

ARM7TDMI-S

Technical Reference Manual

ARM

ARM DDI 0084E

ARM7TDMI-S

Technical Reference Manual

© Copyright ARM Limited 1999. All rights reserved.

Release information

Change history

Date	Issue	Change
March 1999	E	Major edits, and sections rewritten.

Proprietary notice

ARM, the ARM Powered logo, Thumb and StrongARM are registered trademarks of ARM Limited.

The ARM logo, AMBA, Angel, ARMulator, EmbeddedICE, ModelGen, Multi-ICE, PrimeCell, ARM7TDMI, ARM7TDMI-S, ARM9TDMI, TDMI and STRONG are trademarks of ARM Limited.

All other products or services mentioned herein may be trademarks of their respective owners.

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

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

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

Document confidentiality status

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

Product status

The information in this document is Preliminary (information on a product under development).

ARM web address

<http://www.arm.com>

Preface

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

- *About this document* on page iv
- *Further reading* on page vii
- *Feedback* on page viii.

About this document

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

Intended audience

This document has been written for experienced hardware and software engineers who may or may 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.

Chapter 2 *Programmer's Model*

Read this chapter for a description of the programmer's model.

Chapter 3 *Memory Interface*

Read this chapter for a description of the memory interface.

Chapter 4 *Coprocessor Interface*

Read this chapter for a description of the coprocessor interface.

Chapter 5 *Debug Interface*

Read this chapter for a description of the debug interface.

Chapter 6 *Instruction Cycle Timings*

Read this chapter for instruction cycle timings.

Chapter 7 *AC Parameters*

Read this chapter for the AC parameters.

Appendix A *Signal Descriptions*

Read this chapter for a description of the ARM7TDMI-S signals.

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

Read this chapter for a description of the differences between the ARM7TDMI-S and the ARM7TDMI hard macrocell.

Appendix C *Implications of Removing the Debugger or 64-bit Multiply Support*

Read this chapter for details of the implications of removing the debugger or multiplier.

Appendix D *Debug in Depth*

Read this chapter for a detailed description of the debug interface.

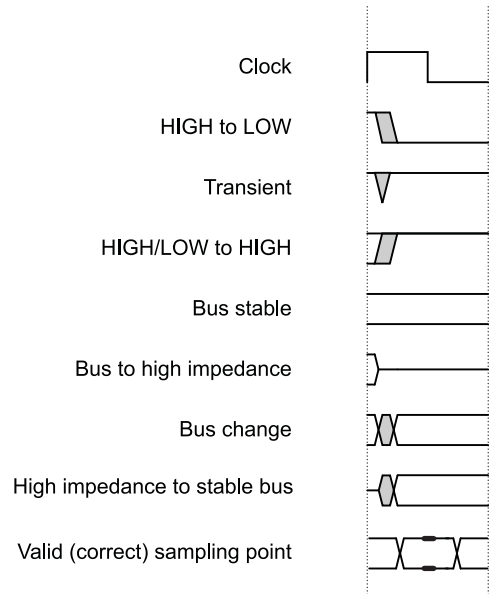
Typographical conventions

The following typographical conventions are used in this document:

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

Timing diagram conventions

This manual contains a number of timing diagrams. The following key explains the components used in these diagrams. Any variations are clearly labelled when they occur. Therefore, no additional meaning should 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

ARM Architecture Reference Manual (ARM DDI 0100).

ARM7TDMI Data Sheet (ARM DDI 0029).

Other publications

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

Feedback

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

If you have any problems with the ARM7TDMI-S, 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 stand-alone 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.

Contents

ARM7TDMI-S Technical Reference Manual

	Preface	
	About this document	iv
	Further reading.....	vii
	Feedback	viii
Chapter 1	Introduction	
	1.1 About the ARM7TDMI-S	1-2
	1.2 ARM7TDMI-S architecture	1-4
	1.3 ARM7TDMI-S block, core, and functional diagrams	1-6
	1.4 ARM7TDMI-S instruction set summary.....	1-9
Chapter 2	Programmer's Model	
	2.1 About the programmer's model.....	2-2
	2.2 Processor operating states	2-3
	2.3 Memory formats	2-4
	2.4 Instruction length.....	2-5
	2.5 Data types	2-6
	2.6 Operating modes	2-7
	2.7 Registers.....	2-8
	2.8 The program status registers	2-14
	2.9 Exceptions	2-17
	2.10 Interrupt latencies	2-24
	2.11 Reset.....	2-25

Chapter 3	Memory Interface	
3.1	About the memory interface	3-2
3.2	Bus interface signals	3-3
3.3	Bus cycle types.....	3-4
3.4	Addressing signals	3-10
3.5	Data timed signals	3-13
3.6	Use of CLKEN to control bus cycles.....	3-17
Chapter 4	Coprocessor Interface	
4.1	About coprocessors.....	4-2
4.2	Coprocessor interface signals	4-4
4.3	Pipeline following signals.....	4-5
4.4	Coprocessor interface handshaking	4-6
4.5	Connecting coprocessors	4-12
4.6	If you are not using an external coprocessor.....	4-14
4.7	Undefined instructions	4-15
4.8	Privileged instructions.....	4-16
Chapter 5	Debug Interface	
5.1	Overview of the debug interface.....	5-2
5.2	Debug systems.....	5-4
5.3	Debug interface signals	5-6
5.4	ARM7TDMI-S core clock domains	5-10
5.5	Determining the core and system state	5-11
5.6	Overview of EmbeddedICE	5-12
5.7	Disabling EmbeddedICE	5-14
5.8	The debug communications channel.....	5-15

Chapter 6	Instruction Cycle Timings	
6.1	Introduction to instruction cycle timings	6-3
6.2	Instruction cycle count summary.....	6-5
6.3	Branch and ARM branch with link.....	6-7
6.4	Thumb branch with link.....	6-8
6.5	Branch and exchange	6-9
6.6	Data operations.....	6-10
6.7	Multiply and multiply accumulate	6-12
6.8	Load register	6-14
6.9	Store register	6-16
6.10	Load multiple registers	6-17
6.11	Store multiple registers	6-19
6.12	Data swap	6-20
6.13	Software interrupt and exception entry	6-21
6.14	Coprocessor data processing operation	6-22
6.15	Load coprocessor register (from memory to coprocessor)	6-23
6.16	Store coprocessor register (from coprocessor to memory).....	6-25
6.17	Coprocessor register transfer (move from coprocessor to ARM register)...	6-27
6.18	Coprocessor register transfer (move from ARM register to coprocessor)...	6-28
6.19	Undefined instructions and coprocessor absent.....	6-29
6.20	Unexecuted instructions.....	6-30
Chapter 7	AC Parameters	
7.1	Timing diagrams	7-2
7.2	AC timing parameter definitions.....	7-7
Appendix A	Signal Descriptions	
A.1	Signal descriptions.....	A-2
Appendix B	Differences Between the ARM7TDMI-S and the ARM7TDMI	
B.1	Interface signals.....	B-2
B.2	ATPG scan interface.....	B-7
B.3	Timing parameters	B-8
B.4	ARM7TDMI-S design considerations.....	B-9
Appendix C	Implications of Removing the Debugger or 64-bit Multiply Support	
C.1	Implications of removing EmbeddedICE	C-2
C.2	Using MUL32	C-3

Appendix D

Debug in Depth

D.1	Scan chains and JTAG interface	D-3
D.2	Resetting the TAP controller.....	D-5
D.3	Instruction register	D-6
D.4	Public instructions.....	D-7
D.5	Test data registers.....	D-10
D.6	ARM7TDMI-S core clock domains	D-14
D.7	Determining the core and system state	D-15
D.8	Behavior of the program counter during debug	D-21
D.9	Priorities and exceptions	D-24
D.10	Scan interface timing	D-25
D.11	The watchpoint registers	D-27
D.12	Programming breakpoints	D-32
D.13	Programming watchpoints	D-34
D.14	The debug control register.....	D-35
D.15	The debug status register	D-36
D.16	Coupling breakpoints and watchpoints.....	D-38
D.17	Disabling EmbeddedICE	D-40
D.18	EmbeddedICE timing.....	D-41

Chapter 1

Introduction

This chapter introduces the ARM7TDMI-S:

- *About the ARM7TDMI-S* on page 1-2
- *ARM7TDMI-S architecture* on page 1-4
- *ARM7TDMI-S block, core, and functional diagrams* on page 1-6
- *ARM7TDMI-S instruction set summary* on page 1-9.

1.1 About the ARM7TDMI-S

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

A three-stage pipeline is used, so instructions are executed in three stages, fetch, decode and execute. This is shown in Figure 1-1.

