprocessor absent
CPB	Coprocessor to ARM7TDMI-S	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 has a coprocessor number that matches the coprocessor ID of the coprocessor.

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

1. The coprocessor attempts to execute the instruction.
2. The coprocessor signals back to the ARM7TDMI-S core using **CPA** and **CPB**.

4.4.2 The ARM7TDMI-S core

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

1. The coprocessor instruction has reached the Execute stage of the pipeline. (It might not if it was preceded by a branch.)
2. The instruction has passed its conditional execution tests.
3. A coprocessor in the system has signaled on **CPA** and **CPB** that it is able to accept the instruction.

If all these requirements are met, the ARM7TDMI-S core signals by taking **CPnI** LOW, committing the coprocessor to the execution of the coprocessor instruction.

4.4.3 Coprocessor signaling

The coprocessor signals as follows:

Coprocessor absent

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

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.

Coprocessor busy (busy-wait)

If a coprocessor can accept an instruction, but is currently unable to process that request, it can stall the ARM7TDMI-S core 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.

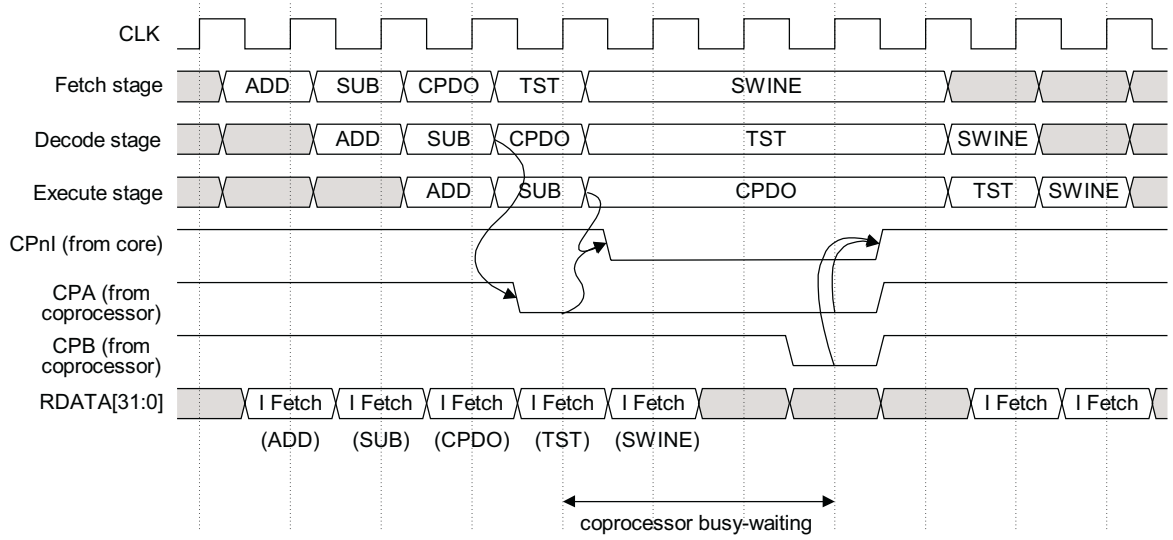


Figure 4-1 Coprocessor busy-wait sequence

4.4.4 Consequences of busy-waiting

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

Caution

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

4.4.5 Coprocessor register transfer instructions

The coprocessor register transfer instructions, MCR and MRC, transfer data between a register in the ARM7TDMI-S 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.

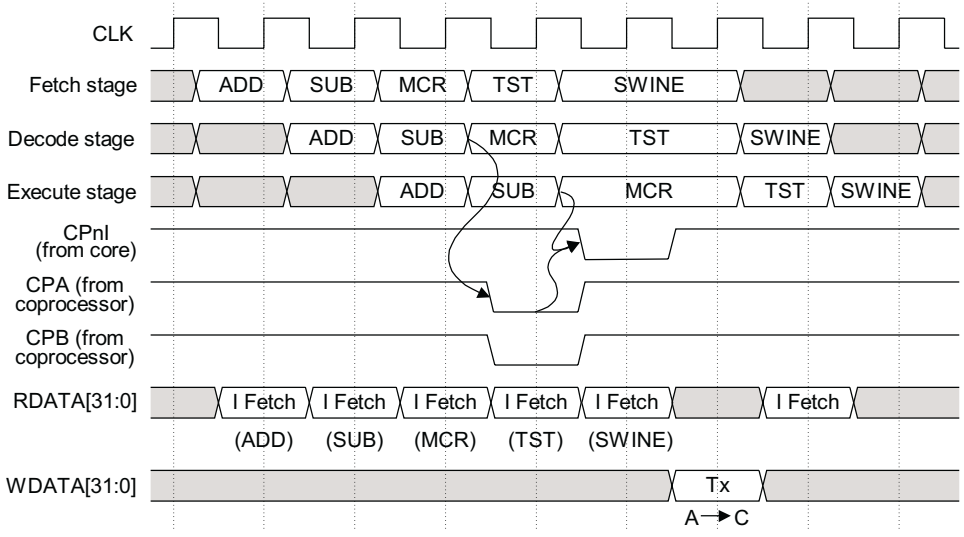


Figure 4-2 Coprocessor register transfer sequence

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-S core and the coprocessor as a result of this operation. An example sequence is shown in Figure 4-3.

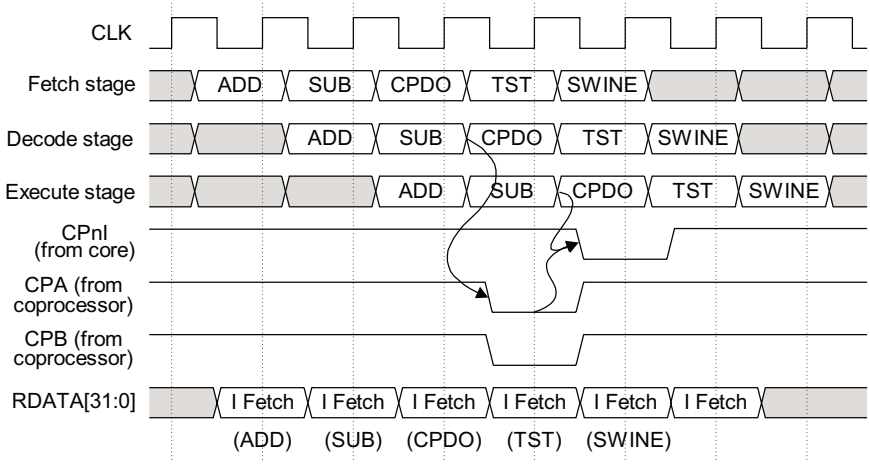


Figure 4-3 Coprocessor data operation sequence

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 a coprocessor must not transfer more than 16 words of data in a single instruction. An example sequence is shown in Figure 4-4.

Note

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

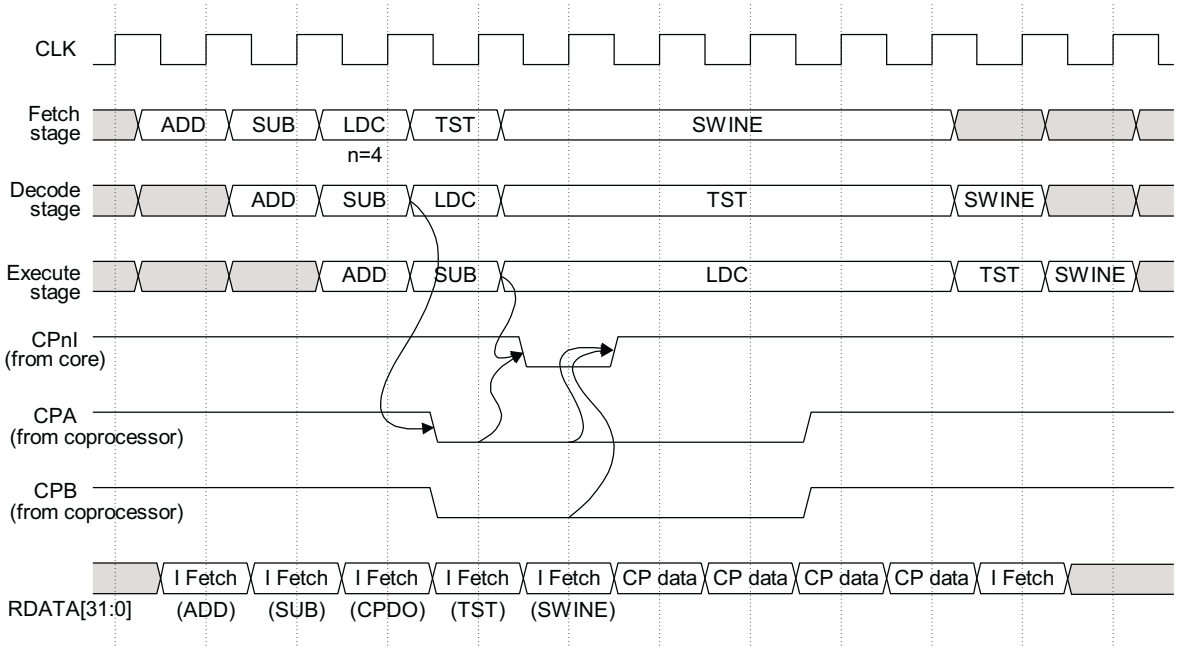


Figure 4-4 Coprocessor load sequence

4.5 Connecting coprocessors

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

- transfer data from memory (instruction stream and LDC)
- write data from the ARM7TDMI-S (MCR)
- read data to the ARM7TDMI-S (MRC).

4.5.1 Connecting a single coprocessor

An example of how to connect a coprocessor into an ARM7TDMI-S processor-based system is shown in Figure 4-5.

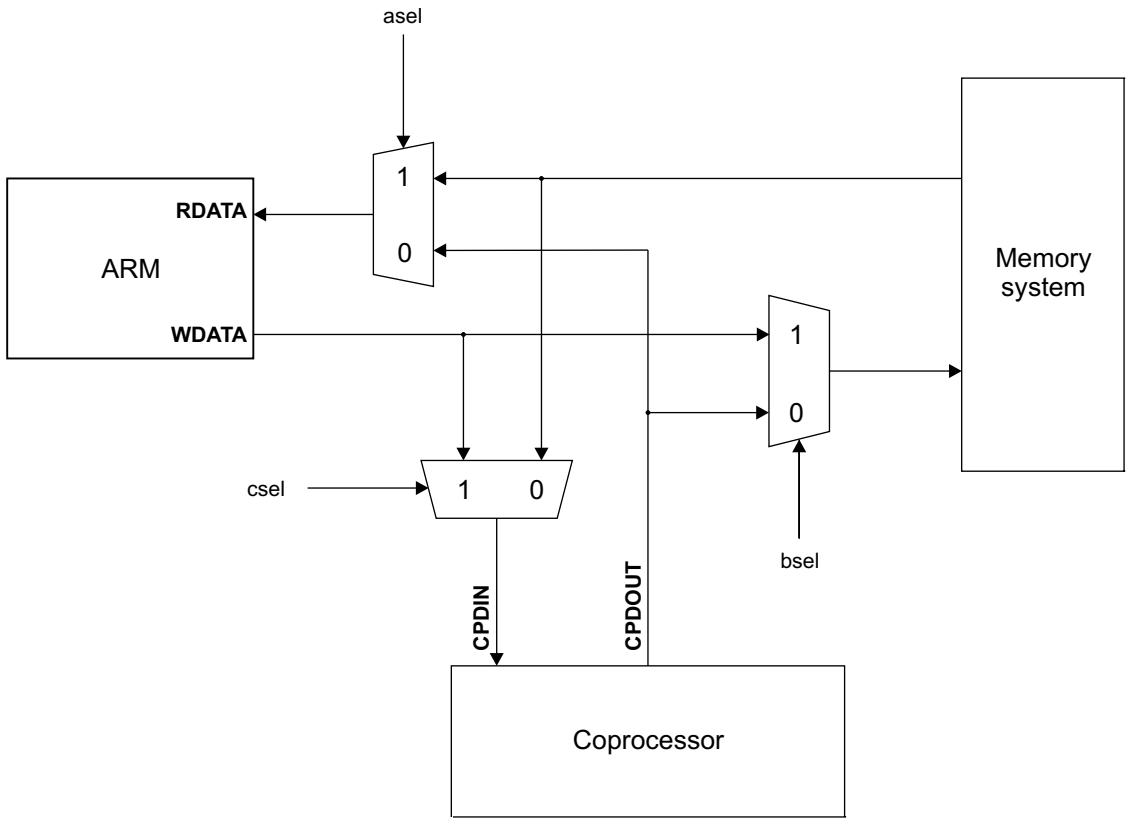


Figure 4-5 Coprocessor connections

The fragments of Verilog that describe the register logic to derive `asel`, `bsel`, and `csel` from the relevant ARM7TDMI-S processor or ARM7TDMI processor pins are described in this section.

The logic for asel, bsel, and csel is as follows:

```
assign asel = ~(cprt | (cpdt & nRW_r));
assign bsel = ~cpdt;
assign csel = cprt;



assign cpdt = ~nMREQ_r & ~CPA_r2 & nOPC_r;
assign cprt = nMREQ_r & SEQ_r;
```

Note

`cpdt` shows that the current cycle is a load or store cycle due to an LDC or STC instruction.
`cprt` shows that the current cycle is a coprocessor register transfer cycle.

The other signals used to drive these terms are as follows:

```
always @(posedge CLK)
if (CLKEN)
    begin
        nMREQ_r <= CPnMREQ; // Output from ARM7TDMI-S
            SEQ_r   <= CPSEQ;   // Output from ARM7TDMI-S
            nOPC_r  <= CPnOPC;  // Output from ARM7TDMI-S
            nRW_r   <= WRITE;   // Output from ARM7TDMI-S
            CPA_r   <= CPA;     // Input to ARM7TDMI-S
            CPA_r2  <= CPA_r;
        end
```

Note

If you are building a system with an ETM and an ARM7TDMI-S processor, you must directly connect the ETM7 **RDATA[31:0]** and **WDATA[31:0]** to the ARM7TDMI-S **RDATA[31:0]** and **WDATA[31:0]** buses. This enables the ETM to correctly trace coprocessor instructions.

4.5.2 Connecting multiple coprocessors

If you have multiple coprocessors in your system, connect the handshake signals as shown in Table 4-3.

Table 4-3 Handshake signal connections

Signal	Connection
CPnI	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-S processor

You must also multiplex the output data from the coprocessors.

4.6 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 coprocessor-specific outputs from the ARM7TDMI-S processor must be left unconnected:

- **CPnMREQ**
- **CPSEQ**
- **CPnTRANS**
- **CPnOPC**
- **CPnI**
- **CPTBIT**.

4.7 Undefined instructions

The ARM7TDMI-S processor implements full ARM architecture v4T undefined instruction handling. This means that any instruction defined in the *ARM Architecture Reference Manual* as `UNDEFINED`, automatically causes the ARM7TDMI-S processor to take the undefined instruction trap. Any coprocessor instructions that are not accepted by a coprocessor also result in the ARM7TDMI-S processor taking the undefined instruction trap.

4.8 Privileged instructions

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

Table 4-4 CPnTRANS signal meanings

CPnTRANS	Meaning
LOW	User mode instruction
HIGH	Privileged mode instruction

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

Note

If a User mode process (**CPnTRANS** LOW) tries to access a coprocessor instruction that can only be executed in a privileged mode, the coprocessor must respond with **CPA** and **CPB** HIGH. This causes the ARM7TDMI-S processor to take the undefined instruction trap.

Chapter 5
Debugging Your System

This chapter describes the debug features of the ARM7TDMI-S processor. It contains the following sections:

5.1 About debugging your system

The advanced debugging features of the ARM7TDMI-S (Rev 4) processor make it easier to develop application software, operating systems, and the hardware itself.

5.1.1 A typical debug system

The ARM7TDMI-S processor forms one component of a debug system that interfaces from the high-level debugging that you perform to the low-level interface supported by the ARM7TDMI-S processor. Figure 5-1 shows a typical debug system.

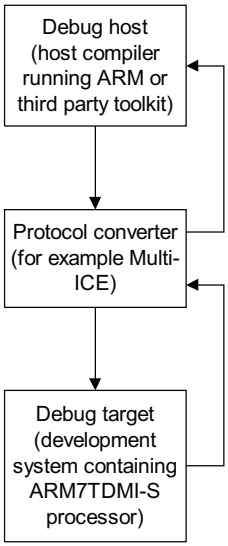


Figure 5-1 Typical debug system

A debug system usually has three parts:

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

Protocol converter This interfaces between the high-level commands issued by the debug host and the low-level commands of the ARM7TDMI-S processor JTAG interface. Typically it interfaces to the host through an interface such as an enhanced parallel port.

Debug target The ARM7TDMI-S processor has hardware extensions that ease debugging at the lowest level. These extensions enable you to:

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

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

5.2 Controlling debugging

The major blocks of the ARM7TDMI-S processor are:

ARM CPU core This has hardware support for debug.

EmbeddedICE-RT macrocell

A set of registers and comparators that you use to generate debug exceptions (such as breakpoints). This unit is described in *The EmbeddedICE-RT macrocell* on page 5-14.

TAP controller Controls the action of the scan chains using a JTAG serial interface. For more details, see *The TAP controller* on page 5-26.

These blocks are shown in Figure 5-2.

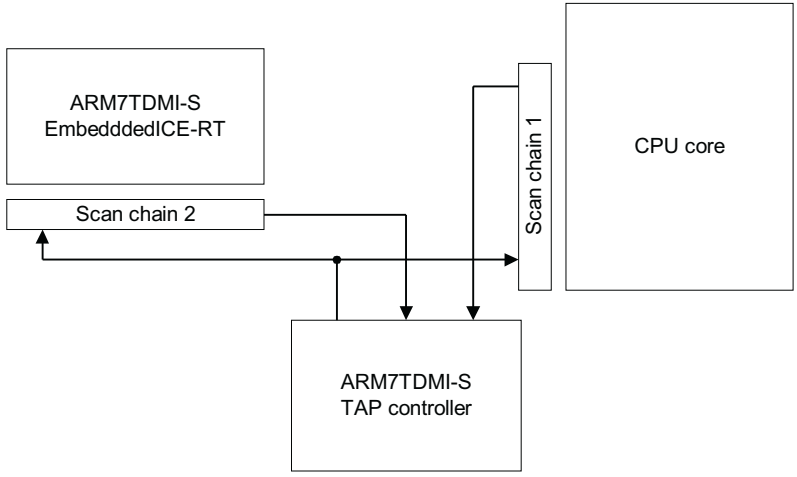


Figure 5-2 ARM7TDMI-S block diagram

5.2.1 Debug modes

You can perform debugging in either of the following modes:

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

For more information, see *Entry into debug state* on page 5-7.

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

For more information, see *Monitor mode debugging* on page 5-18.

5.2.2 Examining system state during debugging

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

In halt mode, this enables instructions to be inserted serially into the core pipeline without using the external data bus. For example, when in debug state, a *Store Multiple* (`STM`) can be inserted into the instruction pipeline to export the contents of the ARM7TDMI-S processor registers. This data can be serially shifted out without affecting the rest of the system. For more information, see *Examining the core and the system in debug state* on page 5-39.

In monitor mode, the JTAG interface is used to transfer data between the debugger and a simple monitor program running on the ARM7TDMI-S core.

For detailed information about the scan chains and the JTAG interface, see *Scan chains and the JTAG interface* on page 5-24.

5.3 Entry into debug state

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

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

Note

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

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

- address bus
- data bus
- control signals.

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

When a breakpoint or watchpoint is generated, there might be a delay before the ARM7TDMI-S core enters debug state. When it enters debug state, the **DBGACK** signal is asserted. The timing for an externally-generated breakpoint is shown in Figure 5-3 on page 5-8.

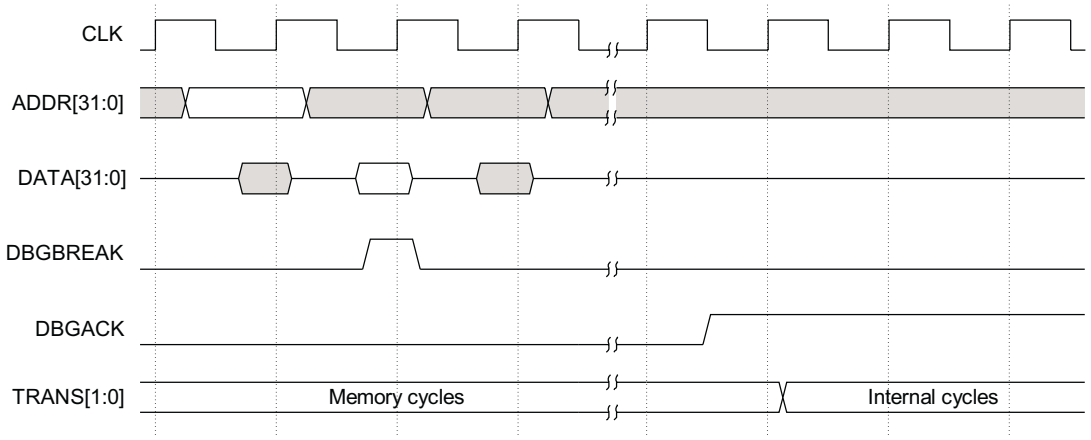


Figure 5-3 Debug state entry

5.3.1 Entry into debug state on breakpoint

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

Breakpointed instructions are not executed. Instead, the ARM7TDMI-S core enters debug state. When you examine the internal state, you see the state before the breakpointed instruction. When your examination is complete, remove the breakpoint. Program execution restarts from the previously-breakpointed instruction.

When a breakpointed conditional instruction reaches the Execute stage of the pipeline, the breakpoint is always taken if the system is in halt mode. The ARM7TDMI-S core enters debug state regardless of whether the instruction condition is met.

A breakpointed instruction does not cause the ARM7TDMI-S 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 ARM7TDMI-S processor flushes the instruction pipeline, so canceling the breakpoint.
- An exception occurs, causing the ARM7TDMI-S processor to flush the instruction pipeline, and cancel the breakpoint. In normal circumstances, on exiting from an exception, the ARM7TDMI-S core branches back to the instruction that would have been executed next before the exception occurred. In this case, the pipeline is refilled and the breakpoint is reflagged.

5.3.2 Entry into debug state on watchpoint

Watchpoints occur on data accesses. In halt mode, the core processing stops. In monitor mode, an abort exception is executed (see *Abort* on page 2-22). A watchpoint is always taken, but a core in halt mode might not enter debug state immediately because the current instruction always completes. If the current instruction is a multiword load or store (an `LDM` or `STM`), many cycles can elapse before the watchpoint is taken.

On a watchpoint, the following sequence occurs:

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

Note

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

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

5.3.3 Entry into debug state on debug request

An ARM7TDMI-S core in halt mode can be forced into debug state on debug request in either of the following ways:

- through EmbeddedICE-RT programming (see *Programming breakpoints* on page 5-53, and *Programming watchpoints* on page 5-55.)
- by asserting the **DBGRQ** pin.

When the **DBGRQ** pin has been asserted, the core normally enters debug state at the end of the current instruction. However, when the current instruction is a busy-waiting access to a coprocessor, the instruction terminates, and the ARM7TDMI-S core enters debug state immediately. This is similar to the action of **nIRQ** and **nFIQ**.

5.3.4 Action of the ARM7TDMI-S in debug state

When the ARM7TDMI-S processor enters debug state, the core forces **TRANS[1:0]** to indicate internal cycles. This action enables the rest of the memory system to ignore the ARM7TDMI-S core and to function as normal. Because the rest of the system continues to operate, the ARM7TDMI-S core is forced to ignore aborts and interrupts.

Caution

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

Note

The system must not change the **CFGBIGEND** signal during debug. From the point of view of the programmer, if **CFGBIGEND** changes, the ARM7TDMI-S processor changes, with the debugger unaware that the core has reset. You must also ensure that **nRESET** is held stable during debug. When the system applies reset to the ARM7TDMI-S processor (that is, **nRESET** is driven LOW), the ARM7TDMI-S processor state changes with the debugger unaware that the core has reset.

5.3.5 Clocks

The system and test clocks must be synchronized externally to the macrocell. The ARM Multi-ICE debug agent directly supports one or more cores within an ASIC design. Synchronizing off-chip debug clocking with the ARM7TDMI-S macrocell requires a three-stage synchronizer. The off-chip device (for example, Multi-ICE) issues a **TCK** signal and waits for the **RTCK** (Returned **TCK**) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next **TCK** until after **RTCK** is received.

Figure 5-4 on page 5-11 shows this synchronization.

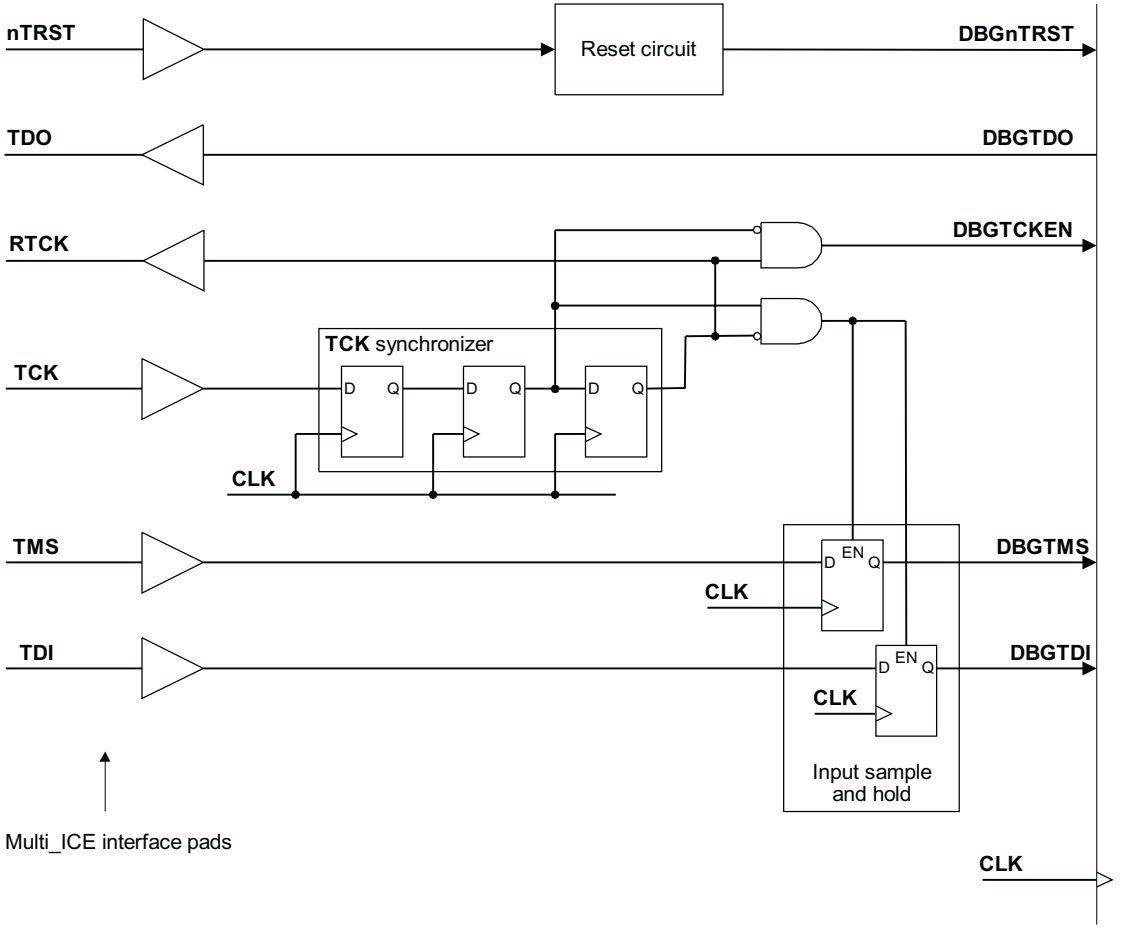


Figure 5-4 Clock synchronization

Note

All the D-types shown in Figure 5-4 are reset by **DBGnTRST**.

5.4 Debug interface

The ARM7TDMI-S processor debug interface is based on IEEE Std. 1149.1- 1990, *Standard Test Access Port and Boundary-Scan Architecture*. Refer to this standard for an explanation of the terms used in this chapter, and for a description of the TAP controller states.

5.4.1 Debug interface signals

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

- **DBGBREAK** and **DBGRQ** are system requests for the ARM7TDMI-S core to enter debug state
- **DBGACK** is used by the ARM7TDMI-S core to flag back to the system that it is in debug state.

5.5 ARM7TDMI-S core clock domains

The ARM7TDMI-S processor has a single clock, **CLK**, that is qualified by two clock enables:

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

During normal operation, **CLKEN** conditions **CLK** to clock the core. When the ARM7TDMI-S processor is in debug state, **DBGTCKEN** conditions **CLK** to clock the core.

5.6 The EmbeddedICE-RT macrocell

The ARM7TDMI-S processor EmbeddedICE-RT macrocell module provides integrated on-chip debug support for the ARM7TDMI-S core.

EmbeddedICE-RT is programmed serially using the ARM7TDMI-S processor TAP controller. Figure 5-5 illustrates the relationship between the core, EmbeddedICE-RT, and the TAP controller, showing only the signals that are pertinent to EmbeddedICE-RT.

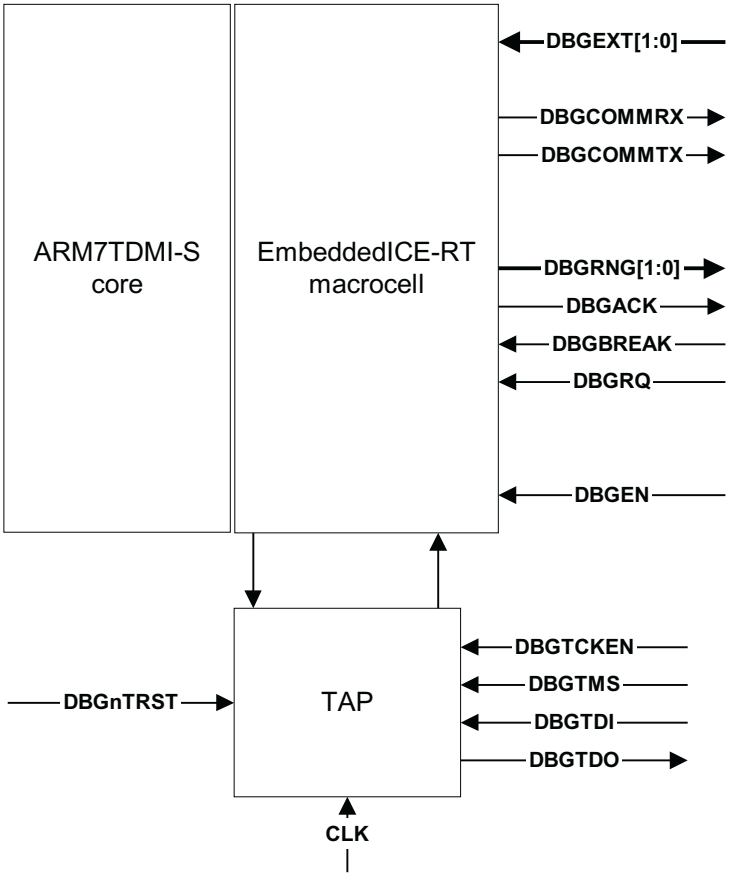


Figure 5-5 The ARM7TDMI-S core, TAP controller, and EmbeddedICE-RT macrocell

The EmbeddedICE-RT logic comprises the following:

Two real-time watchpoint units

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

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

For more details, see *Watchpoint unit registers* on page 5-48.

Abort status register

This register identifies the cause of an abort exception entry. For more information, see *Abort status register* on page 5-56.

Debug Communications Channel (DCC)

The DCC passes information between the target and the host debugger. For more information, see *The debug communications channel* on page 5-20.

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

- *Debug control register* on page 5-57
- *Debug status register* on page 5-60.

The locations of the EmbeddedICE-RT registers are given in *EmbeddedICE-RT register map* on page 5-17.

5.7 Disabling EmbeddedICE-RT

You can disable EmbeddedICE-RT in two ways:

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

When **DBGEN** is LOW:

- **DBGBREAK** and **DBGRQ** are ignored by the core
- **DBGACK** is forced LOW by the ARM7TDMI-S core
- interrupts pass through to the processor uninhibited
- the EmbeddedICE-RT logic enters low-power mode.

Caution

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

Temporarily By setting bit 5 in the debug control register (described in *Debug control register* on page 5-57). Bit 5 is also known as the EmbeddedICE-RT disable bit.

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

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

5.8 EmbeddedICE-RT register map

The locations of the EmbeddedICE-RT registers are shown in Table 5-1.

Table 5-1 Function and mapping of EmbeddedICE-RT registers

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

5.9 Monitor mode debugging

The ARM7TDMI-S (Rev 4) processor contains logic that enables the debugging of a system without stopping the core entirely. This means that critical interrupt routines continue to be serviced while the core is being interrogated by the debugger.

5.9.1 Enabling monitor mode

The debugging mode is controlled by bit 4 of the debug control register (described in *Debug control register* on page 5-57). Bit 4 of this register is also known as the monitor mode enable bit:

Bit 4 set Enables the monitor mode features of the ARM7TDMI-S processor. When this bit is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the ARM7TDMI-S core to enter abort mode, taking the Prefetch or Data Abort vectors respectively.

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

5.9.2 Restrictions on monitor-mode debugging

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

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

The fact that an abort has been generated by the monitor mode is recorded in the abort status register in coprocessor 14 (see *Abort status register* on page 5-56).

The monitor mode enable bit does not put the ARM7TDMI-S processor into debug state. For this reason, it is necessary to change the contents of the watchpoint registers while external memory accesses are taking place, rather than changing them when in debug state where the core is halted.

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

1. Disable the watchpoint unit by setting bit 5 in the debug control register (also known as the EmbeddedICE-RT disable bit).
2. Poll the debug control register until the EmbeddedICE-RT disable bit is read back as set.
3. Change the other registers.
4. Re-enable the watchpoint unit by clearing the EmbeddedICE-RT disable bit in the debug control register.

5.10 The debug communications channel

The ARM7TDMI-S (Rev 4) EmbeddedICE-RT contains a *Debug Communication Channel* (DCC) for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The DCC comprises two registers, as follows:

DCC control register

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

DCC data register

A 32-bit register, used for data transfers between the debugger and the processor. For more details, see *Communications through the DCC* on page 5-22.

These registers occupy fixed locations in the EmbeddedICE-RT memory map, as shown in Table 5-1 on page 5-17. They are accessed from the processor using `MCR` and `MRC` instructions to coprocessor 14.

The registers are accessed as follows:

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

By the processor Through coprocessor register transfer instructions.

5.10.1 DCC control register

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

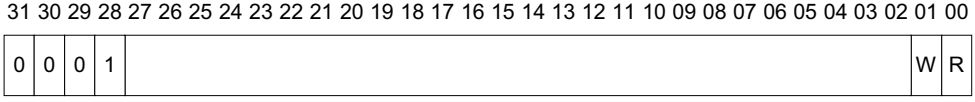


Figure 5-6 DCC control register

The DCC control register bit assignments are shown in Table 5-2.

Table 5-2 DCC control register bit assignments

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

Note

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

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

Instructions

The following instructions must be used:

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

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

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

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

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

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

Note

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

5.10.2 Communications through the DCC

Messages can be sent and received through the DCC.

Sending a message to the debugger

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

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

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

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

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

Receiving a message from the debugger

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

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

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

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

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

5.11 Scan chains and the JTAG interface

There are two JTAG-style scan chains within the ARM7TDMI-S processor. These allow debugging and EmbeddedICE-RT programming.

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

5.11.1 Scan chain implementation

The two scan paths are referred to as scan chain 1 and scan chain 2. They are shown in Figure 5-7. Scan chain 0 is not implemented on the ARM7TDMI-S processor.

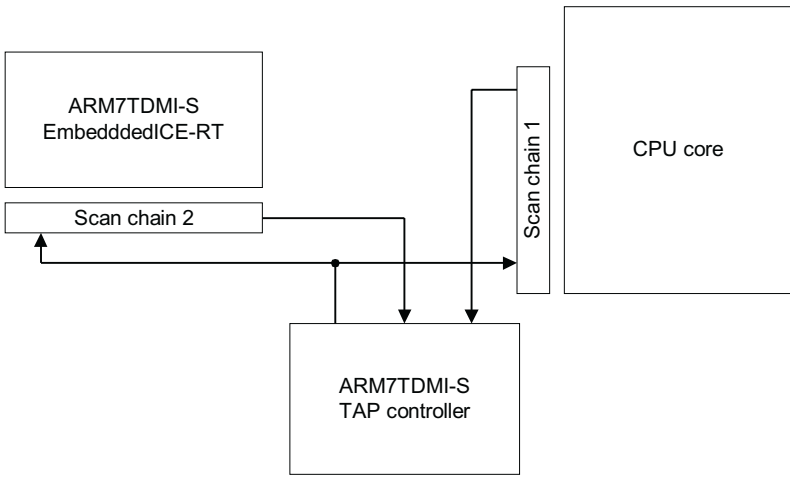


Figure 5-7 ARM7TDMI-S scan chain arrangements

Scan chain 1

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

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

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

Scan chain 2

Scan chain 2 enables access to the EmbeddedICE-RT registers. See *Test data registers* on page 5-31 for details.

5.11.2 Controlling the JTAG interface

The JTAG interface is driven by the currently-loaded instruction in the instruction register (described in *Instruction register* on page 5-32). The loading of instructions is controlled by the *Test Access Port* (TAP) controller.

For more information about the TAP controller, see *The TAP controller* on page 5-26.

5.12 The TAP controller

The TAP controller is a state machine that determines the state of the ARM7TDMI-S boundary-scan test signals **DBGTDI** and **DBGTDO**. Figure 5-8shows the state transitions that occur in the TAP controller.

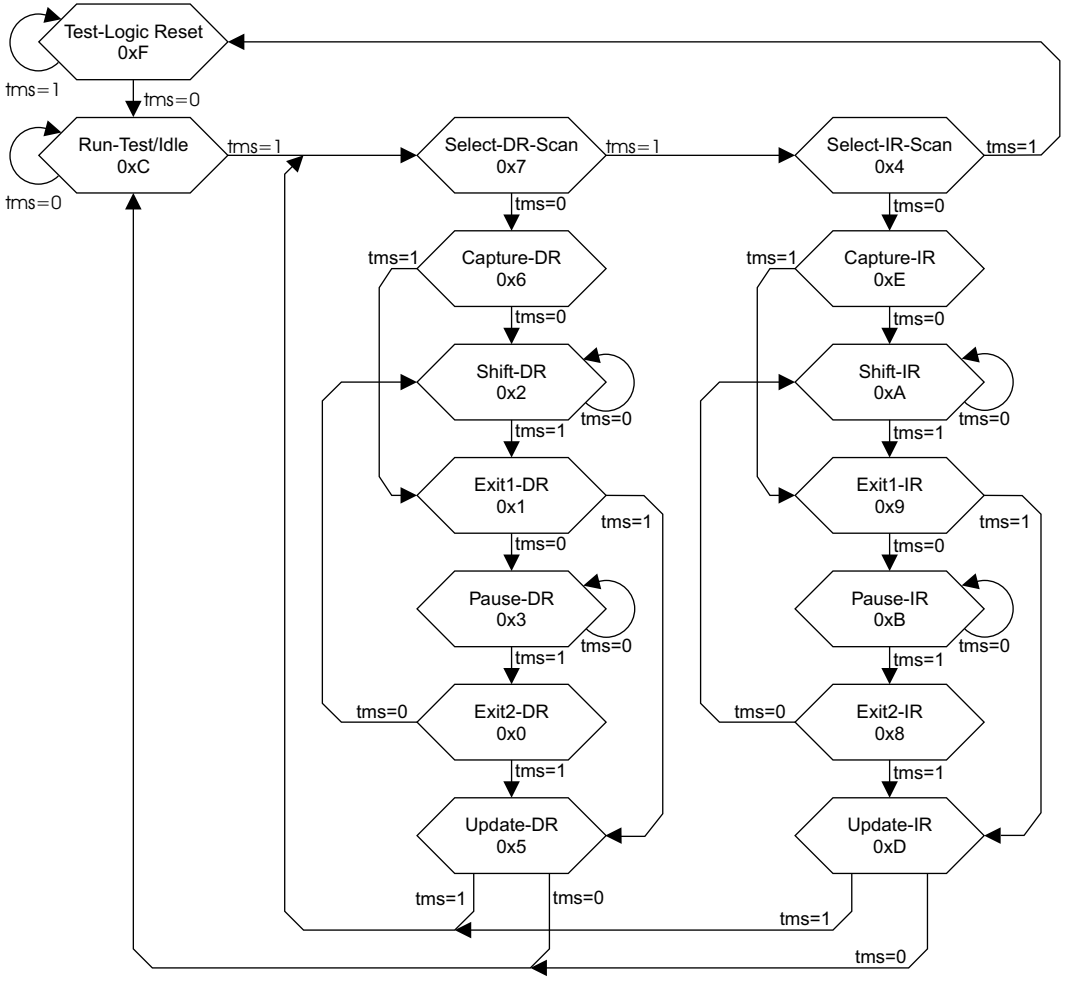


Figure 5-8 Test access port controller state transitions

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

5.12.1 Resetting the TAP controller

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

- When the boundary-scan interface is to be used, **DBGnTRST** must be driven LOW and then HIGH again.
- When the boundary-scan interface is not to be used, you can tie the **DBGnTRST** input LOW.