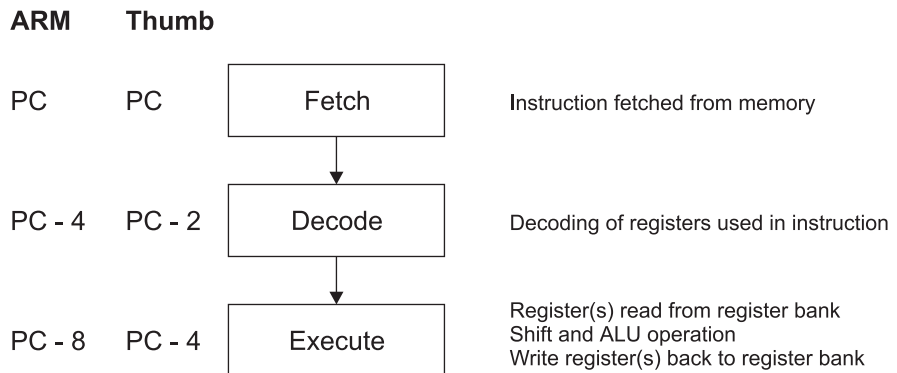


Figure 1-1 The instruction pipeline

———— **Note** ————

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

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

- idle 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 is an implementation of the ARMv4T architecture. For full details of both the ARM and Thumb instruction sets, refer to the *ARM Architecture Reference Manual*.

1.2.1 Instruction compression

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

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

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

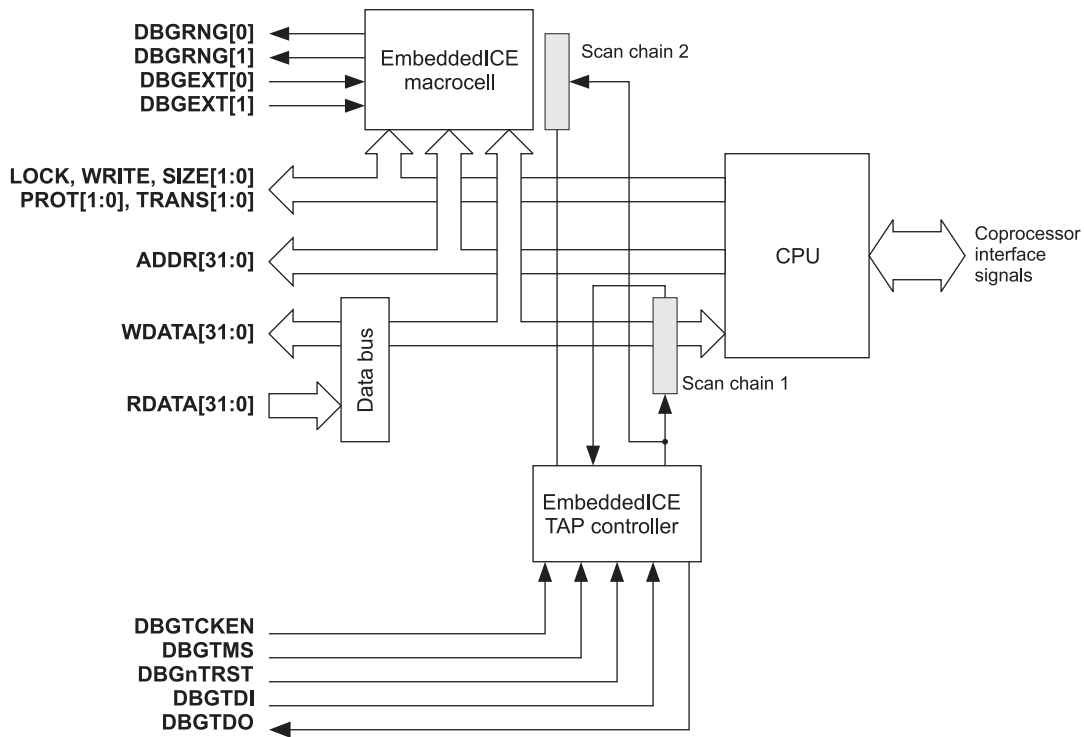


Figure 1-2 ARM7TDMI-S block diagram

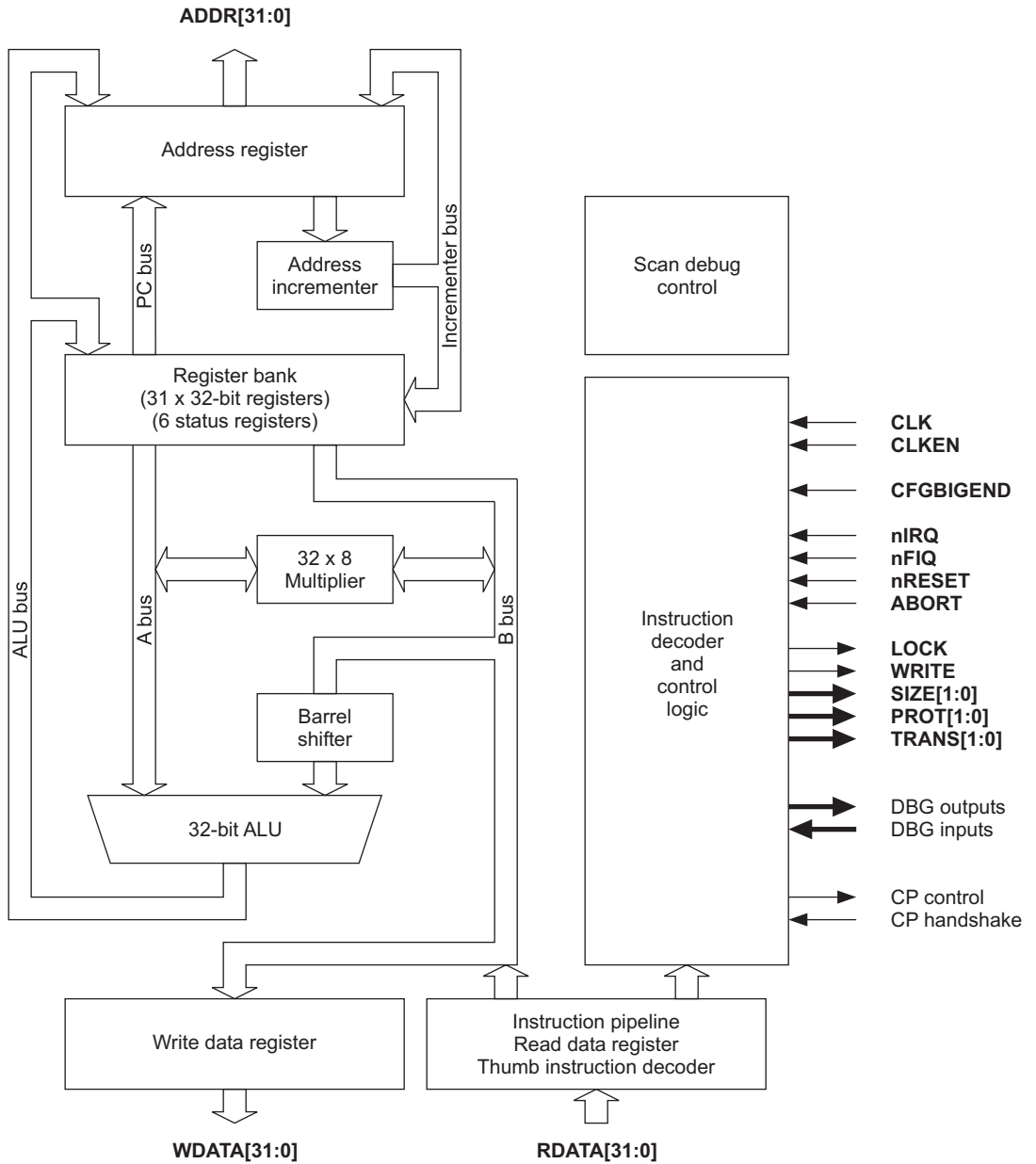


Figure 1-3 ARM7TDMI-S core

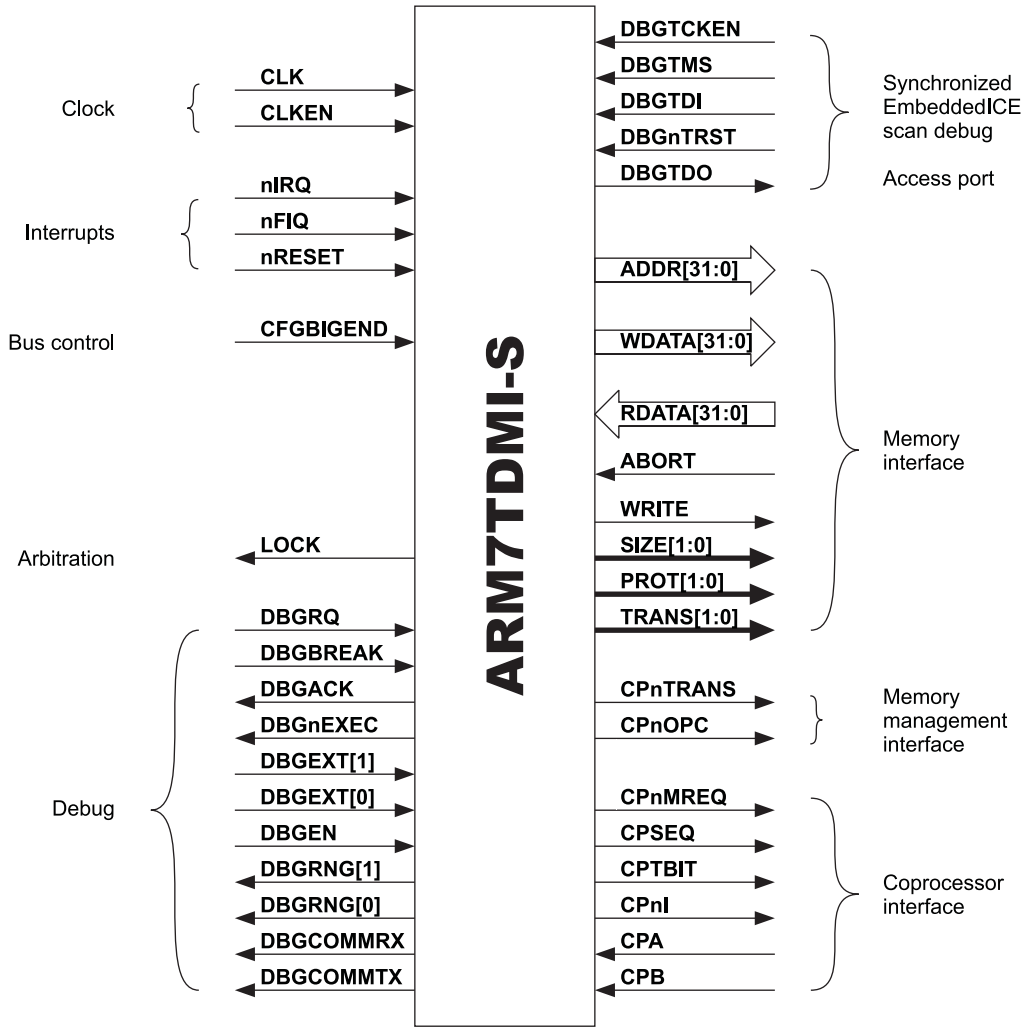


Figure 1-4 ARM7TDMI-S functional diagram

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 is an implementation of the ARMv4T architecture. For a complete description of both instruction sets, please refer to the *ARM Architecture Reference Manual*.

Table 1-1 Key to tables

	Description
{cond}	Refer to Table Condition Field {cond}
<Oprnd2>	Refer to Table Oprnd2
{field}	Refer to Table Field
S	Sets condition codes (optional)
B	Byte operation (optional)
H	Halfword operation (optional)
T	Forces address translation. Cannot be used with pre-indexed addresses
<a_mode2>	Refer to Table Addressing Mode 2
<a_mode2P>	Refer to Table Addressing Mode 2 (Privileged)
<a_mode3>	Refer to Table Addressing Mode 3
<a_mode4L>	Refer to Table Addressing Mode 4 (Load)
<a_mode4S>	Refer to Table Addressing Mode 4 (Store)
<a_mode5>	Refer to Table Addressing Mode 5
#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 given in Table 1-2.

Table 1-2 ARM instruction summary

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

Table 1-2 ARM instruction summary (continued)

Operation		Assembler	
Load	Word	LDR{cond} Rd, <a_mode2>	
	Word with user-mode privilege	LDR{cond}T Rd, <a_mode2P>	
	Byte	LDR{cond}B Rd, <a_mode2>	
	Byte with user-mode privilege	LDR{cond}BT Rd, <a_mode2P>	
	Byte signed	LDR{cond}SB Rd, <a_mode3>	
	Halfword	LDR{cond}H Rd, <a_mode3>	
	Halfword signed	LDR{cond}SH Rd, <a_mode3>	
	Multiple		
	Block data operations		
	Increment before	LDM{cond}IB Rd{!}, <reglist>{^}	
	Increment after	LDM{cond}IA Rd{!}, <reglist>{^}	
	Decrement before	LDM{cond}DB Rd{!}, <reglist>{^}	
	Decrement after	LDM{cond}DA Rd{!}, <reglist>{^}	
	Stack operations	LDM{cond}<a_mode4L> Rd{!}, <reglist>	
	Stack operations and restore CPSR	LDM{cond}<a_mode4L> Rd{!}, <reglist+pc>^	
	User registers	LDM{cond}<a_mode4L> Rd{!}, <reglist>^	
	Store	Word	STR{cond} Rd, <a_mode2>
Word with user-mode privilege		STR{cond}T Rd, <a_mode2P>	
Byte		STR{cond}B Rd, <a_mode2>	
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]

Table 1-2 ARM instruction summary (continued)

Operation	Assembler
Coprocessors	
Data operations	CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>
Move to ARM reg from coproc	MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>
Move to coproc from ARM reg	MCR{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>
Load	LDC{cond} p<cpnum>, CRd, <a_mode5>
Store	STC{cond} p<cpnum>, CRd, <a_mode5>
Software Interrupt	SWI 24bit_Imm

Addressing mode 2 is summarized in Table 1-3.

Table 1-3 Addressing mode 2

Addressing mode 2	
Immediate offset	[Rn, #+/-12bit_Offset]
Register offset	[Rn, +/-Rm]
Scaled register offset	[Rn, +/-Rm, LSL #5bit_shift_imm]
	[Rn, +/-Rm, LSR #5bit_shift_imm]
	[Rn, +/-Rm, ASR #5bit_shift_imm]
	[Rn, +/-Rm, ROR #5bit_shift_imm]
	[Rn, +/-Rm, RRX]
Pre-indexed offset	
Immediate	[Rn, #+/-12bit_Offset]!
Register	[Rn, +/-Rm]!
Scaled register	[Rn, +/-Rm, LSL #5bit_shift_imm]!
	[Rn, +/-Rm, LSR #5bit_shift_imm]!
	[Rn, +/-Rm, ASR #5bit_shift_imm]!
	[Rn, +/-Rm, ROR #5bit_shift_imm]!
	[Rn, +/-Rm, RRX]!
Post-indexed offset	
Immediate	[Rn], #+/-12bit_Offset
Register	[Rn], +/-Rm
Scaled register	[Rn], +/-Rm, LSL #5bit_shift_imm
	[Rn], +/-Rm, LSR #5bit_shift_imm
	[Rn], +/-Rm, ASR #5bit_shift_imm
	[Rn], +/-Rm, ROR #5bit_shift_imm
	[Rn], +/-Rm, RRX]

Addressing mode 2 (privileged) is summarized in Table 1-4.

Table 1-4 Addressing mode 2 (privileged)

Addressing mode 2 (privileged)	
Immediate offset	[Rn, #+/-12bit_Offset]
Register offset	[Rn, +/-Rm]
Scaled register offset	[Rn, +/-Rm, LSL #5bit_shift_imm]
	[Rn, +/-Rm, LSR #5bit_shift_imm]
	[Rn, +/-Rm, ASR #5bit_shift_imm]
	[Rn, +/-Rm, ROR #5bit_shift_imm]
	[Rn, +/-Rm, RRX]
Post-indexed offset	
Immediate	[Rn], #+/-12bit_Offset
Register	[Rn], +/-Rm
Scaled register	[Rn], +/-Rm, LSL #5bit_shift_imm
	[Rn], +/-Rm, LSR #5bit_shift_imm
	[Rn], +/-Rm, ASR #5bit_shift_imm
	[Rn], +/-Rm, ROR #5bit_shift_imm
	[Rn], +/-Rm, RRX]

Addressing mode 3 is summarized in Table 1-5.

Table 1-5 Addressing mode 3

Addressing mode 3 - signed byte and halfword data transfer	
Immediate offset	[Rn, #+/-8bit_Offset]
Pre-indexed	[Rn, #+/-8bit_Offset]!
Post-indexed	[Rn], #+/-8bit_Offset
Register	[Rn, +/-Rm]
Pre-indexed	[Rn, +/-Rm]!
Post-indexed	[Rn], +/-Rm

Addressing mode 4 (load) is summarized in Table 1-6.

Table 1-6 Addressing mode 4 (load)

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) is summarized in Table 1-7.

Table 1-7 Addressing mode 4 (store)

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 (load) is summarized in Table 1-8.

Table 1-8 Addressing mode 5

Addressing mode 5 - coprocessor data transfer	
Immediate offset	[Rn, #+/- (8bit_Offset*4)]
Pre-indexed	[Rn, #+/- (8bit_Offset*4)]!
Post-indexed	[Rn], #+/- (8bit_Offset*4)

Oprnd2 is summarized in Table 1-9.