Note

A clock on **CLK** with **DBGTCKEN** HIGH is not necessary to reset the device.

The action of reset is as follows:

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

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

5.13 Public JTAG instructions

Table 5-3 shows the public JTAG instructions.

Table 5-3 Public instructions

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

In the following descriptions, the ARM7TDMI-S processor samples **DBGTDI** and **DBGTMS** on the rising edge of **CLK** with **DBGTCKEN** HIGH. The TAP controller states are shown in Figure 5-8 on page 5-26.

5.13.1 SCAN_N (0010)

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

- In the CAPTURE-DR state, the fixed value 1000 is loaded into the register.
- In the SHIFT-DR state, the ID number of the desired scan path is shifted into the scan path select register.
- In the UPDATE-DR state, the scan register of the selected scan chain is connected between **DBGTDI** and **DBGTDO**, and remains connected until a subsequent SCAN_N instruction is issued.
- On reset, scan chain 0 is selected by default.

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

5.13.2 INTEST (1100)

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

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

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

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

5.13.3 IDCODE (1110)

The IDCODE instruction connects the device identification code register (or ID register) between **DBGTDI** and **DBGTDO**. The ID register is a 32-bit register that enables the manufacturer, part number, and version of a component to be read through the TAP. See *ARM7TDMI-S device identification (ID) code register* on page 5-31 for the details of the ID register format.

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

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

5.13.4 BYPASS (1111)

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

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

- In the CAPTURE-DR state, a logic 0 is captured the bypass register.
- In the SHIFT-DR state, test data is shifted into the bypass register through **DBGTDI** and shifted out on **DBGTDO** after a delay of one **CLK** cycle. The first bit to shift out is a zero.

- The bypass register is not affected in the UPDATE-DR state.

All unused instruction codes default to the BYPASS instruction.

5.13.5 RESTART (0100)

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

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

For more information, see *Exit from debug state* on page 5-42.

5.14 Test data registers

The six test data registers that can connect between **DBGTDI** and **DBGTDO** are described in the following sections:

- *Bypass register*
- *ARM7TDMI-S device identification (ID) code register*
- *Instruction register* on page 5-32
- *Scan path select register* on page 5-32
- *Scan chain 1* on page 5-34
- *Scan chain 2* on page 5-34.

In the following descriptions, data is shifted during every **CLK** cycle when **DBGTCKEN** enable is HIGH.

5.14.1 Bypass register

Purpose Bypasses the device during scan testing by providing a path between **DBGTDI** and **DBGTDO**.

Length 1 bit.

Operating mode When the BYPASS instruction is the current instruction in the instruction register, serial data is transferred from **DBGTDI** to **DBGTDO** in the SHIFT-DR state with a delay of one **CLK** cycle enabled by **DBGTCKEN**.
There is no parallel output from the bypass register.
A logic 0 is loaded from the parallel input of the bypass register in the CAPTURE-DR state.

5.14.2 ARM7TDMI-S device identification (ID) code register

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

Length 32 bits. The format of the ID code register is as shown in Figure 5-9.

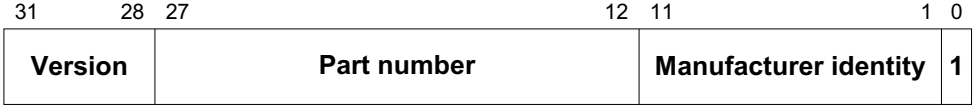


Figure 5-9 ID code register format

The default device identification code is `0x7f1f0f0f`.

Operating mode When the IDCODE instruction is current, the ID register is selected as the serial path between **DBGTDI** and **DBGTDO**.

There is no parallel output from the ID register.

The 32-bit device identification code is loaded into the ID register from its parallel inputs during the CAPTURE-DR state.

5.14.3 Instruction register

Purpose Changes the current TAP instruction.

Length 4 bits.

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

During the CAPTURE-IR state, the binary value 0001 is loaded into this register. This value is shifted out during SHIFT-IR (least significant bit first), while a new instruction is shifted in (least significant bit first).

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

On reset, IDCODE becomes the current instruction.

There is no parity bit.

5.14.4 Scan path select register

Purpose Changes the current active scan chain.

Length 4 bits.

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

During the CAPTURE-DR state, the value 1000 binary is loaded into this register. This value is loaded out during SHIFT-DR (least significant bit first), while a new value is loaded in (least significant bit first). During the UPDATE-DR state, the value in the register selects a scan chain to become the currently active scan chain. All additional instructions, such as INTEST, then apply to that scan chain.

The currently-selected scan chain changes only when a SCAN_N instruction is executed, or when a reset occurs. On reset, scan chain 0 is selected as the active scan chain.

Table 5-4 shows the scan chain number allocation.

Table 5-4 Scan chain number allocation

Scan chain number	Function
0	Reserved[a]
1	Debug
2	EmbeddedICE-RT programming
3	Reserved[a]
4	Reserved[a]
8	Reserved[a]

a. When selected, all reserved scan chains scan out zeros.

5.14.5 Scan chains 1 and 2

The scan chains allow serial access to the core logic, and to the EmbeddedICE-RT hardware for programming purposes. Each scan chain cell is simple and comprises a serial register and a multiplexor.

The scan cells perform three basic functions:

- capture
- shift
- update.

For input cells, the capture stage involves copying the value of the system input to the core into the serial register. During shift, this value is output serially. The value applied to the core from an input cell is either the system input, or the contents of the parallel register (loads from the shift register after UPDATE-DR state) under multiplexor control.

For output cells, capture involves placing the value of a core output into the serial register. During shift, this value is serially output as before. The value applied to the system from an output cell is either the core output, or the contents of the serial register.

All the control signals for the scan cells are generated internally by the TAP controller. The action of the TAP controller is determined by current instruction and the state of the TAP state machine.

Scan chain 1

Purpose Scan chain 1 is used for communication between the debugger, and the ARM7TDMI-S core. It is used to read and write data, and to scan instructions into the pipeline. The SCAN_N TAP instruction can be used to select scan chain 1.

Length 33 bits, 32 bits a for the data value and 1 bit for the scan cell on the **DBGBREAK** core input.

Scan chain order From **DBGTDI** to **DBGTDO**, the ARM7TDMI-S processor data bits, bits 0 to 31, then the 33rd bit, the **DBGBREAK** scan cell.

Scan chain 1, bit 33 serves three purposes:

- Under normal INTEST test conditions, it enables a known value to be scanned into the **DBGBREAK** input.
- While debugging, the value placed in the 33rd bit determines whether the ARM7TDMI-S core synchronizes back to system speed before executing the instruction. See *System speed access* on page 5-46 for more details.
- After the ARM7TDMI-S core has entered debug state, the value of the 33rd bit on the first occasion that it is captured, and scanned out tells the debugger whether the core entered debug state from a breakpoint (bit 33 LOW), or from a watchpoint (bit 33 HIGH).

Scan chain 2

Purpose Scan chain 2 provides access to the EmbeddedICE-RT registers. To do this, scan chain 2 must be selected using the SCAN_N TAP controller instruction, and then the TAP controller must be put in INTEST mode.

Length 38 bits.

Scan chain order From **DBGTDI** to **DBGTDO**, the read/write bit, the register address bits, bits 4 to 0, then the data bits, bits 0 to 31.

No action occurs during CAPTURE-DR.

During SHIFT-DR, a data value is shifted into the serial register. Bits 32 to 36 specify the address of the EmbeddedICE-RT register to be accessed.

During UPDATE-DR, this register is either read or written depending on the value of bit 37 (0 = read, 1 = write). See Figure 5-12 on page 5-49 for more details.

5.15 Scan timing

Figure 5-10 provides general scan timing information.

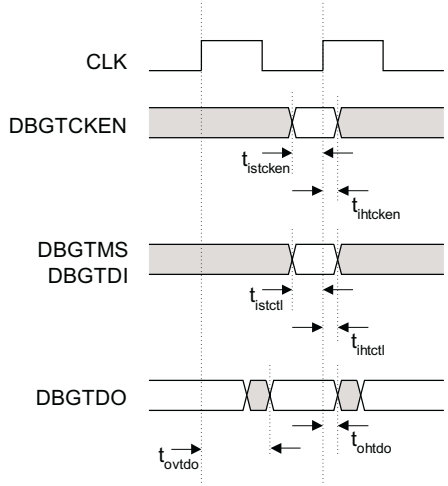


Figure 5-10 Scan timing

5.15.1 Scan chain 1 cells

The ARM7TDMI-S processor provides data for scan chain 1 cells as shown in Table 5-5.

Table 5-5 Scan chain 1 cells

Number	Signal	Type
1	**DATA[0]**	Input/output
2	**DATA[1]**	Input/output
3	**DATA[2]**	Input/output
4	**DATA[3]**	Input/output
5	**DATA[4]**	Input/output
6	**DATA[5]**	Input/output
7	**DATA[6]**	Input/output

Table 5-5 Scan chain 1 cells (continued)

Number	Signal	Type
8	**DATA[7]**	Input/output
9	**DATA[8]**	Input/output
10	**DATA[9]**	Input/output
11	**DATA[10]**	Input/output
12	**DATA[11]**	Input/output
13	**DATA[12]**	Input/output
14	**DATA[13]**	Input/output
15	**DATA[14]**	Input/output
16	**DATA[15]**	Input/output
17	**DATA[16]**	Input/output
18	**DATA[17]**	Input/output
19	**DATA[18]**	Input/output
20	**DATA[19]**	Input/output
21	**DATA[20]**	Input/output
22	**DATA[21]**	Input/output
23	**DATA[22]**	Input/output
24	**DATA[23]**	Input/output
25	**DATA[24]**	Input/output
26	**DATA[25]**	Input/output
27	**DATA[26]**	Input/output
28	**DATA[27]**	Input/output
29	**DATA[28]**	Input/output
30	**DATA[29]**	Input/output

Table 5-5 Scan chain 1 cells (continued)

Number	Signal	Type
31	**DATA[30]**	Input/output
32	**DATA[31]**	Input/output
33	**DBGBREAK**	Input

5.16 Examining the core and the system in debug state

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

Before you can examine the core and system state, the debugger must determine whether the processor entered debug state from Thumb state or ARM state, by examining bit 4 of the EmbeddedICE-RT debug status register, as follows:

Bit 4 HIGH The core has entered debug from Thumb state.

Bit 4 LOW The core has entered debug from ARM state.

5.16.1 Determining the core state

When the processor has entered debug state from Thumb state, the simplest course of action is for the debugger to force the core back into ARM state. The debugger can then execute the same sequence of instructions to determine the processor state.

To force the processor into ARM state, execute the following sequence of Thumb instructions on the core:

```
STR R0, [R0]; Save R0 before use
MOV R0, PC  ; Copy PC into R0
STR R0, [R0]; Now save the PC in R0
BX  PC      ; Jump into ARM state
MOV R8, R8  ; NOP
MOV R8, R8  ; NOP
```

Note

Because all Thumb instructions are only 16 bits long, you can repeat the instruction when shifting scan chain 1. For example, the encoding for BX R0 is `0x4700`, so when `0x47004700` shifts into scan chain 1, the debugger does not have to keep track of the half of the bus on which the processor expects to read the data.

You can use the sequences of ARM instructions below to determine the state of the processor.

With the processor in the ARM state, the first instruction to execute is typically:

```
STM R0, {R0-R15}
```

This instruction causes the contents of the registers to appear on the data bus. You can then sample and shift out these values.

Note

The use of r0 as the base register for the STM is only for illustration, any register can be used.

After you have determined the values in the current bank of registers, you might wish to access the banked registers. To do this, you must change mode. Normally, a mode change can occur only if the core is already in a privileged mode. However, while in debug state, a mode change from one mode into any other mode can occur.

The debugger must restore the original mode before exiting debug state. For example, if the debugger was requested to return the state of the User mode registers, and FIQ mode registers, and debug state was entered in Supervisor mode, the instruction sequence might be:

```
STM R0, {R0-R15}; Save current registers
MRS R0, CPSR
STR R0, R0; Save CPSR to determine current mode
BIC R0, 0x1F; Clear mode bits
ORR R0, 0x10; Select user mode
MSR CPSR, R0; Enter USER mode
STM R0, {R13,R14}; Save register not previously visible
ORR R0, 0x01; Select FIQ mode
MSR CPSR, R0; Enter FIQ mode
STM R0, {R8-R14}; Save banked FIQ registers
```

All these instructions execute at debug speed. Debug speed is much slower than system speed. This is because between each core clock, 33 clocks occur in order to shift in an instruction, or shift out data. Executing instructions this slowly is acceptable for accessing the core state because the ARM7TDMI-S processor is fully static. However, you cannot use this method for determining the state of the rest of the system.

While in debug state, 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.

5.16.2 Determining system state

To meet the dynamic timing requirements of the memory system, any attempt to access system state must occur with the clock qualified by **CLKEN**. To perform a memory access, **CLKEN** must be used to force the ARM7TDMI-S processor to run in normal operating mode. This is controlled by bit 33 of scan chain 1.

An instruction placed in scan chain 1 with bit 33, the **DBGBREAK** bit, LOW executes at debug speed. To execute an instruction at system speed, the instruction prior to it must be scanned into scan chain 1 with bit 33 set HIGH.

After the system speed instruction has scanned into the data bus and clocked into the pipeline, the RESTART instruction must be loaded into the TAP controller. RESTART causes the ARM7TDMI-S processor to:

1. Switch automatically to **CLKEN** control.
2. Execute the instruction at system speed.
3. Reenter debug state.

When the instruction has completed, **DBGACK** is HIGH and the core reverts to **DBGTCKEN** control. It is now possible to select INTEST in the TAP controller and resume debugging.

The debugger must look at both **DBGACK** and **TRANS[1:0]** to determine whether a system speed instruction has completed. To access memory, the ARM7TDMI-S core drives both bits of **TRANS[1:0]** LOW after it has synchronized back to system speed. This transition is used by the memory controller to arbitrate whether the ARM7TDMI-S core can have the bus in the next cycle. If the bus is not available, the ARM7TDMI-S processor might have its clock stalled indefinitely. The only way to determine whether the memory access has completed is to examine the state of both **TRANS[1:0]** and **DBGACK**. When both are HIGH, the access has completed.

The debugger usually uses EmbeddedICE-RT to control debugging, and so the state of **TRANS[1:0]** and **DBGACK** can be determined by reading the EmbeddedICE-RT status register. See *Debug status register* on page 5-60 for more details.

The state of the system memory can be fed back to the debug host by using system speed load multiples and debug speed store multiples.

There are restrictions on which instructions can have bit 33 set. The valid instructions on which to set this bit are:

- loads
- stores
- load multiple
- store multiple.

See also *Exit from debug state* on page 5-42.

When the ARM7TDMI-S processor returns to debug state after a system speed access, bit 33 of scan chain 1 is set HIGH. The state of bit 33 gives the debugger information about why the core entered debug state the first time this scan chain is read.

5.17 Exit from debug state

Leaving debug state involves:

- restoring the ARM7TDMI-S processor internal state
- causing the execution of a branch to the next instruction
- returning to normal operation.

After restoring the internal state, a branch instruction must be loaded into the pipeline. See *The program counter during debug* on page 5-44 for details on calculating the branch.

Bit 33 of scan chain 1 forces the ARM7TDMI-S processor to resynchronize back to **CLKEN**, clock enable. The penultimate instruction of the debug sequence is scanned in with bit 33 set HIGH. The final instruction of the debug sequence is the branch, which is scanned in with bit 33 LOW. The core is then clocked to load the branch instruction into the pipeline, and the RESTART instruction is selected in the TAP controller.

When the state machine enters the RUN-TEST/IDLE state, the scan chain reverts back to System mode. The ARM7TDMI-S processor then resumes normal operation, fetching instructions from memory. This delay, until the state machine is in the RUN-TEST/IDLE state, enables conditions to be set up in other devices in a multiprocessor system without taking immediate effect. When the state machine enters the RUN-TEST/IDLE state, all the processors resume operation simultaneously.

DBGACK informs the rest of the system when the ARM7TDMI-S processor is in debug state. This information can be used to inhibit peripherals, such as watchdog timers, that have real-time characteristics. **DBGACK** can also mask out memory accesses caused by the debugging process.

For example, when the ARM7TDMI-S processor enters debug state after a breakpoint, the instruction pipeline contains the breakpointed instruction, and two other instructions that have been prefetched. On entry to debug state the pipeline is flushed. On exit from debug state the pipeline must therefore revert to its previous state.

Because of the debugging process, more memory accesses occur than are expected normally. **DBGACK** can inhibit any system peripheral that might be sensitive to the number of memory accesses. For example, a peripheral that counts the number of memory cycles must return the same answer after a program has been run with and without debugging. Figure 5-11 on page 5-43 shows the behavior of the ARM7TDMI-S processor on exit from the debug state.

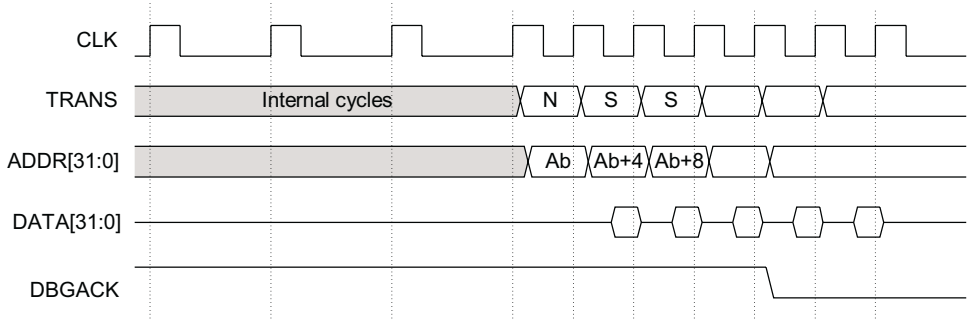


Figure 5-11 Debug exit sequence

Figure 5-3 on page 5-8 shows that the final memory access occurs in the cycle after **DBGACK** goes HIGH. This is the point at which the cycle counter must be disabled. Figure 5-11 shows that the first memory access that the cycle counter has not previously seen occurs in the cycle after **DBGACK** goes LOW. This is the point at which to re-enable the counter.