Table 1-9 Oprnd2

Oprnd2	
Immediate value	#32bit_Imm
Logical shift left	Rm LSL #5bit_Imm
Logical shift right	Rm LSR #5bit_Imm
Arithmetic shift right	Rm ASR #5bit_Imm
Rotate right	Rm ROR #5bit_Imm
Register	Rm
Logical shift left	Rm LSL Rs
Logical shift right	Rm LSR Rs
Arithmetic shift right	Rm ASR Rs
Rotate right	Rm ROR Rs
Rotate right extended	Rm RRX

Fields are summarized in Table 1-10.

Table 1-10 Fields

Field	
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 are summarized in Table 1-11.

Table 1-11 Condition fields

Condition field {cond}	
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 given in Table 1-12.

Table 1-12 Thumb instruction summary

Operation		Assembler
Move	Immediate	MOV Rd, #8bit_Imm
	High to Low	MOV Rd, Hs
	Low to High	MOV Hd, Rs
	High to High	MOV Hd, Hs
Arithmetic	Add	ADD Rd, Rs, #3bit_Imm
	Add Low and Low	ADD Rd, Rs, Rn
	Add High to Low	ADD Rd, Hs
	Add Low to High	ADD Hd, Rs
	Add High to High	ADD Hd, Hs
	Add Immediate	ADD Rd, #8bit_Imm
	Add Value to SP	ADD SP, #7bit_Imm ADD SP, #-7bit_Imm
	Add with carry	ADC Rd, Rs
	Subtract	SUB Rd, Rs, Rn SUB Rd, Rs, #3bit_Imm
	Subtract Immediate	SUB Rd, #8bit_Imm
	Subtract with carry	SBC Rd, Rs
	Negate	NEG Rd, Rs
	Multiply	MUL Rd, Rs
	Compare Low and Low	CMP Rd, Rs
	Compare Low and High	CMP Rd, Hs
	Compare High and Low	CMP Hd, Rs
	Compare High and High	CMP Hd, Hs
Compare Negative	CMN Rd, Rs	
Compare Immediate	CMP Rd, #8bit_Imm	
Logical	AND	AND Rd, Rs
	EOR	EOR Rd, Rs
	OR	ORR Rd, Rs
	Bit clear	BIC Rd, Rs
	Move NOT	MVN Rd, Rs
	Test bits	TST Rd, Rs

Table 1-12 Thumb instruction summary (continued)

Operation	Assembler
Shift/Rotate	Logical shift left LSL Rd, Rs, #5bit_shift_imm
	Logical shift right LSR Rd, Rs, #5bit_shift_imm
	Arithmetic shift right ASR Rd, Rs, #5bit_shift_imm
	Rotate right ROR Rd, Rs
Branch	Conditional
	if Z set BEQ label
	if Z clear BNE label
	if C set BCS label
	if C clear BCC label
	if N set BMI label
	if N clear BPL label
	if V set BVS label
	if V clear BVC label
	if C set and Z clear BHI label
	if C clear and Z set BLS label
	if N set and V set, or if N clear and V clear BGE label
	if N set and V clear, or if N clear and V set BLT label
	if Z clear, and N or V set, or if Z clear, and N or V clear BGT label
	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	

Table 1-12 Thumb instruction summary (continued)

Operation	Assembler
word	LDR Rd, [Rb, Ro]
halfword	LDRH Rd, [Rb, Ro]
signed halfword	LDRSH Rd, [Rb, Ro]
byte	LDRB Rd, [Rb, Ro]
signed byte	LDRSB Rd, [Rb, Ro]
PC-relative	LDR Rd, [PC, #10bit_Offset]
SP-relative	LDR Rd, [SP, #10bit_Offset]
Address	
using PC	ADD Rd, PC, #10bit_Offset
using SP	ADD Rd, SP, #10bit_Offset
Multiple	LDMIA Rb!, <reglist>
Store	With immediate offset
word	STR Rd, [Rb, #7bit_offset]
halfword	STRH Rd, [Rb, #6bit_offset]
byte	STRB Rd, [Rb, #5bit_offset]
	With register offset
word	STR Rd, [Rb, Ro]
halfword	STRH Rd, [Rb, Ro]
byte	STRB Rd, [Rb, Ro]
SP-relative	STR Rd, [SP, #10bit_offset]
Multiple	STMIA Rb!, <reglist>
Push/Pop	Push registers onto stack
	PUSH <reglist>
	Push LR and registers onto stack
	PUSH <reglist, LR>
	Pop registers from stack
	POP <reglist>
	Pop registers and PC from stack
	POP <reglist, PC>
Software Interrupt	SWI 8bit_Imm

Chapter 2

Programmer's Model

This chapter describes the ARM7TDMI-S programmer's model:

- *About the programmer's model* on page 2-2
- *Processor operating states* on page 2-3
- *Memory formats* on page 2-4
- *Instruction length* on page 2-5
- *Data types* on page 2-6
- *Operating modes* on page 2-7
- *Registers* on page 2-8
- *The program status registers* on page 2-14
- *Exceptions* on page 2-17
- *Interrupt latencies* on page 2-24
- *Reset* on page 2-25.

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 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

The operating state of the ARM7TDMI-S core can be switched 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 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, and so on.

The ARM7TDMI-S can treat words in memory as being stored in either:

- big-endian format
- little-endian format.

2.3.1 Big-endian format

In big-endian format, the ARM7TDMI-S 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 through 24. This is shown in Figure 2-1:

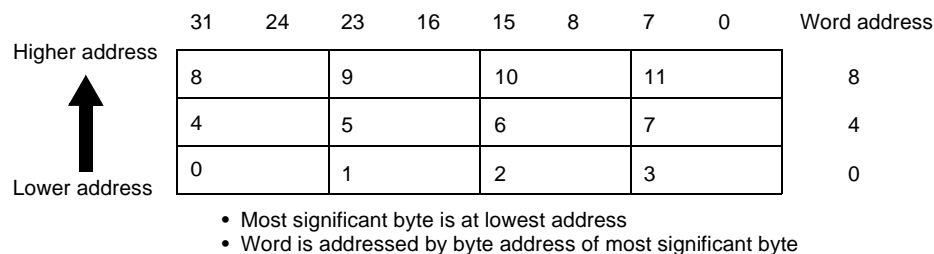


Figure 2-1 Little-endian addresses of bytes within words

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 through 0. This is shown in Figure 2-2:

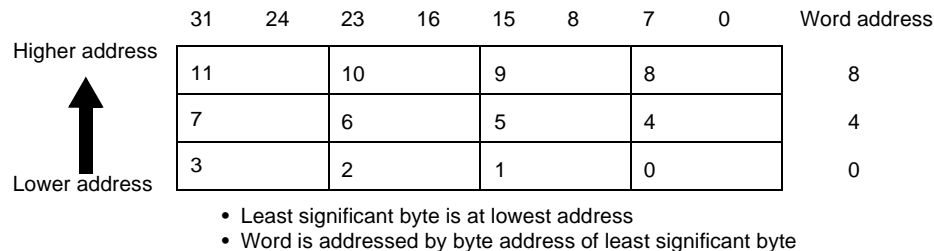


Figure 2-2 Big-endian addresses of bytes within words

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 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 has seven modes of operation:

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

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

2.7 Registers

The ARM7TDMI-S 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 2-10 shows which registers are available in each mode.

The ARM-state register set contains 16 directly-accessible registers, r0 to r15. A further register, the *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, r 15 and the CPSR have the following special functions:

Link register	<p>Register 14 is used as the subroutine <i>link register</i> (LR). r14 receives a copy of r15 when a <i>Branch with Link</i> (BL) instruction is executed.</p> <p>At all other times r14 can be treated as a general-purpose register. The corresponding banked registers r14_svc, r14_irq, r14_fiq, r14_abt and r14_und are similarly used to hold the return values of r15 when interrupts and exceptions arise, or when BL instructions are executed within interrupt or exception routines.</p>
Program counter	<p>Register 15 holds the PC.</p> <p>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.</p>

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

Banked registers have a mode identifier which 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





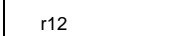
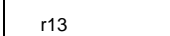
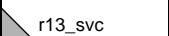
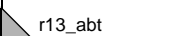
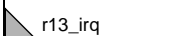
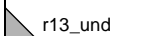


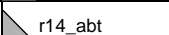
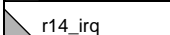
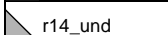
FIQ mode has seven banked registers mapped to r8–r14 (r8_fiq–r14_fiq).

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

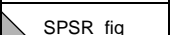
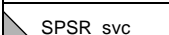
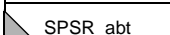
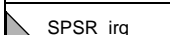
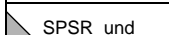
The user, IRQ, supervisor, abort, and undefined modes each have two banked registers mapped to r13 and r14, allowing a private stack pointer and link register for each mode.