Note

When a system speed access from debug state occurs, the ARM7TDMI-S processor temporarily drops out of debug state, so **DBGACK** can go LOW. If there are peripherals that are sensitive to the number of memory accesses, they must be led to believe that the ARM7TDMI-S processor is still in debug state. You can do this by programming the EmbeddedICE-RT control register to force the value on **DBGACK** to be HIGH. See *Debug status register* on page 5-60 for more details.

5.18 The program counter during debug

The debugger must keep track of what happens to the PC, so that the ARM7TDMI-S core can be forced to branch back to the place at which program flow was interrupted by debug. Program flow can be interrupted by any of the following:

- *Breakpoints*
- *Watchpoints*
- *Watchpoint with another exception* on page 5-45
- *Debug request* on page 5-45
- *System speed access* on page 5-46.

5.18.1 Breakpoints

Entry into debug state from a breakpoint advances the PC by four addresses or 16 bytes. Each instruction executed in debug state advances the PC by one address or 4 bytes.

The usual way to exit from debug state after a breakpoint is to remove the breakpoint and branch back to the previously-breakpointed address.

For example, if the ARM7TDMI-S processor entered debug state from a breakpoint set on a given address, and two debug speed instructions were executed, a branch of –7 addresses must occur (4 for debug entry, plus 2 for the instructions, plus 1 for the final branch).

The following sequence shows the data scanned into scan chain 1, most significant bit first. The value of the first digit goes to the **DBGBREAK** bit, and then the instruction data into the remainder of scan chain 1:

```
0 E0802000; ADD R2, R0, R0
1 E1826001; ORR R6, R2, R1
0 EAFFFFF9; B -7 (2's complement)
```

After the ARM7TDMI-S processor enters debug state, it must execute a minimum of two instructions before the branch, although these can both be NOPs (`MOV R0, R0`). For small branches, you can replace the final branch with a subtract, with the PC as the destination (`SUB PC, PC, #28` in the above example).

5.18.2 Watchpoints

The return to program execution after entry to debug state from a watchpoint is made in the same way as the procedure described in *Breakpoints*.

Debug entry adds four addresses to the PC, and every instruction adds one address. The difference from breakpoint is that the instruction that caused the watchpoint has executed, and the program must return to the next instruction.

5.18.3 Watchpoint with another exception

If a watchpointed access simultaneously causes a Data Abort, the ARM7TDMI-S processor enters debug state in abort mode. Entry into debug is held off until the core changes into abort mode and has fetched the instruction from the abort vector.

A similar sequence follows when an interrupt, or any other exception, occurs during a watchpointed memory access. The ARM7TDMI-S processor enters debug state in the mode of the exception. The debugger must check to see whether an exception has occurred by examining the current and previous mode (in the CPSR, and SPSR), and the value of the PC. When an exception has taken place, you are given the choice of servicing the exception before debugging.

Entry to debug state when an exception has occurred causes the PC to be incremented by three instructions rather than four, and this must be considered in return branch calculation when exiting debug state. For example, suppose that an abort occurs on a watchpointed access, and ten instructions have been executed to determine this eventuality. You can use the following sequence to return to program execution.

```
0 E1A00000; MOV R0, R0
1 E1A00000; MOV R0, R0
0 EAFFFFF0; B -16
```

This code forces a branch back to the abort vector, causing the instruction at that location to be refetched and executed.

Note

After the abort service routine, the instruction that caused the abort, and watchpoint is refetched and executed. This triggers the watchpoint again and the ARM7TDMI-S processor reenters debug state.

5.18.4 Debug request

Entry into debug state using a debug request is similar to a breakpoint. However, unlike a breakpoint, the last instruction has completed execution and so must not be refetched on exit from debug state. Therefore, you can assume that entry to debug state adds three addresses to the PC and every instruction executed in debug state adds one address.

For example, suppose you have invoked a debug request, and decide to return to program execution straight away. You could use the following sequence:

```
0 E1A00000; MOV R0, R0
1 E1A00000; MOV R0, R0
0 EAFFFFFA; B -6
```

This code restores the PC and restarts the program from the next instruction.

5.18.5 System speed access

When a system speed access is performed during debug state, the value of the PC increases by three addresses. System speed instructions access the memory system and so it is possible for aborts to take place. If an abort occurs during a system speed memory access, the ARM7TDMI-S processor enters abort mode before returning to debug state.

This scenario is similar to an aborted watchpoint, but the problem is much harder to fix because the abort was not caused by an instruction in the main program, and so the PC does not point to the instruction that caused the abort. An abort handler usually looks at the PC to determine the instruction that caused the abort and also the abort address. In this case, the value of the PC is invalid, but because the debugger can determine which location was being accessed, the debugger can be written to help the abort handler fix the memory system.

5.18.6 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.

5.19 Priorities and exceptions

When a breakpoint, or a debug request occurs, the normal flow of the program is interrupted. Therefore, debug can be treated as another type of exception. The interaction of the debugger with other exceptions is described in *The program counter during debug* on page 5-44. This section covers the following priorities:

- *Breakpoint with Prefetch Abort*
- *Interrupts*
- *Data Aborts*.

5.19.1 Breakpoint with Prefetch Abort

When a breakpointed instruction fetch causes a Prefetch Abort, the abort is taken, and the breakpoint is disregarded. Normally, Prefetch Aborts occur when, for example, an access is made to a virtual address that does not physically exist, and the returned data is therefore invalid. In such a case, the normal action of the operating system is to swap in the page of memory, and to return to the previously-invalid address. This time, when the instruction is fetched, and providing the breakpoint is activated (it can be data-dependent), the ARM7TDMI-S processor enters debug state.

The Prefetch Abort, therefore, takes higher priority than the breakpoint.

5.19.2 Interrupts

When the ARM7TDMI-S processor enters debug state, interrupts are automatically disabled.

If an interrupt is pending during the instruction prior to entering debug state, the ARM7TDMI-S processor enters debug state in the mode of the interrupt. On entry to debug state, the debugger cannot assume that the ARM7TDMI-S processor is in the mode expected by the program of the user. The ARM7TDMI-S core must check the PC, the CPSR, and the SPSR to determine accurately the reason for the exception.

Debug, therefore, takes higher priority than the interrupt, but the ARM7TDMI-S processor does remember that an interrupt has occurred.

5.19.3 Data Aborts

When a Data Abort occurs on a watchpointed access, the ARM7TDMI-S processor enters debug state in abort mode. The watchpoint, therefore, has higher priority than the abort, but the ARM7TDMI-S processor remembers that the abort happened.

5.20 Watchpoint unit 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 5-1 on page 5-17.

5.20.1 Programming and reading watchpoint registers

A watchpoint register is programmed by shifting data into the EmbeddedICE-RT scan chain (scan chain 2). The scan chain is a 38-bit shift register comprising:

- a 32-bit data field
- a 5-bit address field
- a read/write bit.

This setup is shown in Figure 5-12 on page 5-49.

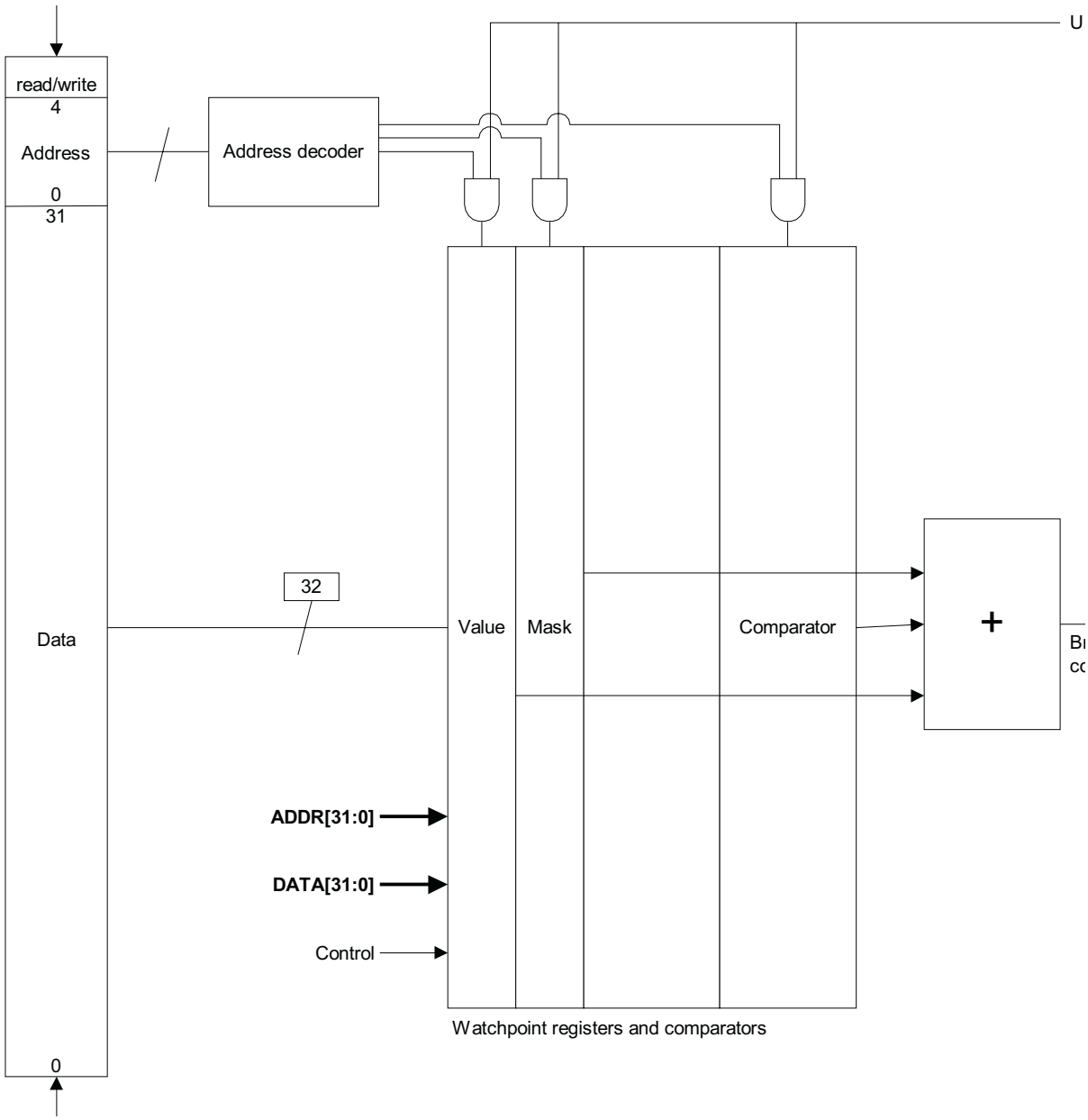


Figure 5-12 EmbeddedICE-RT block diagram

The data to be written is shifted into the 32-bit data field, the address of the register is shifted into the 5-bit address field, and the read/write bit is set.

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 5-1 on page 5-17.

Note

A read or write actually takes place when the TAP controller enters the UPDATE-DR state.

5.20.2 Using the data, and address mask registers

For each value register in a register pair, there is a mask register of the same format. Setting a bit to 1 in the mask register has the effect of making the corresponding bit in the value register disregarded in the comparison.

For example, when a watchpoint is required on a particular memory location, but the data value is irrelevant, the data mask register can be programmed to `0xffffffff` (all bits set to 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.

5.20.3 The control registers

The control value and control mask registers are mapped identically in the lower eight bits, as shown in Figure 5-13.

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	DBGEXT	PROT[1]	PROT[0]	SIZE[1]	SIZE[0]	WRITE

Figure 5-13 Watchpoint control value, and mask format

Bit 8 of the control value register is the ENABLE bit and cannot be masked.

The bits have the following functions:

WRITE Compares against the write signal from the core in order to detect the direction of bus activity. **WRITE** is 0 for a read cycle, and 1 for a write cycle.

SIZE[1:0] Compares against the **SIZE[1:0]** signal from the core in order to detect the size of bus activity.

The encoding is shown in Table 5-6.

Table 5-6 SIZE[1:0] signal encoding

bit 1	bit 0	Data size
0	0	Byte
0	1	Halfword
1	0	Word
1	1	(Reserved)

PROT[0] Is used to detect whether the current cycle is an instruction fetch (**PROT[0]** = 0), or a data access (**PROT[0]** = 1).

PROT[1] Is used to compare against the not translate signal from the core in order to distinguish between user mode (**PROT[1]** = 0), and non-User mode (**PROT[1]** = 1) accesses.

DBGEXT[1:0] Is an external input to EmbeddedICE-RT logic that enables the watchpoint to be dependent on some external condition.

The **DBGEXT** input for Watchpoint 0 is labeled **DBGEXT[0]**.

The **DBGEXT** input for Watchpoint 1 is labeled **DBGEXT[1]**.

CHAIN Can be connected to the chain output of another watchpoint in order to implement, for example, debugger requests of the form `breakpoint on address YYY only when in process XXX`.

In the ARM7TDMI-S processor EmbeddedICE-RT macrocell, the **CHAINOUT** output of Watchpoint 1 is connected to the **CHAIN** input of Watchpoint 0.

The **CHAINOUT** output is derived from a register. The address/control field comparator drives the write enable for the register. The input to the register is the value of the data field comparator.

The **CHAINOUT** register is cleared when the control value register is written, or when **DBGnTRST** is LOW.

RANGE In the ARM7TDMI-S processor EmbeddedICE-RT logic, the **RANGEOUT** output of Watchpoint 1 is connected to the **RANGE** input of Watchpoint 0. Connection enables the two watchpoints to be coupled for detecting conditions that occur simultaneously, such as for range checking.

ENABLE When a watchpoint match occurs, the internal **DBGBREAK** signal is asserted only when the **ENABLE** bit is set. This bit exists only in the value register. It cannot be masked.

For each of the bits [7:0] in the control value register, there is a corresponding bit in the control mask register. These bits remove the dependency on particular signals.

5.21 Programming breakpoints

Breakpoints are classified as hardware breakpoints or software breakpoints:

- *Hardware breakpoints* typically monitor the address value and can be set in any code, even in code that is in ROM or code that is self-modifying. See *Hardware breakpoints* for more details.
- *Software breakpoints* monitor a particular bit pattern being fetched from any address. One EmbeddedICE-RT watchpoint can therefore be used to support any number of software breakpoints. See *Software breakpoints* on page 5-54 for more details.

  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.

5.21.1 Hardware breakpoints

To make a watchpoint unit cause hardware breakpoints (on instruction fetches):

1. Program its address value register with the address of the instruction to be breakpointed.
2. 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 have to match the actual instruction code fetched as well as the address. If the data value is not required, program the data mask register to `0xffffffff` (all bits to 1). Otherwise program it to `0x00000000`.
4. Program the control value register with **PROT[0]** = 0.
5. Program the control mask register with **PROT[0]**= 0.
6. When you have to make the distinction between User and non-User mode instruction fetches, program the **PROT[1]** value and mask bits appropriately.
7. If required, program the **DBGEXT**, **RANGE**, and **CHAIN** bits in the same way.
8. Program the mask bits for all unused control values to 1.

5.21.2 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 `0xdfff`, program `0xdfffdfff`. When a 16-bit instruction is fetched, EmbeddedICE-RT compares only the valid half of the data bus against the contents of the data value register. In this way, you can use a single watchpoint register to catch software breakpoints on both the upper and lower halves of the data bus.
3. Program the data mask register to `0x00000000`.
4. Program the control value register with **PROT[0]** = 0.
5. Program the control mask register with **PROT[0]** = 0 and all other bits to 1.
6. If you want to make the distinction between User and non-User mode instruction fetches, program the **PROT[1]** bit in the control value, and control mask registers accordingly.
7. If required, program the **DBGEXT**, **RANGE**, and **CHAIN** bits in the same way.

Note

You do not have to program the address value register.

Setting the breakpoint

To set the software breakpoint:

1. Read the instruction at the desired address and store it.
2. Write the special bit pattern representing a software breakpoint at the address.

Clearing the breakpoint

To clear the software breakpoint, restore the instruction to the address.

5.22 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 have to match the actual data value read or written as well as the address. If the data value is irrelevant, program the data mask register to `0xffffffff` (all bits set to 1). Otherwise program the data mask register to `0x00000000`.
4. Program the control value register with **PROT[0]**= 1, **WRITE**= 0 for a read, or **WRITE** = 1 for a write, **SIZE[1:0]** with the value corresponding to the appropriate data size.
5. Program the control mask register with **PROT[0]** = 0, **WRITE** = 0, **SIZE[1:0]**= 0, and all other bits to 1. You can set **WRITE**, or **SIZE[1:0]**to 1 when both reads and writes, or data size accesses are to be watchpointed respectively.
6. If you have to make the distinction between User and non-User mode data accesses, program the **PROT[1]** bit in the control value and control mask registers accordingly.
7. If required, program the **DBGEXT**, **RANGE**, and **CHAIN** bits in the same way.

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 example, you can provide simple range breakpoints by setting one or more of the address mask bits.

5.23 Abort status register

Only bit 0 of this 32 bit read/write register is used. It determines whether an abort exception entry was caused by a breakpoint, a watchpoint, or a real abort. The format is shown in Figure 5-14.

31:1	0
SBZ/RAZ	**DbgAbt**

Figure 5-14 Debug abort status register

This bit is set when the ARM7TDMI-S core takes a Prefetch or Data Abort as a result of a breakpoint or watchpoint. If, on a particular instruction or data fetch, both the Debug Abort and the external Abort signal are asserted, the external Abort takes priority, and the DbgAbt bit is not set. Once set, DbgAbt remains set until reset by the user. The register is accessed by `MRC` and `MCR` instructions.

5.24 Debug control register

The debug control register is six bits wide. Writes to the debug control register occur when a watchpoint unit register is written. Reads of the debug control register occur when a watchpoint unit register is read. See *Watchpoint unit registers* on page 5-48 for more information.

Figure 5-15 shows the function of each bit in the debug control register.

5	4	3	2	1	0
EmbeddedICE-RT disable	Monitor mode enable	SBZ/RAZ	**INTDIS**	**DBGRQ**	**DBGACK**

Figure 5-15 Debug control register format

The debug control register bit assignments are shown in Table 5-7.

Table 5-7 Debug control register bit assignments

Bit	Function
5	Used to disable the EmbeddedICE-RT comparator outputs while the watchpoint and breakpoint registers are being programmed. This bit can be read and written through JTAG. Set bit 5 when: • programming breakpoint or watchpoint registers • changing bit 4 of the debug control register. You must clear bit 5 after you have made the changes, to re-enable the EmbeddedICE-RT logic and make the new breakpoints and watchpoints operational.
4	Used to determine the behavior of the core when breakpoints or watchpoints are reached: • If clear, the core enters debug state when a breakpoint or watchpoint is reached. • If set, the core performs an abort exception when a breakpoint or watchpoint is reached. This bit can be read and written from JTAG.
3	This bit must be clear.

Table 5-7 Debug control register bit assignments (continued)

Bit	Function
2	Used to disable interrupts: • If set, the interrupt enable signal of the core (**IFEN**) is forced LOW. The **IFEN** signal is driven as shown in Table 5-8. • If clear, interrupts are enabled.
1	Used to force the value on **DBGRQ**.
0	Used to force the value on **DBGACK**.

5.24.1 Disabling interrupts

IRQs and FIQs are disabled under the following conditions:

- during debugging (**DBGACK** HIGH)
- when the **INTDIS** bit is HIGH.

The **IFEN** signal is driven as shown in Table 5-8.

Table 5-8 Interrupt signal control

DBGACK	INTDIS	IFEN	Interrupts
0	0	1	Permitted
1	x	0	Inhibited
x	1	0	Inhibited

5.24.2 Forcing DBGRQ

Figure 5-17 on page 5-61 shows that the value stored in bit 1 of the debug control register is synchronized and then ORed with the external **DBGRQ** before being applied to the processor. The output of this OR gate is the signal **DBGRQI** which is brought out externally from the macrocell.

The synchronization between debug control register bit 1 and **DBGRQI** assists in multiprocessor environments. The synchronization latch only opens when the TAP controller state machine is in the RUN-TEST-IDLE state. This enables an enter-debug condition to be set up in all the processors in the system while they are still running. When the condition is set up in all the processors, it can be applied to them simultaneously by entering the RUN-TEST-IDLE state.

5.24.3 Forcing DBGACK

Figure 5-17 on page 5-61 shows that the value of the internal signal **DBGACKI** from the core is ORed with the value held in bit 0 of the debug control register, to generate the external value of **DBGACK** seen at the periphery of the ARM7TDMI-S core. This enables the debug system to signal to the rest of the system that the core is still being debugged even when system-speed accesses are being performed (when the internal **DBGACK** signal from the core is LOW).

5.25 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 5-16.

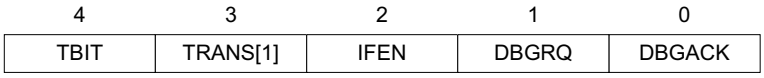


Figure 5-16 Debug status register format

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 state of the **TRANS[1]** signal from the core to be read. This enables the debugger to determine whether a memory access from the debug state has completed.

Bit 2 Enables the state of the core interrupt enable signal (**IFEN**) to be read.

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 5-17 on page 5-61.

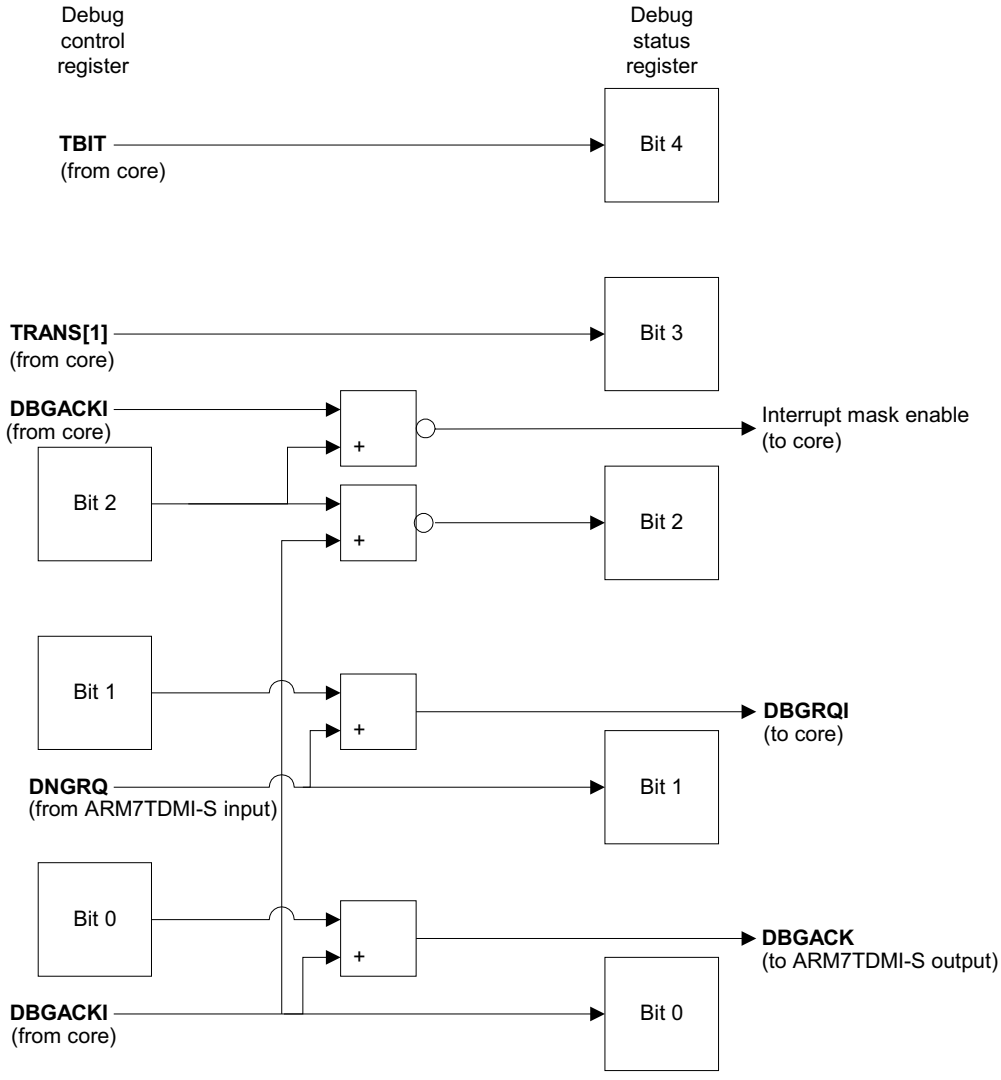


Figure 5-17 Debug control and status register structure

5.26 Coupling breakpoints and watchpoints

You can couple watchpoint units 1 and 0 together using the **CHAIN** and **RANGE** inputs. The use of **CHAIN** enables Watchpoint 0 to be triggered only if Watchpoint 1 has previously matched. The use of **RANGE** enables simple range checking to be performed by combining the outputs of both watchpoints.

5.26.1 Breakpoint and watchpoint coupling example

Let:

`Av[31:0]` Be the value in the address value register

`Am[31:0]` Be the value in the address mask register

`A[31:0]` Be the address bus from the ARM7TDMI-S processor

`Dv[31:0]` Be the value in the data value register

`Dm[31:0]` Be the value in the data mask register

`D[31:0]` Be the data bus from the ARM7TDMI-S processor

`Cv[8:0]` Be the value in the control value register

`Cm[7:0]` Be the value in the control mask register

`C[9:0]` Be the combined control bus from the ARM7TDMI-S core, other watchpoint registers, and the **DBGEXT** signal.

CHAINOUT signal

The **CHAINOUT** signal is derived as follows:

```
WHEN (({Av[31:0],Cv[4:0]} XNOR {A[31:0],C[4:0]}) OR {Am[31:0],Cm[4:0]} == 0xFFFFFFFFF)
CHAINOUT = ((({Dv[31:0],Cv[6:4]} XNOR {D[31:0],C[7:5]}) OR {Dm[31:0],Cm[7:5]}) == 0x7FFFFFFF)
```

The **CHAINOUT** output of watchpoint register 1 provides the **CHAIN** input to Watchpoint 0. This **CHAIN** input enables you to use quite complicated configurations of breakpoints and watchpoints.

Note

There is no **CHAIN** input to Watchpoint 1 and no **CHAIN** output from Watchpoint 0.

Take, for example, the request by a debugger to breakpoint on the instruction at location YYY when running process XXX in a multiprocess system. If the current process ID is stored in memory, you can implement the above function with a watchpoint and

breakpoint chained together. The watchpoint address points to a known memory location containing the current process ID, the watchpoint data points to the required process ID and the **ENABLE** bit is cleared.

The address comparator output of the watchpoint is used to drive the write enable for the **CHAINOUT** latch. The input to the latch is the output of the data comparator from the same watchpoint. The output of the latch drives the **CHAIN** input of the breakpoint comparator. The address YYY is stored in the breakpoint register, and when the **CHAIN** input is asserted, the breakpoint address matches and the breakpoint triggers correctly.

5.26.2 DBGRNG signal

The **DBGRNG** signal is derived as follows:

```
DBGRNG = ((({Av[31:0],Cv[4:0]} XNOR {A[31:0],C[4:0]}) OR {Am[31:0],Cm[4:0]}) == 0xFFFFFFFFF) AND
((({Dv[31:0],Cv[7:5]} XNOR {D[31:0],C[7:5]}) OR
Dm[31:0],Cm[7:5]}) == 0x7FFFFFFF)
```

The **DBGRNG** output of watchpoint register 1 provides the **RANGE** input to watchpoint register 0. This **RANGE** input enables you to couple two breakpoints together to form range breakpoints.

Selectable ranges are restricted to being powers of 2. For example, if a breakpoint is to occur when the address is in the first 256 bytes of memory, but not in the first 32 bytes, program the watchpoint registers as follows:

For Watchpoint 1:

1. Program Watchpoint 1 with an address value of `0x00000000` and an address mask of `0x0000001f`.
2. Clear the **ENABLE** bit.
3. Program all other Watchpoint 1 registers as normal for a breakpoint.

   An address within the first 32 bytes causes the **RANGE** output to go HIGH but does not trigger the breakpoint.

For Watchpoint 0:

1. Program Watchpoint 0 with an address value of `0x00000000`, and an address mask of `0x000000ff`.
2. Set the ENABLE bit.
3. Program the RANGE bit to match a 0.
4. Program all other Watchpoint 0 registers as normal for a breakpoint.

If Watchpoint 0 matches but Watchpoint 1 does not (that is the **RANGE** input to Watchpoint 0 is 0), the breakpoint is triggered.

5.27 EmbeddedICE-RT timing

EmbeddedICE-RT samples the **DBGEXT[1]** and **DBGEXT[0]** inputs on the rising edge of **CLK**.

See Chapter 8 *AC Parameters* for details of the required setup and hold times for these signals.

Chapter 6
ETM Interface

This chapter describes the ETM interface that is provided on the ARM7TDMI-S processor. It contains the following sections:

6.1 About the ETM interface

You can connect an external *Embedded Trace Macrocell* (ETM) to the ARM7TDMI-S processor, so that you can perform real-time tracing of the code that the processor is executing.

Note

If you have more than one ARM processor in your system, each processor must have its own dedicated ETM.

In general, little or no glue logic is required to connect the ETM7 to the ARM7TDMI-S (Rev 4) processor. You program the ETM through a JTAG interface. The interface is an extension of the ARM TAP controller, and is assigned scan chain 6.

Note

See the *ETM7 (Rev 1) Technical Reference Manual* for detailed information about integrating an ETM7 with an ARM7TDMI-S processor.

6.2 Enabling and disabling the ETM7 interface

Under the control of the ARM debug tools, the ETM7 **PWRDOWN** output is used to enable and disable the ETM. When **PWRDOWN** is HIGH, this indicates that the ETM is not currently enabled, so you can stop the **CLK** input and hold the other ETM signals stable. This enables you to reduce power consumption when you are not performing tracing.

When a TAP reset (**DBGnTRST**) occurs, **PWRDOWN** is forced HIGH until the ETM7 control register has been programmed (see the *Embedded Trace Macrocell Specification* for details of this register).

PWRDOWN is automatically cleared at the start of a debug session.

6.3 ETM7 to ARM7TDMI-S (Rev 4) connections

The ETM7 interface port names are a mixture of those from the ARM7TDMI and the ARM7TDMI-S macrocells. Table 6-1 shows the connections that you must make between the ARM7TDMI-S processor and ETM7.

Table 6-1 ETM7 and ARM7TDMI-S (Rev 4) pin connections

ETM7 signal name	ARM7TDMI-S (Rev 4) signal name
A[31:0]	**ADDR[31:0]**
ABORT	**ABORT**
ARMTDO	**DBGTDO**
BIGEND	**CFGBIGEND**
CLK[a]	**CLK**[a]
CLKEN	**CLKEN**
CPA	**CPA**
CPB	**CPB**
DBGACK	**DBGACK**
DBGRQ[b]	**DBGRQ**[b]
nMREQ	**CPnMREQ**
SEQ	**CPSEQ**
MAS[1:0]	**SIZE[1:0]**
nCPI	**CPnI**
nEXEC	**DBGnEXEC**
nOPC	**CPnOPC**
nRESET	**nRESET**
nRW	**WRITE**
nTRST[a]	**DBGnTRST**[a]
PROCID[31:0][c]	-
PROCIDWR[c]	-

Table 6-1 ETM7 and ARM7TDMI-S (Rev 4) pin connections (continued)

ETM7 signal name	ARM7TDMI-S (Rev 4) signal name
RANGEOUT[0]	**DBGRNG[0]**
RANGEOUT[1]	**DBGRNG[1]**
RDATA[31:0]	**RDATA[31:0]**
TBIT	**CPTBIT**
TCK[a]	**CLK**[a]
TCKEN	**DBGTCKEN**
TDI	**DBGTDI**
TDO	**DBGTDO**
TMS	**DBGTMS**
WDATA[31:0]	**WDATA[31:0]**
INSTRVALID	**DBGINSTRVALID**

a. See *Clocks and resets* on page 6-6.
b. See *Debug request wiring* on page 6-7.
c. The ARM7TDMI-S processor does not provide the **PROCID[31:0]** or **PROCIDWR** signals. You must tie these ETM inputs LOW.

6.4 Clocks and resets

The ARM7TDMI-S (Rev 4) processor uses a single clock, **CLK**, as both the main system clock and the JTAG clock. You must connect the processor clock to both **CLK** and **TCK** on the ETM. You can then use **TCKEN** to control the JTAG interface.

To trace through a warm reset of the ARM7TDMI-S processor, use the TAP reset (connect **nTRST** to **DBGnTRST**) to reset the ETM7 state.

For more information about ETM7 clocks and resets, see the *ETM7 Technical Reference Manual*.

6.5 Debug request wiring

It is recommended that you connect together the **DBGRQ** output of the ETM7 to the **DBGRQ** input of the ARM7TDMI-S processor. If this input is already in use, you can OR the **DBGRQ** inputs together. See the *ETM7 Technical Reference Manual* for more details.

Chapter 7
Instruction Cycle Timings

This chapter gives the ARM7TDMI-S processor instruction cycle timings. It contains the following sections:

7.1 About the instruction cycle timings

The **TRANS[1:0]** signals predict the type of the next cycle. These signals are pipelined in the cycle before the one to which they apply and are shown like this in the tables in this section.

In the tables in this chapter, the following signals (which also appear ahead of the cycle) are registered in the cycle to which they apply:

- Address is **ADDR[31:0]**
- Lock is **LOCK**
- Size is **SIZE[1:0]**
- Write is **WRITE**
- Prot1 and Prot0 are **PROT[1:0]**
- Tbit is **CPTBIT**.

The address is incremented for prefetching instructions in most cases. The increment varies with the instruction length:

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

Note

The letter i is used to indicate the instruction lengths.

Size indicates the width of the transfer:

- w (word) represents a 32-bit data access or ARM opcode fetch
- h (halfword) represents a 16-bit data access or Thumb opcode fetch
- b (byte) represents an 8-bit data access.

CPA and **CPB** are pipelined inputs and are shown as sampled by the ARM7TDMI-S processor. They are therefore shown in the tables the cycle after they have been driven by the coprocessor.

Transaction types are shown in Table 7-1 on page 7-4.

Table 7-1 Transaction types

TRANS[1:0]	Transaction type	Description
00	I cycle	Internal (address-only) next cycle
01	C cycle	Coprocessor transfer next cycle
10	N cycle	Memory access to next address is nonsequential
11	S cycle	Memory access to next address is sequential

Note

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

7.2 Instruction cycle count summary

In the pipelined architecture of the ARM7TDMI-S core, while one instruction is being fetched, the previous instruction is being decoded, and the one prior to that is being executed. Table 7-2 shows the number of cycles required by an instruction, when that instruction reaches the Execute stage.

You can calculate the number of cycles for a routine from the figures in Table 7-2. These figures assume execution of the instruction. Unexecuted instructions take one cycle.

In Table 7-2:

n Is the number of words transferred.

m Is 1 if bits [32:8] of the multiplier operand are all zero or one.
Is 2 if bits [32:16] of the multiplier operand are all zero or one.
Is 3 if bits [31:24] of the multiplier operand are all zero or one.
Is 4 otherwise.

b Is the number of cycles spent in the coprocessor busy-wait loop (which can be zero or more).

When the condition is not met, all the instructions take one S-cycle.

Table 7-2 Instruction cycle counts

Instruction	Qualifier	Cycle count
Any unexecuted	Condition codes fail	+S
Data processing	Single-cycle	+S
Data processing	Register-specified shift	+I +S
Data processing	R15 destination	+N +2S
Data processing	R15, register-specified shift	+I +N +2S
`MUL`	-	+(m)I +S
`MLA`	-	+I +(m)I +S
`MULL`	-	+(m)I +I +S
`MLAL`	-	+I +(m)I +I +S
`B, BL`	-	+N +2S
`LDR`	Non-R15 destination	+N +I +S
`LDR`	R15 destination	+N +I +N +2S

Table 7-2 Instruction cycle counts (continued)

Instruction	Qualifier	Cycle count
STR	-	+N +N
SWP	-	+N +N +I +S
LDM	Non-R15 destination	+N +(n–1)S +I +S
LDM	R15 destination	+N +(n–1)S +I +N +2S
STM	-	+N +(n–1)S +I +N
MSR, MRS	-	+S
SWI, trap	-	+N +2S
CDP	-	+(b)I +S
MCR	-	+(b)I +C +N
MRC	-	+(b)I +C +I +S
LDC, STC	-	+(b)I +N +(n – 1)S +N

The cycle types N, S, I, and C are defined in Table 7-1 on page 7-4.

7.3 Branch and ARM branch with link

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

1. During the first cycle, a branch instruction calculates the branch destination 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.
2. During the second cycle, the ARM7TDMI-S core performs a Fetch from the branch destination. The return address is stored in r14 if the link bit is set.
3. During the third cycle, the ARM7TDMI-S core performs a Fetch from the destination + i, refilling the instruction pipeline. When the instruction is a branch with link, r14 is modified (4 is subtracted from it) to simplify return to `MOV PC,R14`. This modification ensures subroutines of the type `STM..{R14} LDM..{PC}` work correctly.

Table 7-3 shows the cycle timings, where:

pc Is the address of the branch instruction.

pc' Is an address calculated by the ARM7TDMI-S core.

(pc') Are the contents of that address.

Table 7-3 Branch instruction cycle operations

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0
1	pc+2i	w/h	0	(pc + 2i)	N cycle	0
2	pc'	w'/h'	0	(pc')	S cycle	0
3	pc'+i	w'/h'	0	(pc' + i)	S cycle	0
	pc'+2i	w'/h'	-	-	-	-

Note

This data applies only to branches in ARM and Thumb states, and to branch with link in ARM state.

7.4 Thumb branch with link

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

1. The first instruction acts as a simple data operation. It takes a single cycle to add the PC to the upper part of the offset and stores the result in r14 (LR).
2. The second instruction acts similar to the ARM BL instruction over three cycles:
   - During the first cycle, the ARM7TDMI-S core calculates the final branch destination while performing a prefetch from the current PC.
   - During the second cycle, the ARM7TDMI-S core performs a Fetch from the branch destination. The return address is stored in r14.
   - During the third cycle, the ARM7TDMI-S core performs a Fetch from the destination +2, refills the instruction pipeline, and modifies r14 (subtracting 2) to simplify the return to `MOV PC, R14`. This modification ensures that subroutines of the type `PUSH {..,LR} ; POP {..,PC}` work correctly.

Table 7-4 shows the cycle timings of the complete operation.

Table 7-4 Thumb long branch with link

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0
1	pc + 4	h	0	(pc + 4)	S cycle	0
2	pc + 6	h	0	(pc + 6)	N cycle	0
3	pc’	h	0	(pc’)	S cycle	0
4	pc’ + 2	h	0	(pc’ + 2)	S cycle	0
	pc’ + 4	-	-	-	-	-

Note

PC is the address of the first instruction of the operation.

Thumb BL operations are explained in detail in the *ARM Architecture Reference Manual*.

7.5 Branch and exchange

A *Branch and eXchange* (BX) operation takes three cycles, it is similar to a Branch:

1. During the first cycle, the ARM7TDMI-S core extracts the branch destination, and the new core state from the register source, while performing a prefetch from the current PC. This prefetch is performed in all cases, because by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch.
2. During the second cycle, the ARM7TDMI-S core performs a Fetch from the branch destination using the new instruction width, dependent on the state that has been selected.
3. During the third cycle, the ARM7TDMI-S core performs a Fetch from the destination +2 or +4 dependent on the new specified state, refilling the instruction pipeline.

Table 7-5 shows the cycle timings.