Figure 2-3 shows the ARM-state registers.

ARM-state general registers and program counter

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

ARM-state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR_fiq	 SPSR_svc	 SPSR_abt	 SPSR_irq	 SPSR_und

 = banked register

Figure 2-3 Register organization in ARM state











2.7.2 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)
- an LR
- the CPSR.

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

Thumb-state general registers and program counter

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

Thumb-state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	 SPSR_fiq	 SPSR_svc	 SPSR_abt	 SPSR_irq	 SPSR_und

 = banked register

Figure 2-4 Register organization in Thumb state

2.7.3 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.

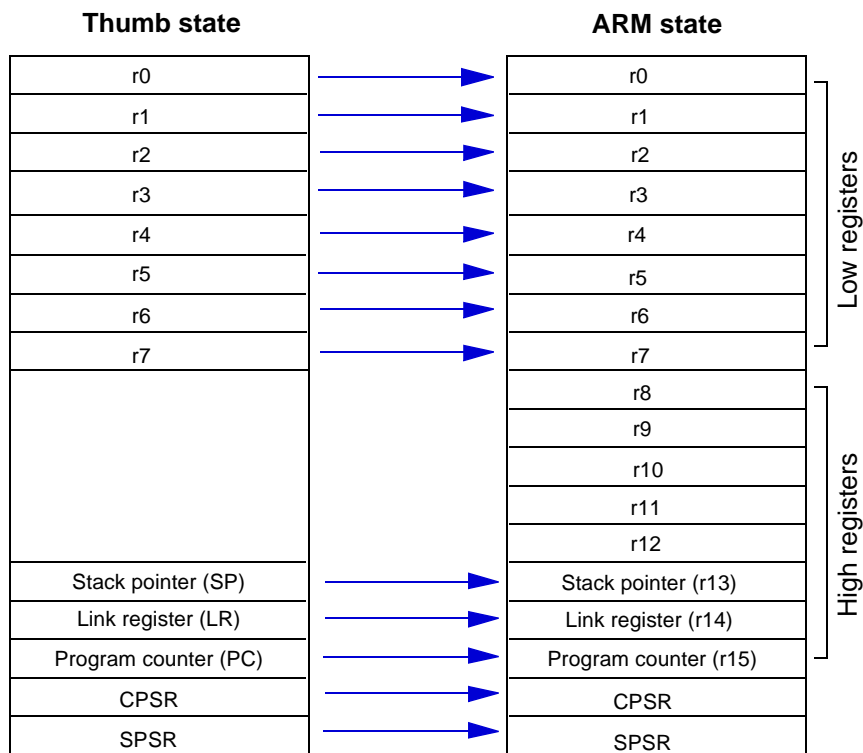


Figure 2-5 Mapping of Thumb-state registers onto ARM-state registers

———— **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 allows you to compare high register values with low register values. The ADD instruction allows you to add high register values to low register values. For more details, please refer to the *ARM Architecture Reference Manual*.

2.8 The program status registers

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

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

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

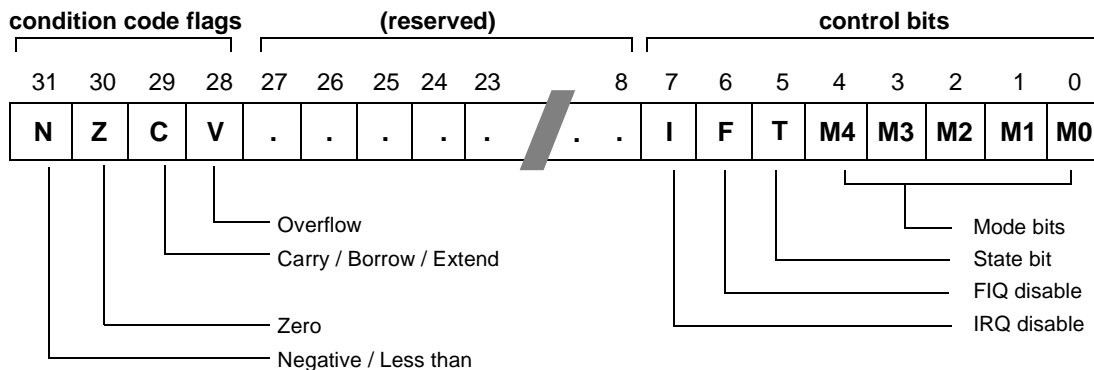


Figure 2-6 Program status register format

———— **Note** ————

To maintain compatibility with future ARM processors, and as good practise, 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, and can be set by arithmetic and logical operations. They can also be set by MSR and LDM instructions. The ARM7TDMI-S tests these flags to determine whether to execute an instruction.

All instructions can execute conditionally in ARM state. In Thumb state, only the Branch instruction can be executed conditionally. For more information about conditional execution, refer to the *ARM Architecture Reference Manual*.

2.8.2 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
11111	System	r0–r7, SP, LR, PC, CPSR	r0–r14, PC, CPSR

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

2.8.3 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 may 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 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 2-23.

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

- *Exception entry/exit summary*
- *Entering an exception* on page 2-18
- *Leaving an exception* on page 2-18.

2.9.1 Exception entry/exit summary

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

Table 2-3 Exception entry/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, or undefined instruction fetch, that had the prefetch abort.
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	
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	
DABT	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	NA	-	-	The value saved in r14_svc upon reset is UNPREDICTABLE.

2.9.2 Entering an exception

When handling an exception the ARM7TDMI-S:

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) that will cause the program to resume from the correct place on return.

The exception handler does not need 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 may also set the interrupt disable flags 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 2-17.
2. Copy the SPSR back to the CPSR.
3. Clear the interrupt disable flags that were set on entry.

———— **Note** —————

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

—————

2.9.4 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 (thus minimizing the overhead of context switching).

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

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

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

FIQ exceptions may be disabled 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. An abort is signalled 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 marks the prefetched instruction as invalid, but does not take the exception until the instruction reaches the execute stage of the pipeline. If the instruction is not executed, for example because it fails its condition codes, or because a branch occurs while it is in the pipeline, the abort does not take place.

After dealing with the reason for the abort, the handler executes the following instruction irrespective of the processor operating state:

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

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

Data abort

When a data abort occurs, the action taken depends on the instruction type:

- Single data transfer instructions (LDR, 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 prevents all register overwriting after an abort is indicated, which means that the ARM7TDMI-S always preserves r15 (always the last register to be transferred) in an aborted LDM instruction.

The abort mechanism allows the implementation of a demand-paged virtual memory system. In such a system, the processor is allowed to generate arbitrary addresses. When the data at an address is unavailable, the *Memory Management Unit* (MMU) signals an abort. The abort handler must then work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

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 instruction* (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:

```
MOV SPC, 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 encounters an instruction that neither it, nor any coprocessor in the system can handle, the ARM7TDMI-S 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 is fully compliant with the ARM Instruction Set Architecture version 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, refer to 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 and 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 enters the data abort handler, and proceeds immediately to the FIQ vector.
A normal return from the FIQ causes the data abort handler to resume execution. Data aborts must have higher priority than FIQs to ensure that the transfer error does not escape detection. You must add the time for this exception entry to the worst-case FIQ latency calculations in a system that uses aborts to support virtual memory.

2.10 Interrupt latencies

The calculations for maximum and minimum latency are described below.

2.10.1 Maximum interrupt latencies

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

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

The total latency is therefore 27 processor cycles, just under 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, could 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 a reset occurs, the **nRESET** signal goes LOW, and the ARM7TDMI-S abandons the executing instruction.

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

1. Forces M[4:0] to 10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR T bit.
2. Forces the PC to fetch the next instruction from address 0x00.
3. 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 ARM7TDMI-S memory interface:

- *About the memory interface* on page 3-2
- *Bus interface signals* on page 3-3
- *Bus cycle types* on page 3-4
- *Addressing signals* on page 3-10
- *Data timed signals* on page 3-13
- *Use of CLKEN to control bus cycles* on page 3-17.

3.1 About the memory interface

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

- nonsequential
- sequential
- internal
- coprocessor register transfer.

3.2 Bus interface signals

The signals in the ARM7TDMI-S 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 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 *Use of 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 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.

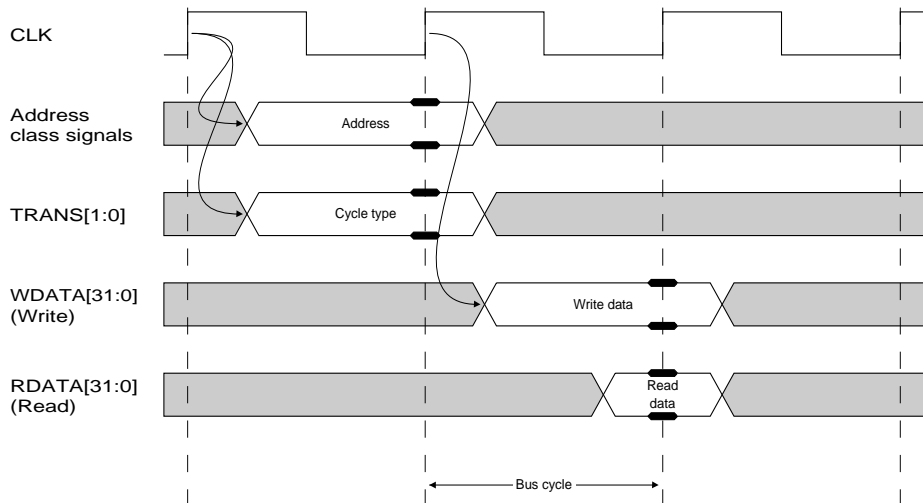


Figure 3-1 Simple memory cycle

The ARM7TDMI-S 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 should commit to a memory access only on an N cycle or an S cycle.

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

- a nonsequential cycle, during which the ARM7TDMI-S core requests a transfer to or from an address which is unrelated to the address used in the preceding cycle
- a sequential cycle, during which the ARM7TDMI-S core requests a transfer to or from an address which is either one word, or one halfword greater than the address used in the preceding cycle
- an internal cycle, during which 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
- a coprocessor register transfer cycle, during which 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 bus cycle, and occurs when the ARM7TDMI-S requests a transfer to or from an address which 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.

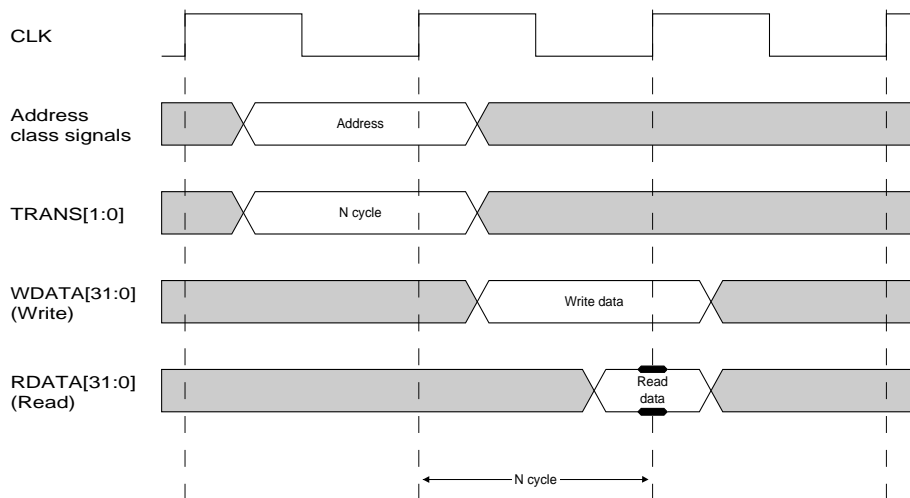


Figure 3-2 Nonsequential memory cycle

The ARM7TDMI-S 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, and your memory system is unable to cope with this case, use the **CLKEN** signal to extend the bus cycle to allow sufficient cycles for the memory system. See *Use of CLKEN to control bus cycles* on page 3-17.

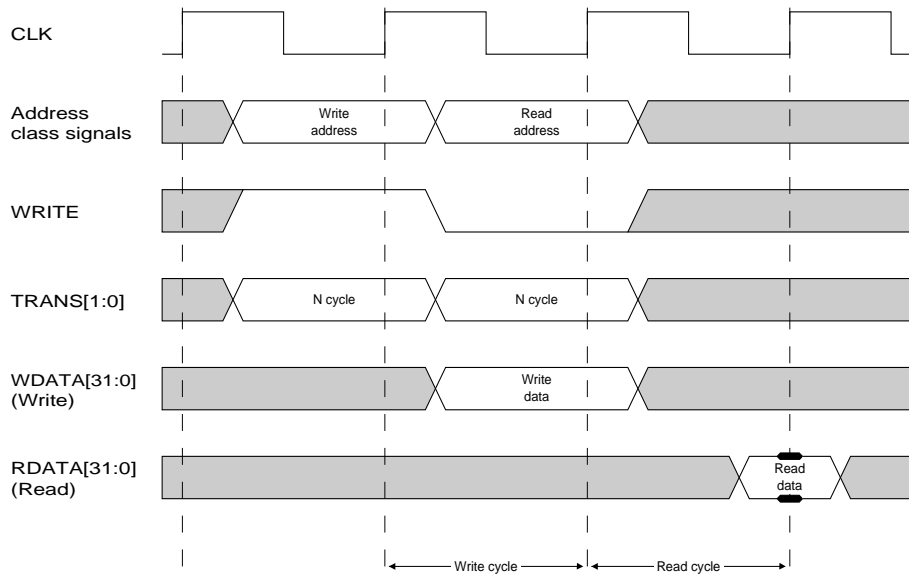


Figure 3-3 Back to back memory cycles

3.3.2 Sequential cycles

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

During a sequential cycle, the ARM7TDMI-S requests a memory location which is part of a sequential burst. If this is the first cycle in the burst, the address may 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 access, 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 are unaffected by a burst.

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

Table 3-2 Burst types

Burst type	Address increment	Cause