Table 7-5 Branch and exchange instruction cycle operations

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0	Tbit
1	pc + 2i	w/h	0	(pc + 2i)	N cycle	0	t
2	pc’	w’/h’	0	(pc’)	S cycle	0	t’
3	pc’+ i’	w’/h’	0	(pc’+i’)	S cycle	0	t’
	pc’ + 2i’	-	-	-	-	-	-

Note

i and i’ represent the instruction widths before and after the BX respectively.

In ARM state, Size is 2, and in Thumb state Size is 1. When changing from Thumb to ARM state, i equals 1, and i’ equals 2.

t, and t’ represent the states of the T bit before and after the BX respectively. In ARM state, Tbit is 0, and in Thumb state Tbit is 1. When changing from ARM to Thumb state, t equals 0, and t’ equals 1.

7.6 Data operations

A data operation executes in a single data path cycle except where the shift is determined by the contents of a register. The ARM7TDMI-S core reads a first register onto the A bus, and a second register or the immediate field onto the B bus.

The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction. The ARM7TDMI-S core writes the result (when required) into the destination register. (Compares and tests 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 PC is incremented.

When a register specifies the shift length, an additional data path cycle occurs before the data operation to copy the bottom 8 bits of that register into a holding latch in the barrel shifter. The instruction prefetch occurs during this first cycle. The operation cycle is internal (it does not request memory). Because the address remains stable through both cycles, the memory manager can merge this internal cycle with the following sequential access.

The PC can be one or more of the register operands. When the PC is the destination, external bus activity can be affected. When the ARM7TDMI-S core writes the result to the PC, the contents of the instruction pipeline are invalidated, and the ARM7TDMI-S core takes the address for the next instruction prefetch from the ALU rather than the address incrementer. The ARM7TDMI-S processor refills the instruction pipeline before any more execution takes place. During this time exceptions are locked out.

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

The data operation timing cycles are shown in Table 7-6.

Table 7-6 Data operation instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0
normal	1	pc+2i	w/h	0	(pc+2i)	S cycle	0
		pc+3i	-	-	-	-	-
dest=pc	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
	2	pc’	w/h	0	(pc’)	S cycle	0
	3	pc’+i	w/h	0	(pc’+i)	S cycle	0
		pc’+2i	-	-	-	-	-

Table 7-6 Data operation instruction cycle operations (continued)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0
shift(Rs)	1	pc+2i	w/h	0	(pc+2i)	I cycle	0
	2	pc+3i	w/h	0	-	S cycle	1
		pc+3i	-	-	-	-	-
shift(Rs)	1	pc+8	w	0	(pc+8)	I cycle	0
dest=pc	2	pc+12	w	0	-	N cycle	1
	3	pc’	w	0	(pc’)	S cycle	0
	4	pc’+4	w	0	(pc’+4)	S cycle	0
		pc’+8	-	-	-	-	-

Note

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

7.7 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 shown in Table 7-7 to Table 7-10 on page 7-13, in which m is the number of cycles required by the multiplication algorithm (see *Instruction cycle count summary* on page 7-5).

Table 7-7 Multiply instruction cycle operations

Cycle	Address	Write	Size	Data	TRANS[1:0]	Prot0
1	pc+2i	0	w/h	(pc+2i)	I cycle	0
2	pc+3i	0	w/h	-	I cycle	1
•	pc+3i	0	w/h	-	I cycle	1
m	pc+3i	0	w/h	-	I cycle	1
m+1	pc+3i	0	w/h	-	S cycle	1
	pc+3i	-	-	-	-	-

Table 7-8 Multiply-accumulate instruction cycle operations

Cycle	Address	Write	Size	Data	TRANS[1:0]	Prot0
1	pc+2i	0	w/h	(pc+2i)	I cycle	0
2	pc+2i	0	w/h	-	I cycle	1
•	pc+3i	0	w/h	-	I cycle	1
m	pc+3i	0	w/h	-	I cycle	1
m+1	pc+3i	0	w/h	-	I cycle	1
m+2	pc+3i	0	w/h	-	S cycle	1
	pc+3i	-	-	-	-	-

Table 7-9 Multiply long instruction cycle operations

Cycle	Address	Write	Size	Data	TRANS[1:0]	Prot0
1	pc+8	0	w	(pc+8)	I cycle	0
2	pc+12	0	w	-	I cycle	1
•	pc+12	0	w	-	I cycle	1
m	pc+12	0	w	-	I cycle	1
m+1	pc+12	0	w	-	I cycle	1
m+2	pc+12	0	w	-	S cycle	1
	pc+12	-	-	-	-	-

Note

Multiply long is available only in ARM state.

Table 7-10 Multiply-accumulate long instruction cycle operations

Cycle	Address	Write	Size	Data	TRANS[1:0]	Prot0
1	pc+8	0	w	(pc+8)	I cycle	0
2	pc+8	0	w	-	I cycle	1
•	pc+12	0	w	-	I cycle	1
m	pc+12	0	w	-	I cycle	1
m+1	pc+12	0	w	-	I cycle	1
m+2	pc+12	0	w	-	I cycle	1
m+3	pc+12	0	w	-	S cycle	1
	pc+12	-	-	-	-	-

Note

Multiply-accumulate long is available only in ARM state.

7.8 Load register

A load register instruction takes a variable number of cycles:

1. During the first cycle, the ARM7TDMI-S processor calculates the address to be loaded.
2. During the second cycle, the ARM7TDMI-S processor fetches the data from memory and performs the base register modification (if required).
3. During the third cycle, the ARM7TDMI-S processor transfers the data to the destination register. (External memory is not used.) Normally, the ARM7TDMI-S core merges this third cycle with the next prefetch to form one memory N-cycle.

The load register cycle timings are shown in Table 7-11, where:

b, h, and w Are byte, halfword and word as defined in Table 5-6 on page 5-51.

s Represents current supervisor-mode-dependent value.

u Is either 0, when the force translation bit is specified in the instruction (LDRT), or s at all other times.

Table 7-11 Load register instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	Prot1
normal	1	pc+2i	w/h	0	(pc+2i)	N cycle	0	s
	2	pc’	w/h/b	0	(pc’)	I cycle	1	u/s
	3	pc+3i	w/h	0	-	S cycle	1	s
		pc+3i	-	-	-	-	-	-
dest=pc	1	pc+8	w	0	(pc+8)	N cycle	0	s
	2	da	w/h/b	0	pc’	I cycle	1	u/s
	3	pc+12	w	0	-	N cycle	1	s
	4	pc’	w	0	(pc’)	S cycle	0	s
	5	pc’+4	w	0	(pc’+4)	S cycle	0	s
		pc’+8	-	-	-	-	-	-

Either the base or the destination (or both) can be the PC. The prefetch sequence changes when the PC is affected by the instruction. If the Data Fetch aborts, the ARM7TDMI-S processor prevents modification of the destination register.

Note

Destination equals PC is not possible in Thumb state.

7.9 Store register

A store register has two cycles:

1. During the first cycle, the ARM7TDMI-S core calculates the address to be stored.
2. During the second cycle, the ARM7TDMI-S core performs the base modification, and writes the data to memory (if required).

The store register cycle timings are shown in Table 7-12, where:

s Represents current mode-dependent value.

t Is either 0, when the T bit is specified in the instruction (STRT) or c at all other times.

Table 7-12 Store register instruction cycle operations

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0	Prot1
1	pc+2i	w/h	0	(pc+2i)	N cycle	0	s
2	da	b/h/w	1	Rd	N cycle	1	t
	pc+3i	-	-	-	-	-	-

7.10 Load multiple registers

A *LoaD Multiple* (LDM) takes four cycles:

1. During the first cycle, the ARM7TDMI-S core calculates the address of the first word to be transferred, while performing a prefetch from memory.
2. During the second cycle, the ARM7TDMI-S core fetches the first word and performs the base modification.
3. During the third cycle, the ARM7TDMI-S core moves the first word to the appropriate destination register and fetches the second word from memory. The ARM7TDMI-S latches the modified base internally, in case it is required after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed.
4. During the fourth and final (internal) cycle, the ARM7TDMI-S core moves the last word to its destination register. The last cycle can be merged with the next instruction prefetch to form a single memory N-cycle.

When an abort occurs, the instruction continues to completion. The ARM7TDMI-S core prevents all register writing after the abort. The ARM7TDMI-S core changes the final cycle to restore the modified base register (which the load activity before the abort occurred might have overwritten).

When the PC is in the list of registers to be loaded, the ARM7TDMI-S core invalidates the current instruction pipeline. The PC is always the last register to load, so an abort at any point prevents the PC from being overwritten.

Note

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

The LDM cycle timings are shown in Table 7-13.

Table 7-13 Load multiple registers instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0
1 register	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
	2	da	w	0	da	I cycle	1
	3	pc+3i	w/h	0	-	S cycle	1
		pc+3i	-	-	-	-	-

Table 7-13 Load multiple registers instruction cycle operations (continued)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0
1 register	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
dest=pc	2	da	w	0	pc’	I cycle	1
	3	pc+3i	w/h	0	-	N cycle	1
	4	pc’	w/h	0	(pc’)	S cycle	0
	5	pc’+i	w/h	0	(pc’+i)	S cycle	0
		pc’+2i	-	-	-	-	-
n registers	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
(n>1)	2	da	w	0	da	S cycle	1
	•	da++	w	0	(da++)	S cycle	1
	n	da++	w	0	(da++)	S cycle	1
	n+1	da++	w	0	(da++)	I cycle	1
	n+2	pc+3i	w/h	0	-	S cycle	1
		pc+3i	-	-	-	-	-
n registers	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
(n>1)	2	da	w	0	da	S cycle	1
incl pc	•	da++	w	0	(da++)	S cycle	1
	n	da++	w	0	(da++)	S cycle	1
	n+1	da++	w	0	pc’	I cycle	1
	n+2	pc+3i	w/h	0	-	N cycle	1
	n+3	pc’	w/h	0	(pc’)	S cycle	0
	n+4	pc’+i	w/h	0	(pc’+i)	S cycle	0
		pc’+2i	-	-	-	-	-

7.11 Store multiple registers

STore Multiple (STM) proceeds very much as LDM, although without the final cycle. There are therefore two cycles:

1. During the first cycle, the ARM7TDMI-S core calculates the address of the first word to be stored.
2. During the second cycle, the ARM7TDMI-S core performs the base modification, and writes the data to memory.

Restart is straightforward because there is no general overwriting of registers.

The STM cycle timings are shown in Table 7-14.

Table 7-14 Store multiple registers instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0
1 register	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
	2	da	w	1	R	N cycle	1
		pc+3i					
n registers	1	pc+8	w/h	0	(pc+2i)	N cycle	0
(n>1)	2	da	w	1	R	S cycle	1
	•	da++	w	1	R’	S cycle	1
	n	da++	w	1	R”	S cycle	1
	n+1	da++	w	1	R”’	N cycle	1
		pc+12					

7.12 Data swap

Data swap is similar to the load and store register instructions, although the swap takes place in cycles 2 and 3. The data is fetched from external memory in the second cycle, and in the third cycle the contents of the source register are written to the external memory. In the fourth cycle the data read during cycle 2 is written into the destination register.

The data swapped can be a byte or word quantity (b/w).

The ARM7TDMI-S core might abort the swap operation in either the read or write cycle. The swap operation (read or write) does not affect the destination register.

The data swap cycle timings are shown in Table 7-15, where b and w are byte and word as defined in Table 5-6 on page 5-51.

Table 7-15 Data swap instruction cycle operations

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0	Lock
1	pc+8	w	0	(pc+8)	N cycle	0	0
2	Rn	w/b	0	(Rn)	N cycle	1	1
3	Rn	w/b	1	Rm	I cycle	1	1
4	pc+12	w	0	-	S cycle	1	0
	pc+12						

Note

Data swap cannot be executed in Thumb state.

The **LOCK** output of the ARM7TDMI-S processor is driven HIGH for both load and store data cycles to indicate to the memory controller that this is an atomic operation.

7.13 Software interrupt, and exception entry

Exceptions, and *SoftWare Interrupts* (SWIs) force the PC to a specific value, and refill the instruction pipeline from this address:

1. During the first cycle, the ARM7TDMI-S core constructs the forced address, and a mode change might take place. The ARM7TDMI-S core moves the return address to r14 and moves the CPSR to SPSR_svc.
2. During the second cycle, the ARM7TDMI-S core modifies the return address to facilitate return (although this modification is less useful than in the case of branch with link).
3. The third cycle is required only to complete the refilling of the instruction pipeline.

The SWI cycle timings are shown in Table 7-16, where:

s Represents the current supervisor mode dependent value.

t Represents the current Thumb state value.

pc Is, for software interrupts, the address of the SWI instruction.
For exceptions, this is the address of the instruction following the last one to be executed before entering the exception.
For Prefetch Aborts, this is the address of the aborting instruction.
For Data Aborts, this is the address of the instruction following the one that attempted the aborted data transfer.

Xn Is the appropriate trap address.

Table 7-16 Software interrupt instruction cycle operations

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0	Prot1	Mode	Tbit
1	pc+2i	w/h	0	(pc+2i)	N cycle	0	s	old mode	t
2	Xn	w’	0	(Xn)	S cycle	0	1	exception mode	0
3	Xn+4	w’	0	(Xn+4)	S cycle	0	1	exception mode	0
	Xn+8								

7.14 Coprocessor data processing operation

A *Coprocessor Data Processing* (CDP) operation is a request from the ARM7TDMI-S core for the coprocessor to initiate some action. There is no need to complete the action immediately, but the coprocessor must commit to completion before driving CPB LOW.

If the coprocessor cannot perform the requested task, it leaves **CPA** and **CPB** HIGH. When the coprocessor is able to perform the task, but cannot commit immediately, the coprocessor drives **CPA** LOW, but leaves **CPB** HIGH until able to commit. The ARM7TDMI-S processor busy-waits until **CPB** goes LOW. However, an interrupt might cause the ARM7TDMI-S core to abandon a busy-waiting coprocessor instruction (see *Consequences of busy-waiting* on page 4-8).

The coprocessor data operations cycle timings are shown in Table 7-17.

Table 7-17 Coprocessor data operation instruction cycle operations

Cycle		Address	Write	Size	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
ready	1	pc+8	0	w	(pc+8)	N cycle	0	0	0	0
		pc+12								
not ready	1	pc+8	0	w	(pc+8)	I cycle	0	0	0	1
	2	pc+8	0	w	-	I cycle	1	0	0	1
	•	pc+8	0	w	-	I cycle	1	0	0	1
	n	pc+8	0	w	-	N cycle	1	0	0	0
		pc+12								

Note

Coprocessor operations are available only in ARM state.

7.15 Load coprocessor register (from memory to coprocessor)

The *LoaD Coprocessor* (LDC) operation transfers one or more words of data from memory to coprocessor registers.

The coprocessor commits to the transfer only when it is ready to accept the data. The **WRITE** line is driven LOW during the transfer cycle. When **CPB** goes LOW, the ARM7TDMI-S core produces 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. An interrupt can cause the ARM7TDMI-S core to abandon a busy-waiting coprocessor instruction (see *Consequences of busy-waiting* on page 4-8).

The first cycle (and any busy-wait cycles) generates the transfer address. The second cycle performs the write-back of the address base. The coprocessor indicates the last transfer cycle by driving **CPA** and **CPB** HIGH.

The load coprocessor register cycle timings are shown in Table 7-18.

Table 7-18 Load coprocessor register instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
1 register ready	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
	2	da	w	0	(da)	N cycle	1	1	1	1
		pc+12								
1 register not ready	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
	2	pc+8	w	0	-	I cycle	1	0	0	1
	•	pc+8	w	0	-	I cycle	1	0	0	1
	n	pc+8	w	0	-	N cycle	1	0	0	0
	n+1	da	w	0	(da)	N cycle	1	1	1	1
		pc+12								
m registers (m>1) ready	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
	2	da	w	0	(da)	S cycle	1	1	0	0
	•	da++	w	0	(da++)	S cycle	1	1	0	0
	m	da++	w	0	(da++)	S cycle	1	1	0	0
	m+1	da++	w	0	(da++)	N cycle	1	1	1	1
		pc+12								

Table 7-18 Load coprocessor register instruction cycle operations (continued)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
m registers (m>1) not ready	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
	2	pc+8	w	0	-	I cycle	1	0	0	1
	•	pc+8	w	0	-	I cycle	1	0	0	1
	n	pc+8	w	0	-	N cycle	1	0	0	0
	n+1	da	w	0	(da)	S cycle	1	1	0	0
	•	da++		0	(da++)	S cycle	1	1	0	0
	n+m	da++	w	0	(da++)	S cycle	1	1	0	0
	n+m+1	da++	w	0	(da++)	N cycle	1	1	1	1
		pc+12								

Note

Coprocessor operations are available only in ARM state.

7.16 Store coprocessor register (from coprocessor to memory)

The *STore Coprocessor* (STC) operation transfers one or more words of data from coprocessor registers to memory.

The coprocessor commits to the transfer only when it is ready to write data. The **WRITE** line is driven HIGH during the transfer cycle. When **CPB** goes LOW, the ARM7TDMI-S core produces addresses, and expects the coprocessor to write the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred. An interrupt can cause the ARM7TDMI-S core to abandon a busy-waiting coprocessor instruction (see *Consequences of busy-waiting* on page 4-8).

The first cycle (and any busy-wait cycles) generates the transfer address. The second cycle performs the write-back of the address base. The coprocessor indicates the last transfer cycle by driving **CPA** and **CPB** HIGH.

The store coprocessor register cycle timings are shown in Table 7-19.

Table 7-19 Store coprocessor register instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
1 register ready	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
	2	da	w	1	CPdata	N cycle	1	1	1	1
		pc+12								
1 register not ready	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
	2	pc+8	w	0	-	I cycle	1	0	0	1
	•	pc+8	w	0	-	I cycle	1	0	0	1
	n	pc+8	w	0	-	N cycle	1	0	0	0
	n+1	da	w	1	CPdata	N cycle	1	1	1	1
		pc+12								
m registers (m>1) ready	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
	2	da	w	1	CPdata	S cycle	1	1	0	0
	•	da++	w	1	CPdata’	S cycle	1	1	0	0
	m	da++	w	1	CPdata’’	S cycle	1	1	0	0
	m+1	da++	w	1	CPdata’’’	N cycle	1	1	1	1
		pc+12								

Table 7-19 Store coprocessor register instruction cycle operations (continued)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
m registers (m>1) not ready	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
	2	pc+8	w	0	-	I cycle	1	0	0	1
	•	pc+8	w	0	-	I cycle	1	0	0	1
	n	pc+8	w	0	-	N cycle	1	0	0	0
	n+1	da	w	1	CPdata	S cycle	1	1	0	0
	•	da++	w	1	CPdata	S cycle	1	1	0	0
	n+m	da++	w	1	CPdata	S cycle	1	1	0	0
	n+m+1	da++	w	1	CPdata	N cycle	1	1	1	1
		pc+12								

Note

Coprocessor operations are available only in ARM state.

7.17 Coprocessor register transfer (move from coprocessor to ARM register)

The *Move fRom Coprocessor* (MRC) operation reads a single coprocessor register into the specified ARM register.

Data is transferred in the second cycle and written to the ARM register during the third cycle of the operation.

If the coprocessor signals busy-wait by asserting **CPB**, an interrupt can cause the ARM7TDMI-S core to abandon the coprocessor instruction (see *Consequences of busy-waiting* on page 4-8).

As is the case with all ARM7TDMI-S register load instructions, the ARM7TDMI-S core might merge the third cycle with the following prefetch cycle into a merged I-S cycle.

The MRC cycle timings are shown in Table 7-20.

Table 7-20 Coprocessor register transfer (MRC)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
ready	1	pc+8	w	0	(pc+8)	C cycle	0	0	0	0
	2	pc+12	w	0	CPdata	I cycle	1	1	1	1
	3	pc+12	w	0	-	S cycle	1	1	-	-
		pc+12								
not ready	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
	2	pc+8	w	0	-	I cycle	1	0	0	1
	•	pc+8	w	0	-	I cycle	1	0	0	1
	n	pc+8	w	0	-	C cycle	1	0	0	0
	n+1	pc+12	w	0	CPdata	I cycle	1	1	1	1
	n+2	pc+12	w	0	-	S cycle	1	1	-	-
		pc+12								

Note

This operation cannot occur in Thumb state.