Word read	4 bytes	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.

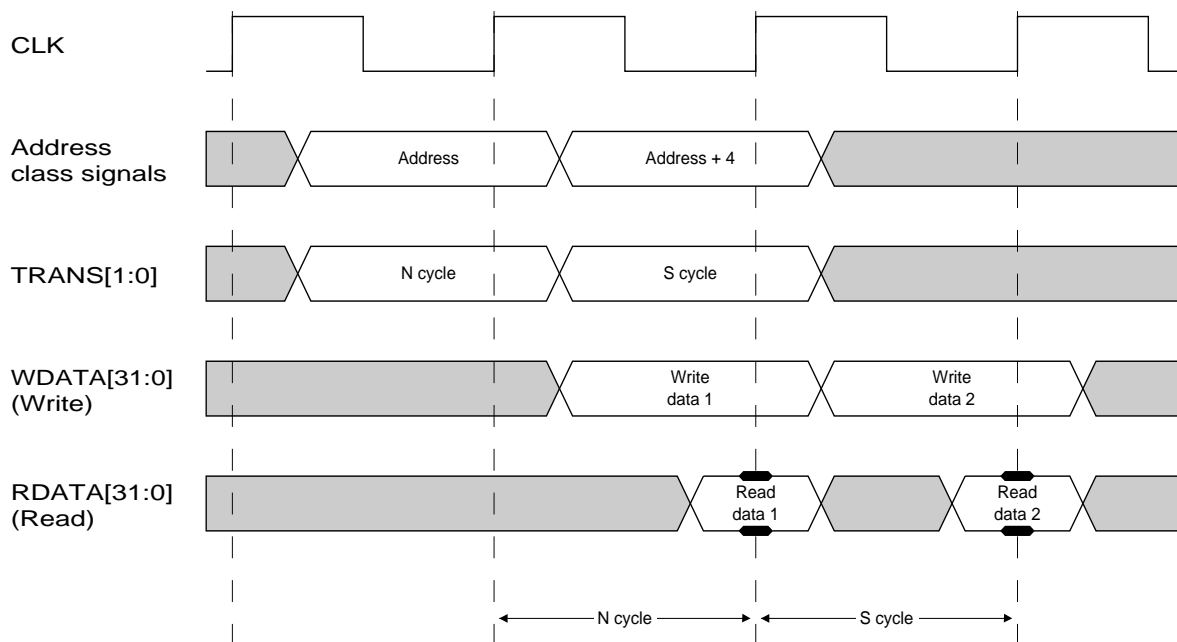


Figure 3-4 Sequential access cycles

3.3.3 Internal cycles

During an internal cycle, the ARM7TDMI-S 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 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 further described in *Merged I-S cycles*, below.

3.3.4 Merged I-S cycles

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

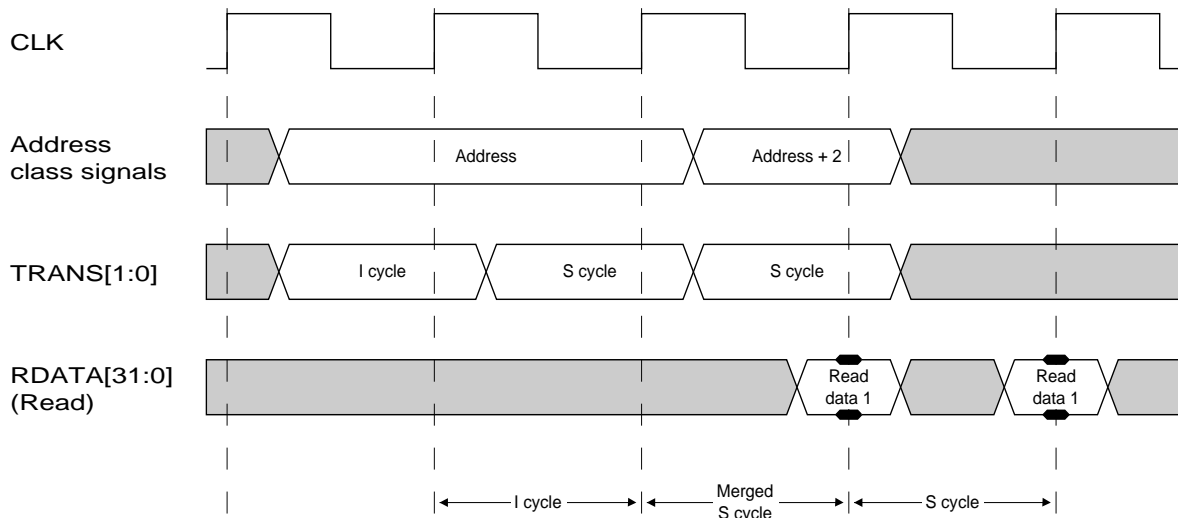


Figure 3-5 Merged I-S cycle

Note

When designing a memory controller, make sure that the design will also work when an I cycle is followed by an N cycle to a different address. This sequence may occur during exceptions, or during writes to the program counter. 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 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:

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

These are described below.

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 4 for each cycle.

The address bus provides a 4GB of linear addressing space. When a word access is signalled the memory system should ignore the bottom two bits, **ADDR[1:0]**, and when a halfword access is signalled the memory system should ignore the bottom bit, **ADDR[0]**.

3.4.2 WRITE

WRITE specifies the direction of the transfer. **WRITE** indicates an ARM7TDMI-S write cycle when HIGH, and an ARM7TDMI-S read cycle when LOW. A burst of S cycles is always either a read burst, or a write burst, because 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 can transfer word, halfword, and byte quantities. This is encoded on **SIZE[1:0]** as shown in Table 3-3.

Table 3-3 Transfer widths

SIZE[1:0]	Transfer width
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 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 may not be possible to use either of these capabilities.

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 signals can therefore be used to implement an access permission scheme. The encoding of **PROT[1:0]** is as shown in Table 3-4.

Table 3-4 PROT encoding

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

3.4.5 LOCK

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

3.4.6 CPTBIT

CPTBIT is used to indicate the operating state of the ARM7TDMI-S. When in:

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

3.5 Data timed signals

The data timed signals are:

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

These are described below.

3.5.1 WDATA[31:0]

WDATA[31:0] is the write data bus. All data written out from the ARM7TDMI-S is broadcast on this bus. Data transfers from the ARM7TDMI-S 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 value on **WDATA[31:0]** 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 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.

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 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 2-19.

3.5.4 Byte and halfword accesses

The ARM7TDMI-S 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 based system should support 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 is always a byte address. However, the memory system should ignore the bottom bits of the address. The significant address bits are listed 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 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 are as 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, make sure that the data is presented to the correct byte lanes on the ARM7TDMI-S as shown in Table 3-8 and Table 3-9 below.

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]

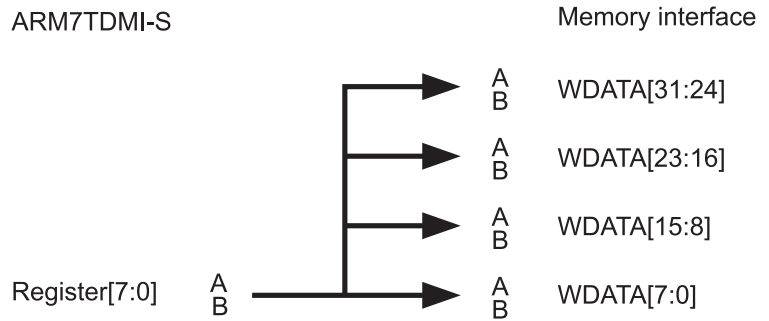
Table 3-9 Byte accesses

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 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.

Byte writes



Halfword writes

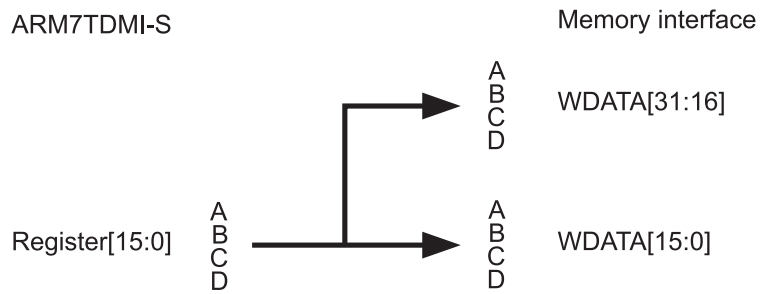


Figure 3-6 Data replication

3.6 Use of CLKEN to control bus cycles

The pipelined nature of the ARM7TDMI-S 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 may 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.

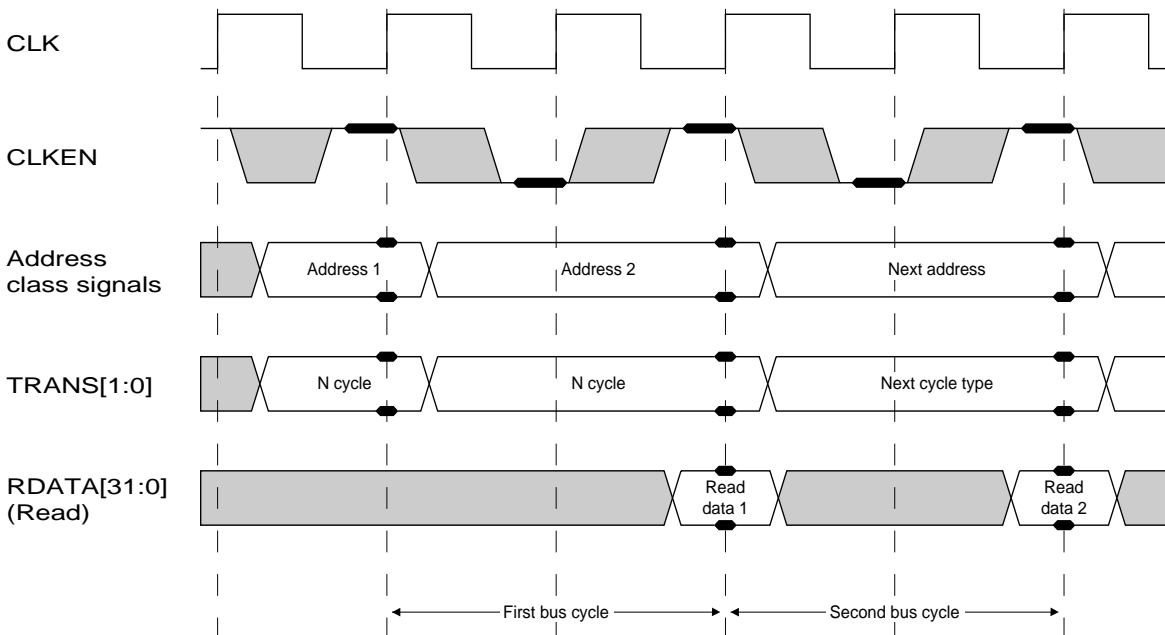


Figure 3-7 Use of CLKEN

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 will ensure 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:

- *About coprocessors* on page 4-2
- *Coprocessor interface signals* on page 4-4
- *Pipeline following signals* on page 4-5