7.18 Coprocessor register transfer (move from ARM register to coprocessor)

The *Move to CoprocessoR* (MCR) operation transfers the contents of a single ARM register to a specified coprocessor register.

The data is transferred to the coprocessor during the second cycle. If the coprocessor signals busy-wait by asserting **CPB**, an interrupt can cause the ARM7TDMI-S core to abandon the coprocessor instruction (see *Consequences of busy-waiting* on page 4-8).

The MCR cycle timings are shown in Table 7-21.

Table 7-21 Coprocessor register transfer (MCR)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
ready	1	pc+8	w	0	(pc+8)	C cycle	0	0	0	0
	2	pc+12	w	1	Rd	N cycle	1	1	1	1
		pc+12								
not ready	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
	2	pc+8	w	0	-	I cycle	1	0	0	1
	•	pc+8	w	0	-	I cycle	1	0	0	1
	n	pc+8	w	0	-	C cycle	1	0	0	0
	n+1	pc+12	w	1	Rd	N cycle	1	1	1	1
		pc+12								

Note

Coprocessor operations are available only in ARM state.

7.19 Undefined instructions and coprocessor absent

The undefined instruction trap is taken if an undefined instruction is executed. For a definition of undefined instructions, see the *ARM Architecture Reference Manual*.

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

Note

By default **CPA** and **CPB** must be driven HIGH unless the coprocessor instruction is being handled by a coprocessor.

Undefined instruction cycle timings are shown in Table 7-22.

Table 7-22 Undefined instruction cycle operations

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA and CPB	Prot1	Mode	Tbit
1	pc+2i	w/h	0	(pc+2i)	I cycle	0	0	1	s	Old	t
2	pc+2i	w/h	0	-	N cycle	0	1	1	s	Old	t
3	Xn	w’	0	(Xn)	S cycle	0	1	1	1	00100	0
4	Xn+4	w’	0	(Xn+4)	S cycle	0	1	1	1	00100	0
	Xn+8										

where:

s Represents the current mode-dependent value.

t Represents the current state-dependent value.

Note

Coprocessor operations are available only in ARM state.

7.20 Unexecuted instructions

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

Unexecuted instruction cycle timings are shown in Table 7-23.

Table 7-23 Unexecuted instruction cycle operations

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0
1	pc+2i	w/h	0	(pc+2i)	S cycle	0
	pc+3i					

Chapter 8
AC Parameters

This chapter gives the AC timing parameters of the ARM7TDMI-S processor. It contains the following sections:

8.1 Timing diagrams

This section contains timing diagrams, as follows:

- *Timing parameters for data accesses*
- *Coprocessor timing* on page 8-4
- *Exception and configuration input timing* on page 8-5
- *Debug timing* on page 8-6
- *Scan timing* on page 8-7.

8.1.1 Timing parameters for data accesses

Timing parameters for data accesses are shown in Figure 8-1 on page 8-3.

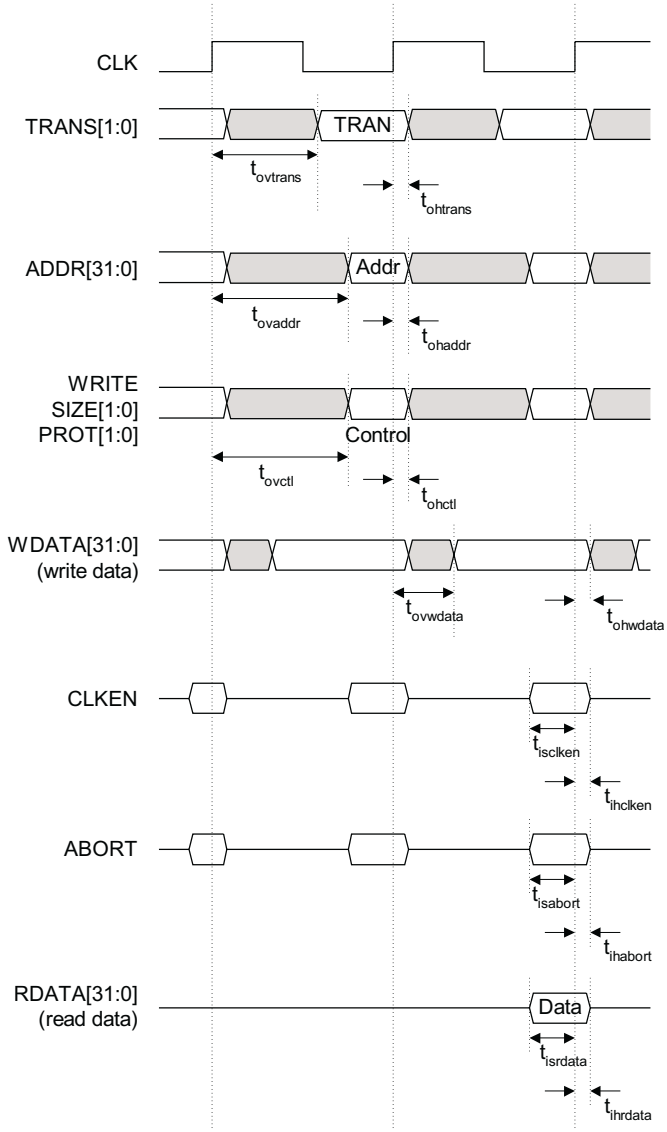


Figure 8-1 Timing parameters for data accesses

Note

The timing for both read and write data access are superimposed in Figure 8-1 on page 8-3. The **WRITE** signal conveys whether the access uses the **RDATA** or **WDATA** port.

CLKEN LOW stretches the data access when the read or write transaction is unable to complete within a single cycle.

The data buses are used for transfer only when the transaction signals **TRANS[1:0]** indicate a valid memory cycle or a coprocessor register transfer cycle.

8.1.2 Coprocessor timing

Coprocessor timing parameters are shown in Figure 8-2.

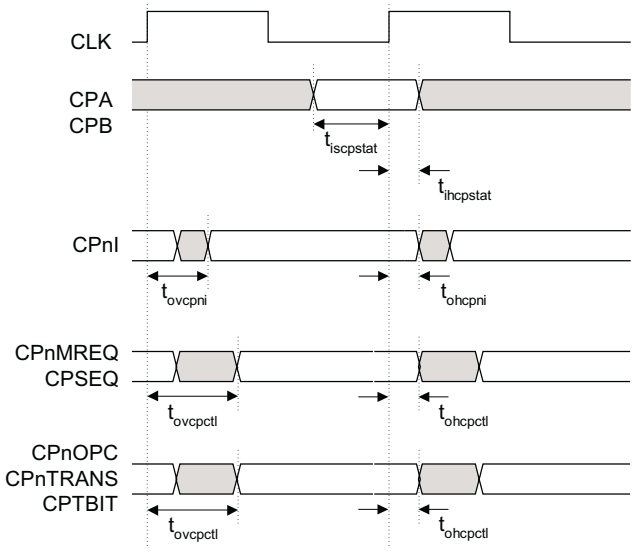


Figure 8-2 Coprocessor timing

8.1.3 Exception and configuration input timing

Exception and configuration input timing parameters are shown in Figure 8-3.

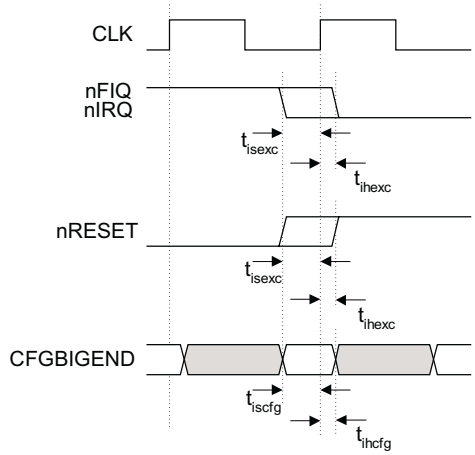


Figure 8-3 Exception and configuration input timing

8.1.4 Debug timing

Debug timing parameters are shown in Figure 8-4.

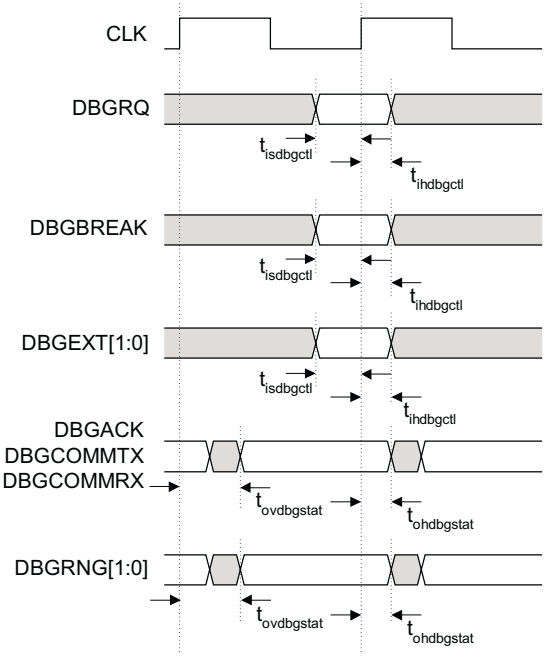


Figure 8-4 Debug timing

Note

DBGBREAK is sampled on rising clock, so external data-dependent breakpoints and watchpoints must be matched and signaled by this edge.

8.1.5 Scan timing

Scan timing parameters are shown in Figure 8-5.

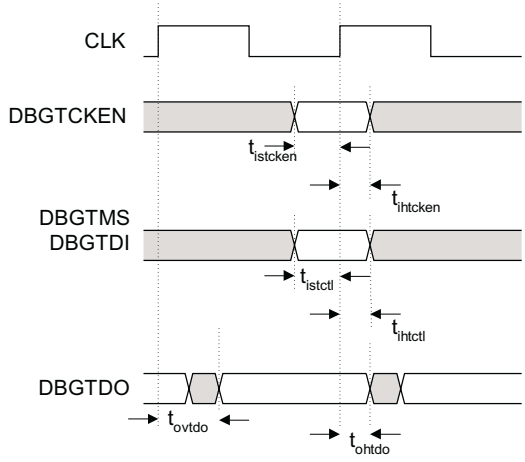


Figure 8-5 Scan timing

8.2 AC timing parameter definitions

Table 8-1 shows target AC parameters. All figures are expressed as percentages of the **CLK** period at maximum operating frequency. Please contact your silicon supplier for more details.

Note

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

Table 8-1 Provisional AC parameters

Symbol	Parameter	Min	Max
t_{cyc}	**CLK** cycle time	100%	-
$t_{isclken}$	**CLKEN** input setup to rising **CLK**	40%	-
$t_{ihclken}$	**CLKEN** input hold from rising **CLK**	-	0%
$t_{isabort}$	**ABORT** input setup to rising **CLK**	15%	-
$t_{ihabort}$	**ABORT** input hold from rising **CLK**	-	0%
$t_{isrdata}$	**RDATA** input setup to rising **CLK**	10%	-
$t_{ihrdata}$	**RDATA** input hold from rising **CLK**	-	0%
t_{ovaddr}	Rising **CLK** to **ADDR** valid	-	90%
t_{ohaddr}	**ADDR** hold time from rising **CLK**	>0%	-
t_{ovctl}	Rising **CLK** to control valid	-	90%
t_{ohctl}	Control hold time from rising **CLK**	>0%	-
$t_{ovtrans}$	Rising **CLK** to transaction type valid	-	50%
$t_{ohtrans}$	Transaction type hold time from rising **CLK**	>0%	-
$t_{ovwdata}$	Rising **CLK** to **WDATA** valid	-	40%
$t_{ohwdata}$	**WDATA** hold time from rising **CLK**	>0%	-
$t_{iscpstat}$	**CPA**, **CPB** input setup to rising **CLK**	20%	-
$t_{ihcpstat}$	**CPA**, **CPB** input hold from rising **CLK**	-	0%

Table 8-1 Provisional AC parameters (continued)

Symbol	Parameter	Min	Max
$t_{ovcpctl}$	Rising **CLK** to coprocessor control valid	-	80%
$t_{ohcpctl}$	Coprocessor control hold time from rising **CLK**	>0%	-
t_{ovcpni}	Rising **CLK** to coprocessor **CPnI** valid	-	40%
t_{ohcpni}	Coprocessor **CPnI** hold time from rising **CLK**	>0%	-
t_{isexc}	**nFIQ**, **nIRQ**, **nRESET** setup to rising **CLK**	10%	-
t_{ihexc}	**nFIQ**, **nIRQ**, **nRESET** hold from rising **CLK**	-	0%
t_{iscfg}	**CFGBIGEND** setup to rising **CLK**	10%	-
t_{ihcfg}	**CFGBIGEND** hold from rising **CLK**	-	0%
$t_{isdbgstat}$	Debug status inputs setup to rising **CLK**	10%	-
$t_{ihdbgstat}$	Debug status inputs hold from rising **CLK**	-	0%
$t_{ovdbgctl}$	Rising **CLK** to debug control valid	-	40%
$t_{ohdbctl}$	Debug control hold time from rising **CLK**	>0%	-
$t_{istcken}$	**DBGTCKEN** input setup to rising **CLK**	40%	-
$t_{ihtcken}$	**DBGTCKEN** input hold from rising **CLK**	-	0%
t_{istctl}	**DBGTDI**, **DBGTMS** input setup to rising **CLK**	35%	-
t_{ihtctl}	**DBGTDI**, **DBGTMS** input hold from rising **CLK**	-	0%
t_{ovtdo}	Rising **CLK** to **DBGTDO** valid	-	20%
t_{ohtdo}	**DBGTDO** hold time from rising **CLK**	>0%	-
$t_{ovdbgstat}$	Rising **CLK** to debug status valid	40%	-
$t_{ohdbgstat}$	Debug status hold time	>0%	-

Appendix A
Signal Descriptions

This appendix lists and describes all the ARM7TDMI-S processor signals. It contains the following section:

A.1 Signal descriptions

The signals of the ARM7TDMI-S processor are shown in Table A-1.

Table A-1 Signal descriptions

Name	Type	Description
ABORT	Input	Memory abort or bus error. This is an input that is used by the memory system to signal to the processor that a requested access is disallowed.
ADDR[31:0]	Output	This is the processor address bus.
CFGBIGEND	Input	Big-endian configuration. When this signal is HIGH, the processor treats bytes in memory as being in big-endian format. When the signal is LOW, memory is treated as little-endian. **CFGBIGEND** is normally a static configuration signal. This signal is analogous to **BIGEND** on the hard macrocell.
CLK	Input	Clock input. This clock times all ARM7TDMI-S memory accesses and internal operations. All outputs change from the rising edge of **CLK** and all inputs are sampled on the rising edge of **CLK**. The **CLKEN** input can be used with a free-running **CLK** to add synchronous wait-states. Alternatively, the clock can be stretched indefinitely in either phase to allow access to slow peripherals or memory or to put the system into a low-power state. **CLK** is also used for serial scan-chain debug operation with the EmbeddedICE-RT tool-chain. This signal is analogous to inverted **MCLK** on the hard macrocell.
CLKEN	Input	Wait state control. When accessing slow peripherals, the ARM7TDMI-S can be made to wait for an integer number of **CLK** cycles by driving **CLKEN**LOW. When the **CLKEN** control is not used, it must be tied HIGH. This signal is analogous to **nWAIT** on the hard macrocell.
CPA	Input	Coprocessor absent handshake. A coprocessor that is capable of performing the operation that the ARM7TDMI-S is requesting (by asserting **CPnI**), takes **CPA** LOW, set up to the cycle edge that precedes the coprocessor access. When **CPA** is signaled HIGH and the coprocessor cycle is executed (as signaled by **CPnI** signaled LOW), the ARM7TDMI-S aborts the coprocessor handshake and takes the undefined instruction trap. When **CPA** is LOW and remains LOW, the ARM7TDMI-S busy-waits until **CPB** is LOW and then completes the coprocessor instruction.
CPB	Input	Coprocessor busy handshake. A coprocessor is capable of performing the operation requested by the ARM7TDMI-S (by asserting **CPnI**), but cannot commit to starting it immediately, this is indicated by driving **CPB**HIGH. When the coprocessor is ready to start, it takes **CPB** LOW, with the signal being set up before the start of the coprocessor instruction execution cycle.

Table A-1 Signal descriptions (continued)

Name	Type	Description
CPnI	Output	Not coprocessor instruction. When the ARM7TDMI-S executes a coprocessor instruction, it takes this output LOW and waits for a response from the coprocessor. The action taken depends on this response, which the coprocessor signals on the **CPA** and **CPB** inputs.
CPnMREQ	Output	Not memory request. When LOW, this signal indicates that the processor requires memory access during the next transaction. This signal is analogous to **nMREQ** on the hard macrocell.
CPnOPC	Output	Not opcode fetch. When LOW, this signal indicates that the processor is fetching an instruction from memory. When HIGH, data (if present) is being transferred. This signal is analogous to **nOPC** on the hard macrocell and to **BPROT[0]** on the AMBA ASB.
CPSEQ	Output	Sequential address. This output signal becomes HIGH when the address of the next memory cycle is related to that of the last memory access. The new address is either the same as the previous one or four greater in ARM state or two greater when fetching opcodes in Thumb state. This signal is analogous to **SEQ** on the hard macrocell.
CPTBIT	Output	When HIGH, this signal indicates to a coprocessor that the processor is executing the Thumb instruction set. When LOW, the processor is executing the ARM instruction set.
CPnTRANS	Output	Not memory translate. When LOW, this signal indicates that the processor is in User mode. It can be used to signal to memory management hardware when to bypass translation of the addresses or as an indicator of privileged mode activity. This signal is analogous to **nTRANS** on the hard macrocell.
DBGACK	Output	Debug acknowledge. When HIGH, this signal **DBGBREAK** indicates that the ARM7TDMI-S is in debug state. It is enabled only when **DBGEN** is HIGH.
DBGBREAK	Input	EmbeddedICE-RT breakpoint/watchpoint indicator. This signal enables external hardware to halt the execution of the processor for debug purposes. When HIGH, this signal causes the current memory access to be breakpointed. When the memory access is an instruction fetch, the ARM7TDMI-S enters debug state if the instruction reaches the execute stage of the ARM7TDMI-S pipeline. When the memory access is for data, the ARM7TDMI-S enters debug state after the current instruction completes execution. This enables extension of the internal breakpoints provided by the EmbeddedICE-RT module. **DBGBREAK** is enabled only when **DBGEN** is HIGH. This signal is analogous to **BREAKPT** on the hard macrocell.

Table A-1 Signal descriptions (continued)

Name	Type	Description
DBGCOMMRX	Output	EmbeddedICE-RT communications channel receive. When HIGH, this signal indicates that the comms channel receive buffer is full. **DBGCOMMRX** is enabled only when **DBGEN** is HIGH. This signal is analogous to **COMMRX** on the hard macrocell.
DBGCOMMTX	Output	EmbeddedICE-RT communications channel transmit. When HIGH, this signal denotes that the comms channel transmit buffer is empty. **DBGCOMMTX** is enabled only when **DBGEN** is HIGH. This signal is analogous to **COMMTX** on the hard macrocell.
DBGEN	Input	Debug enable. This input signal enables the debug features of the ARM7TDMI-S. If you intend to use the ARM7TDMI-S debug features, tie this signal HIGH. Drive this signal LOW only when debugging is not required.
DBGnEXEC	Output	Not executed. When HIGH, this signal indicates that the instruction in the execution unit is not being executed (because, for example, it has failed its condition code check).
DBGEXT[1:0]	Input	EmbeddedICE-RT external input 0, external input 1. These are inputs to the EmbeddedICE-RT macrocell logic in the ARM7TDMI-S that allow breakpoints and/or watchpoints to be dependent on an external condition. The inputs are enabled only when **DBGEN** is HIGH. These signals are analogous to **EXTERN[1:0]** on the hard macrocell.
DBGINSTRVALID	Output	Instruction executed signal. Goes HIGH for one cycle for each instruction committed to the execute stage of the pipeline. Used by ETM7 to trace the ARM7TDMI-S processor pipeline. This signal is analogous to **INSTRVALID** on the hard macrocell.
DBGRNG[1:0]	Output	EmbeddedICE-RT rangeout. This signal indicates that EmbeddedICE-RT watchpoint register has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint enable control bit. The signal is enabled only when **DBGEN** is HIGH. This signal is analogous to **RANGE[1:0]** on the hard macrocell.
DBGRQ	Input	Debug request. This internally synchronized input signal requests the processor to enter debug state. **DBGRQ** is enabled only when **DBGEN** is HIGH.
DBGTCKEN	Input	Test clock enable. **DBGTCKEN** is enabled only when **DBGEN** is HIGH.
DBGTDI	Input	EmbeddedICE-RT data in. JTAG test data input. **DBGTDI** is enabled only when **DBGEN** is HIGH.
DBGTDO	Output	EmbeddedICE-RT data out. Output from the boundary scan logic. **DBGTDO** is enabled only when **DBGEN** is HIGH.

Table A-1 Signal descriptions (continued)

Name	Type	Description
DBGnTDOEN	Output	Not **DBGTDO** enable. When LOW, this signal denotes that serial data is being driven out on the **DBGTDO** output. **DBGnTDOEN** is normally used as an output enable for a **DBGTDO** pin in a packaged part.
DBGTMS	Input	EmbeddedICE-RT mode select. JTAG test mode select. **DBGTMS** is enabled only when **DBGEN** is HIGH.
DBGnTRST	Input	Not test reset. This is the active-low reset signal for the EmbeddedICE-RT macrocell internal state.
DMORE	Output	Asserted for `LDM` and `STM` instructions (new for Rev 4). This signal has the effect of making memory accesses more efficient.
nFIQ	Input	Active-low fast interrupt request. This is a high priority synchronous interrupt request to the processor. If the appropriate enable in the processor is active when this signal is taken LOW, the processor is interrupted. This signal is level-sensitive and must be held LOW until a suitable interrupt acknowledge response is received from the processor. This signal is analogous to **nFIQ** on the hard macrocell when **ISYNC** is HIGH.
nIRQ	Input	Active-low interrupt request. This is a low priority synchronous interrupt request to the processor. If the appropriate enable in the processor is active when this signal is taken LOW, the processor is interrupted. This signal is level-sensitive and must be held LOW until a suitable interrupt acknowledge response is received from the processor. This signal is analogous to **nIRQ** on the hard macrocell when **ISYNC** is HIGH.
LOCK	Output	Locked transaction operation. When **LOCK** is HIGH, the processor is performing a locked memory access, the arbiter must wait until **LOCK** goes LOW before allowing another device to access the memory.
PROT[1:0]	Output	These output signals to the memory system indicate whether the output is code or data and whether the access is User Mode or privileged access: `x0` opcode fetch `x1` data access `0x` User-mode access `1x` supervisor or privileged mode access.
RDATA[31:0]	Input	Read data input bus. This is the read data bus used to transfer instructions and data between the processor and memory. The data on this bus is sampled by the processor at the end of the clock cycle during read accesses (that is, when **WRITE** is LOW). This signal is analogous to **DIN[31:0]** on the hard macrocell.

Table A-1 Signal descriptions (continued)

Name	Type	Description
nRESET	Input	Not reset. This input signal forces the processor to terminate the current instruction and subsequently to enter the reset vector in supervisor mode. It must be asserted for at least two cycles. A LOW level forces the instruction being executed to terminate abnormally on the next nonwait cycle and causes the processor to perform idle cycles at the bus interface. When **nRESET** becomes HIGH for at least one clock cycle, the processor restarts from address 0.
SCANENABLE	Input	Scan test path enable (for automatic test pattern generation) is LOW for normal system configuration and HIGH during scan testing.
SCANIN	Input	Scan test path serial input (for automatic test pattern generation). Serial shift register input is active when **SCANENABLE** is active (HIGH).
SCANOUT	Output	Scan test path serial output (for automatic test pattern generation). Serial shift register output is active when **SCANENABLE** is active (HIGH).
SIZE[1:0]	Output	Memory access width. These output signals indicate to the external memory system when a word transfer or a halfword or byte length is required: 00 8-bit byte access (addressed in word by **ADDR[1:0]**) 01 16-bit halfword access (addressed in word by **ADDR[1]**) 10 32-bit word access (always word-aligned) 11 (reserved) This signal is analogous to **MAS[1:0]** on the hard macrocell.
TRANS[1:0]	Output	Next transaction type. **TRANS** indicates the next transaction type: 00 address-only (internal operation cycle) 01 coprocessor 10 memory access at nonsequential address 11 memory access at sequential burst address. The **TRANS[1]** signal is analogous to inverted **nMREQ** and the **TRANS[0]** signal is analogous to **SEQ** on the hard macrocell. **TRANS** is analogous to **BTRAN** on the AMBA system bus.
V_{DD}		Power supply to the device.

Table A-1 Signal descriptions (continued)

Name	Type	Description
V_{SS}		Ground reference for all signals.
WDATA[31:0]	Output	Write data output bus. This is the write data bus, used to transfer data from the processor to the memory or coprocessor system. Write data is set up to the end of the cycle of store accesses (that is, when **WRITE** is HIGH) and remains valid throughout wait states. This signal is analogous to **DOUT[31:0]** on the hard macrocell.
WRITE	Output	Write/read access. When HIGH, **WRITE** indicates a processor write cycle, when LOW, it indicates a processor read cycle. This signal is analogous to **nRW** on the hard macrocell.

Appendix B
Differences Between the ARM7TDMI-S and the ARM7TDMI

This appendix describes the differences between the ARM7TDMI-S and ARM7TDMI macrocell interfaces. It contains the following sections:

B.1 Interface signals

The signal names have prefixes that identify groups of functionally-related signals:

CFGxxx Shows configuration inputs (typically hard-wired for an embedded application).

CPxxx Shows coprocessor expansion interface signals.

DBGxxx Shows scan-based EmbeddedICE-RT debug support input or output.

Other signals provide the system designer interface, which is primarily memory-mapped. Table B-1 shows the ARM7TDMI-S (Rev 4) processor signals with their ARM7TDMI (Rev 4) hard macrocell equivalent signals.

Table B-1 ARM7TDMI-S processor signals and ARM7TDMI hard macrocell equivalents

ARM7TDMI-S processor signal	Function	ARM7TDMI hard macrocell equivalent
ABORT	1 = memory abort or bus error. 0 = no error.	**ABORT**
ADDR[31:0] [a]	32-bit address output bus, available in the cycle preceding the memory cycle.	**A[31:0]**
CFGBIGEND	1 = big-endian configuration. 0 = little-endian configuration.	**BIGEND**
CLK [b]	Master rising edge clock. All inputs are sampled on the rising edge of **CLK**. All timing dependencies are from the rising edge of **CLK**.	**MCLK**
CLKEN [c]	System memory interface clock enable: 1 = advance the core on rising **CLK**. 0 = prevent the core advancing on rising **CLK**.	**nWAIT**
CPA [d]	Coprocessor absent. Tie HIGH when no coprocessor is present.	**CPA**
CPB [d]	Coprocessor busy. Tie HIGH when no coprocessor is present.	**CPB**
CPnI	Active LOW coprocessor instruction execute qualifier.	**nCPI**
CPnMREQ	Active LOW memory request signal, pipelined in the preceding access. This is a coprocessor interface signal. Use the ARM7TDMI-S output **TRANS[1:0]** for bus interface design.	**nMREQ**

Table B-1 ARM7TDMI-S processor signals and ARM7TDMI hard macrocell equivalents (continued)

ARM7TDMI-S processor signal	Function	ARM7TDMI hard macrocell equivalent
CPnOPC	Active LOW opcode fetch qualifier output, pipelined in the preceding access. This is a coprocessor interface signal. Use the ARM7TDMI-S output **PROT[1:0]** for bus interface design.	**nOPC**
CPnTRANS	Active LOW supervisor mode access qualifier output. This is a coprocessor interface signal. Use the ARM7TDMI-S output **PROT[1:0]** for bus interface design.	**nTRANS**
CPSEQ	Sequential address signal. This is a coprocessor interface signal. Use the ARM7TDMI-S output **TRANS[1:0]** for bus interface design.	**SEQ**
CPTBIT	Instruction set qualifier output: 1 = THUMB instruction set. 0 = ARM instruction set.	**TBIT**
DBGACK	Debug acknowledge qualifier output: 1 = processor in debug state (real-time stopped). 0 = normal system state.	**DBGACK**
DBGBREAK	External breakpoint (tie LOW when not used).	**BREAKPT**
DBGCOMMRX	EmbeddedICE-RT communication channel receive buffer full output.	**COMMRX**
DBGCOMMTX	EmbeddedICE-RT communication channel transmit buffer empty output.	**COMMTX**
DBGEN	Debug enable. Tie this signal HIGH to be able to use the debug features of the ARM7TDMI.	**DBGEN**
DBGEXT[1:0]	EmbeddedICE-RT **EXTERN** debug qualifiers (tie LOW when not required).	**EXTERN0, EXTERN1**
DBGINSTRVALID [e]	Signals instruction execution to ETM7.	**INSTRVALID**
DBGnEXEC	Active LOW condition codes success at Execute stage.	**nEXEC**
DBGnTDOEN [f]	Active LOW TAP controller **DBGTDO** output qualifier.	**nTDOEN**

Table B-1 ARM7TDMI-S processor signals and ARM7TDMI hard macrocell equivalents (continued)

ARM7TDMI-S processor signal	Function	ARM7TDMI hard macrocell equivalent
DBGnTRST [f]	Active LOW TAP controller reset (asynchronous assertion). Resets the ICEBreaker subsystem.	**nTRST**
DBGRNG[1:0]	EmbeddedICE-RT rangeout qualifier outputs.	**RANGEOUT1**, **RANGEOUT0**
DBGRQ [g]	External debug request (tie LOW when not required).	**DBGRQ**
DBGTCKEN	Multi-ICE clock input qualifier sampled on the rising edge of **CLK**. Used to qualify **CLK** to enable the debug subsystem.	
DBGTDI [f]	Multi-ICE **TDI** test data input.	**TDI**
DBGTDO [f]	EmbeddedICE-RT TAP controller serial data output.	**TDO**
DBGTMS [f]	Multi-ICE **TMS** test mode select input.	**TMS**
DMORE	Asserted for `LDM` and `STM` instructions. No equivalent on the ARM7TDMI processor.	
LOCK [a]	Indicates whether the current address is part of locked access. This signal is generated by execution of a `SWP` instruction.	**LOCK**
nFIQ [h]	Active LOW fast interrupt request input.	**nFIQ**
nIRQ [h]	Active LOW interrupt request input.	**nIRQ**
nRESET	Active LOW reset input (asynchronous assertion). Resets the processor core subsystem.	**nRESET**
PROT[1:0] [a, i]	Protection output, indicates whether the current address is being accessed as instruction or data, and whether it is being accessed in a privileged mode or User mode.	**nOPC**, **nTRANS**
RDATA[31:0] [j]	Unidirectional 32-bit input data bus.	**DIN[31:0]**
SIZE[1:0]	Indicates the width of the bus transaction to the current address: 00 = 8-bit 01 = 16-bit 10 = 32-bit 11 = not supported.	**MAS[1:0]**

Table B-1 ARM7TDMI-S processor signals and ARM7TDMI hard macrocell equivalents (continued)

ARM7TDMI-S processor signal	Function	ARM7TDMI hard macrocell equivalent
TRANS[1:0]	Next transaction type output bus: 00 = address-only/idle transaction next 01 = coprocessor register transaction next 10 = non-sequential (new address) transaction next 11 = sequential (incremental address) transaction next.	**nMREQ**, **SEQ**
WDATA[31:0]	Unidirectional 32-bit output data bus	**DOUT[31:0]**
WRITE	Write access indicator.	**nRW**

a. All the address-class signals (**ADDR[31:0]**, **WRITE**, **SIZE[1:0]**, **PROT[1:0]**, and **LOCK**) change on the rising edge of **CLK**.
In a system with a low-frequency clock this means that it is possible for the signals to change in the first phase of the clock cycle. This is unlike the ARM7TDMI hard macrocell where they would always change in the last phase of the cycle.

b. **CLK** is a rising-edge clock. It is inverted with respect to the **MCLK** signal used on the ARM7TDMI hard macrocell.

c. **CLKEN** is sampled on the rising edge of **CLK**. The **nWAIT** signal on the ARM7TDMI hard macrocell must be held throughout the HIGH phase of **MCLK**. This means that the address-class outputs (**ADDR[31:0]**, **WRITE**, **SIZE[1:0]**, **PROT[1:0]**, and **LOCK**) might still change in a cycle in which **CLKEN** is taken LOW.
You must take this possibility into account when designing a memory system.

d. **CPA** and **CPB** are sampled on the rising edge of **CLK**. They can no longer change in the first phase of the next cycle, as is possible with the ARM7TDMI hard macrocell.

e. **DBGINSTRVALID** is implemented on the ARM7TDMI-S (Rev 3) and ARM7TDMI-S (Rev 4) soft core and ARM7TDMI (Rev 4) hard core macrocells. This siganl is not implemented on previous versions.

f. All JTAG signals are synchronous to **CLK** on the ARM7TDMI-S processor. There is no asynchronous **TCLK** as on the ARM7TDMI hard macrocell.
You can use an external synchronizing circuit to generate **TCLKEN** when an asynchronous **TCLK** is required.

g. **DBGRQ** must be synchronized externally to the macrocell. It is not an asynchronous input as on the ARM7TDMI hard macrocell.

h. **nFIQ** and **nIRQ** are synchronous inputs to the ARM7TDMI-S processor, and are sampled on the rising edge of **CLK**.
Asynchronous interrupts are not supported.

i. **PROT[0]** is the equivalent of **nOPC**, and **PROT[1]** is the equivalent of **nTRANS** on the ARM7TDMI hard macrocell.

j. The ARM7TDMI-S processor supports only unidirectional data buses, **RDATA[31:0]** and **WDATA[31:0]**. When a bidirectional bus is required, you must implement external bus combining logic.

B.2 ATPG scan interface

Where automatic scan path is inserted for automatic test pattern generation, three signals are instantiated on the macrocell interface:

- **SCANENABLE** is LOW for normal usage, HIGH for scan test
- **SCANIN** is the serial scan path input
- **SCANOUT** is the serial scan path output.

B.3 Timing parameters

The timing constraints have been adjusted to balance the external timing parameters with the area of the synthesized core. All inputs are sampled on the rising edge of **CLK**. The timing diagrams associated with these timing parameters are shown in *Timing diagrams* on page 8-2.

The clock enables are sampled on every rising clock edge:

- **CLKEN** setup time is $t_{isclken}$, hold time is $t_{ihclken}$
- **DBGTCKEN** setup time is $t_{istcken}$, hold time is $t_{ihtcken}$.

All other inputs are sampled on the rising edge of **CLK** when the clock enable is active HIGH:

- **ABORT** setup time is $t_{isabort}$, hold time is $t_{ihabort}$, when **CLKEN** is active
- **RDATA** setup time is $t_{isrdata}$, hold time is $t_{ihrdata}$, when **CLKEN** is active
- **DBGTMS**, **DBGTDI** setup time is t_{istctl}, hold time is t_{ihtctl}, when **DBGTCKEN** is active.

Outputs are all sampled on the rising edge of **CLK** with the appropriate clock enable active:

- **ADDR** output hold time is t_{ohaddr}, valid time is t_{ovaddr} when **CLKEN** is active
- **TRANS** output hold time is $t_{ohtrans}$, valid time is $t_{ovtrans}$ when **CLKEN** is active
- **LOCK**, **PROT**, **SIZE**, **WRITE** control output hold time is t_{ohctl}, valid time is t_{ovctl} when **CLKEN** is active
- **WDATA** output hold time is $t_{ohwdata}$, valid time is $t_{ovwdata}$ when **CLKEN** is active.

Similarly, all coprocessor and debug signal expansion signals are defined with input setup parameters of t_{is}... , hold parameters of t_{ih}... , output hold parameters of t_{oh}... and output valid parameters of t_{ov}... .

B.4 ARM7TDMI-S design considerations

When an ARM7TDMI hard macrocell design is being converted to the ARM7TDMI-S soft core, the following areas require special consideration:

- *Master clock*
- *JTAG interface timing*
- *TAP controller*
- *Interrupt timing*
- *Interrupt timing*.

B.4.1 Master clock

The master clock to the ARM7TDMI-S processor, **CLK**, is inverted with respect to **MCLK** used on the ARM7TDMI hard macrocell. The rising edge of the clock is the active edge of the clock, on which all inputs are sampled, and all outputs are causal.

B.4.2 JTAG interface timing

All JTAG signals on the ARM7TDMI-S processor are synchronous to the master clock input, **CLK**. When an external **TCLK** is used, use an external synchronizer to the ARM7TDMI-S processor.

B.4.3 TAP controller

The ARM7TDMI-S processor does not have a boundary scan chain. Consequently support for some JTAG instructions have been removed.

Optional JTAG specification instructions are:

- **CLAMP**
- **HIGHZ**
- **CLAMPZ**.

When scan chain 1 or scan chain 2 is selected, you can not use the **EXTEST**, **SAMPLE**, and **PRELOAD** instructions because:

- unpredictable behavior occurs
- instructions are only supported for designer added scan chains.

B.4.4 Interrupt timing

As with all ARM7TDMI-S processor signals, the interrupt signals **nIRQ** and **nFIQ** are sampled on the rising edge of **CLK**.

When you are converting an ARM7TDMI hard macrocell design where the **ISYNC** signal is asserted LOW, add a synchronizer to the design to synchronize the interrupt signals before they are applied to the ARM7TDMI-S processor.

B.4.5 Address-class signal timing

The address-class outputs (**ADDR[31:0]**, **WRITE**, **SIZE[1:0]**, **PROT[1:0]**, and **LOCK**) on the ARM7TDMI-S processor all change in response to the rising edge of **CLK**. This means that they can change in the first phase of the clock in some systems. When exact compatibility is required, add latches to the outside of the ARM7TDMI-S processor to make sure that they can change only in the second phase of the clock.

Because the **CLKEN** signal is sampled only on the rising edge of the clock, the address-class outputs still change in a cycle in which **CLKEN** is LOW. (This is similar to the behavior of **nMREQ** and **SEQ** in an ARM7TDMI hard macrocell system, when a wait state is inserted using **nWAIT**.) Make sure that the memory system design takes this into account.

Also make sure that the correct address is used for the memory cycle, even though **ADDR[31:0]** might have moved on to address for the next memory cycle.

For more details, see Chapter 3 *Memory Interface*.

B.4.6 ARM7TDMI signals not implemented on ARM7TDMI-S processor

The following ARM7TDMI signals are not implemented on the ARM7TDMI-S processor.

Table B-2 Unimplemented ARM7TDMI processor signals

Description	Signal name
Bus enables	**ABE** **DBE** **TBE**
BiDirectional data bus	**D**
Address timing control inputs	**ALE** **APE**
Byte latch controls	**BL**

Table B-2 Unimplemented ARM7TDMI processor signals

Description	Signal name
Data bus timing control signals	**BUSDIS** **BUSEN** **nENIN** **nENOUT** **nENOUTI**
Mode output	**nM**
Interrupt configuration signal	**ISYNC**
Debug signals	**DBGRQI** **ECLK**
JTAG expansion signals	**DRIVEBS** **ECAPCLK** **ECAPCLKBS** **HIGHZ** **ICAPCLKBS** **IR** **nHIGHZ** **PCLKBS** **RSTCLKBS** **SCREG** **SDINBS** **SDOUTBS** **SHCLKBS** **SHCLK2BS** **TAPSM** **TCK1** **TCK2**

For more details on any of these signals, see the *ARM7TDMI Technical Reference Manual*.

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

J

L

M

N

O

P

R

S

T

U

W