- *Coprocessor interface handshaking* on page 4-6
- *Connecting coprocessors* on page 4-12
- *If you are not using an external coprocessor* on page 4-14
- *Undefined instructions* on page 4-15
- *Privileged instructions* on page 4-16.

4.1 About coprocessors

The ARM7TDMI-S instruction set allows specialized additional instructions to be implemented using coprocessors. These are separate processing units which are tightly coupled to the ARM7TDMI-S processor. A typical coprocessor contains:

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

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

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

The ARM7TDMI-S:

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

The coprocessor:

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

Up to 16 coprocessors can be connected into a system, each with a unique coprocessor ID number to identify it. The ARM7TDMI-S contains two internal coprocessors:

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

External coprocessors, therefore, cannot be assigned to coprocessor numbers 14 or 15. Other coprocessor numbers have also been reserved by ARM. Coprocessor availability is listed 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 email with `coprocessor` in the subject line to `info@arm.com` for up-to-date information on which coprocessor numbers have been allocated.

4.2 Coprocessor interface signals

The signals used to interface the ARM7TDMI-S 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:

- **CPnCPI**
- **CPA**
- **CPB**

The data signals are:

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

These signals and their use are discussed in the rest of this chapter.

4.3 Pipeline following signals

Every coprocessor in the system must contain a pipeline follower to track the instructions executing in the ARM7TDMI-S pipeline. The coprocessors connect to the ARM7TDMI-S 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 which will 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 ARM7TDMI-S state opcode fetch, so the new opcode must be sampled into the pipeline.

- The pipeline should 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 pipeline will never signal on **CPnCPI** 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 ARM7TDMI-S instructions.

4.4 Coprocessor interface handshaking

The ARM7TDMI-S and any coprocessors in the system perform a handshake using the following signals:

Table 4-2 Handshaking signals

Signal	Direction	Meaning
CPnCPI	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 signalling* 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 which 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 using **CPA** and **CPB**.

4.4.2 The ARM7TDMI-S

Coprocessor instructions progress down the ARM7TDMI-S 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 may not have if it was preceded by a branch.)
2. The instruction has passed its conditional execution tests.
3. A coprocessor in the system has signalled on **CPA** and **CPB** that it is able to accept the instruction.

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

4.4.3 Coprocessor signalling

The coprocessor signals as follows:

Coprocessor absent If a coprocessor cannot accept the instruction currently in decode it should leave **CPA** and **CPB** both HIGH.

Coprocessor present If a coprocessor can accept an instruction, and can start that instruction immediately, it should 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 by asserting busy-wait. This is signalled 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.

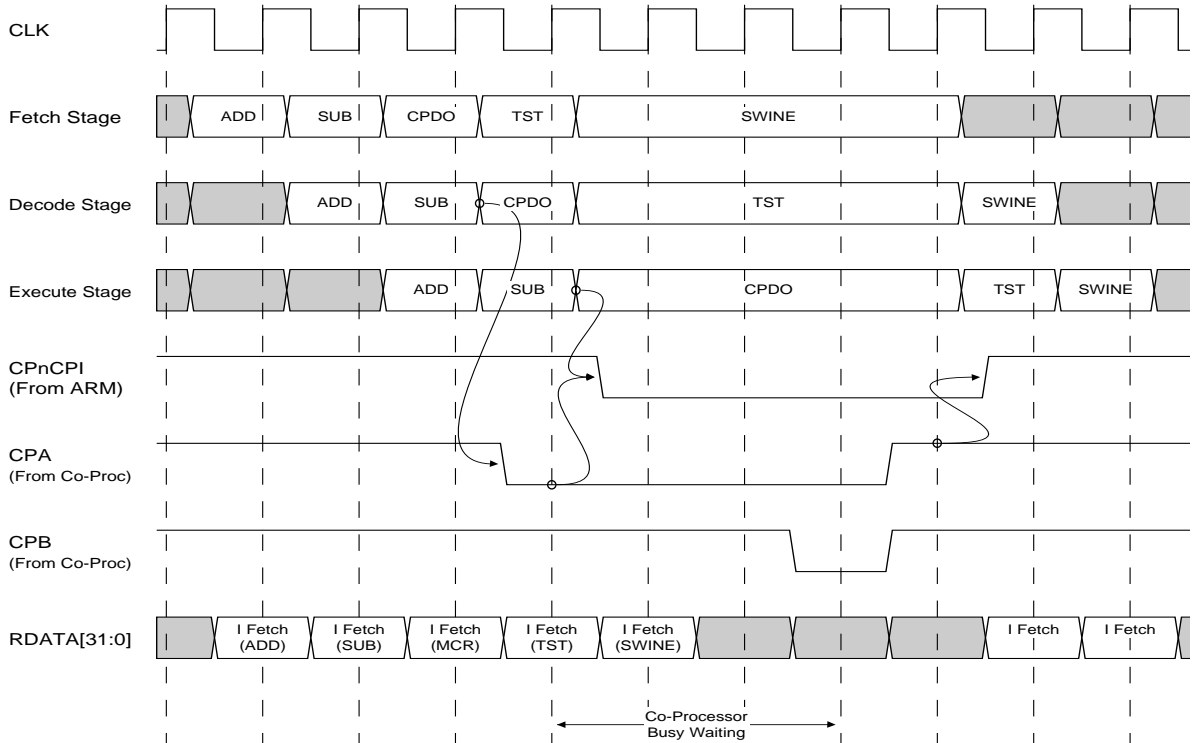


Figure 4-1 Coprocessor busy-wait sequence

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 set in the CSPR), the ARM7TDMI-S abandons the coprocessor instruction, and signals this by taking **nCPI HIGH**. A coprocessor which is capable of busy-waiting must monitor **nCPI** to detect this condition. When the ARM7TDMI-S abandons a coprocessor instruction, the coprocessor also abandons the instruction, and continues tracking the ARM7TDMI-S 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 once the instruction has been executed.

4.4.5 Coprocessor register transfer instructions

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

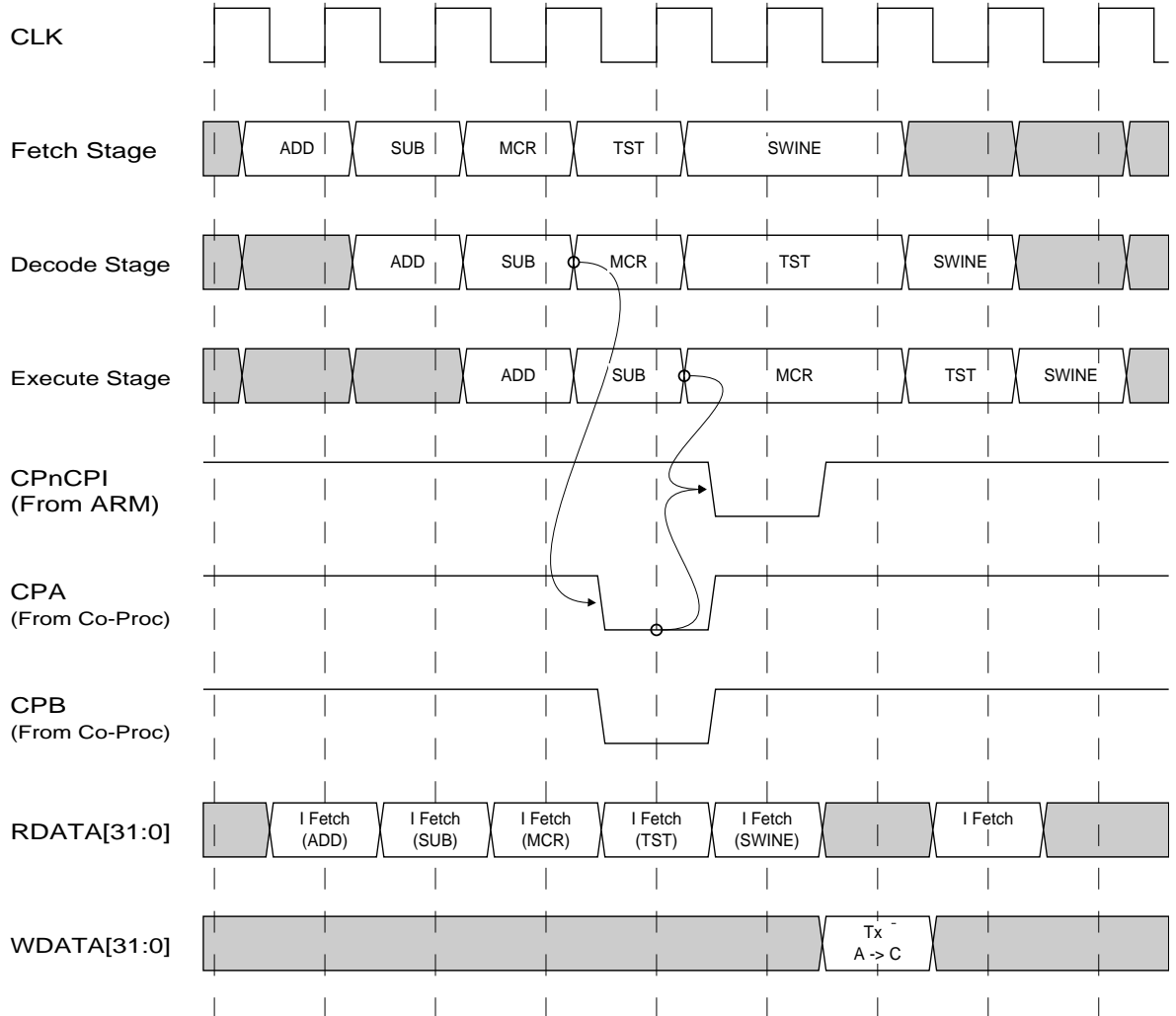


Figure 4-2 Coprocessor register transfer sequence

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

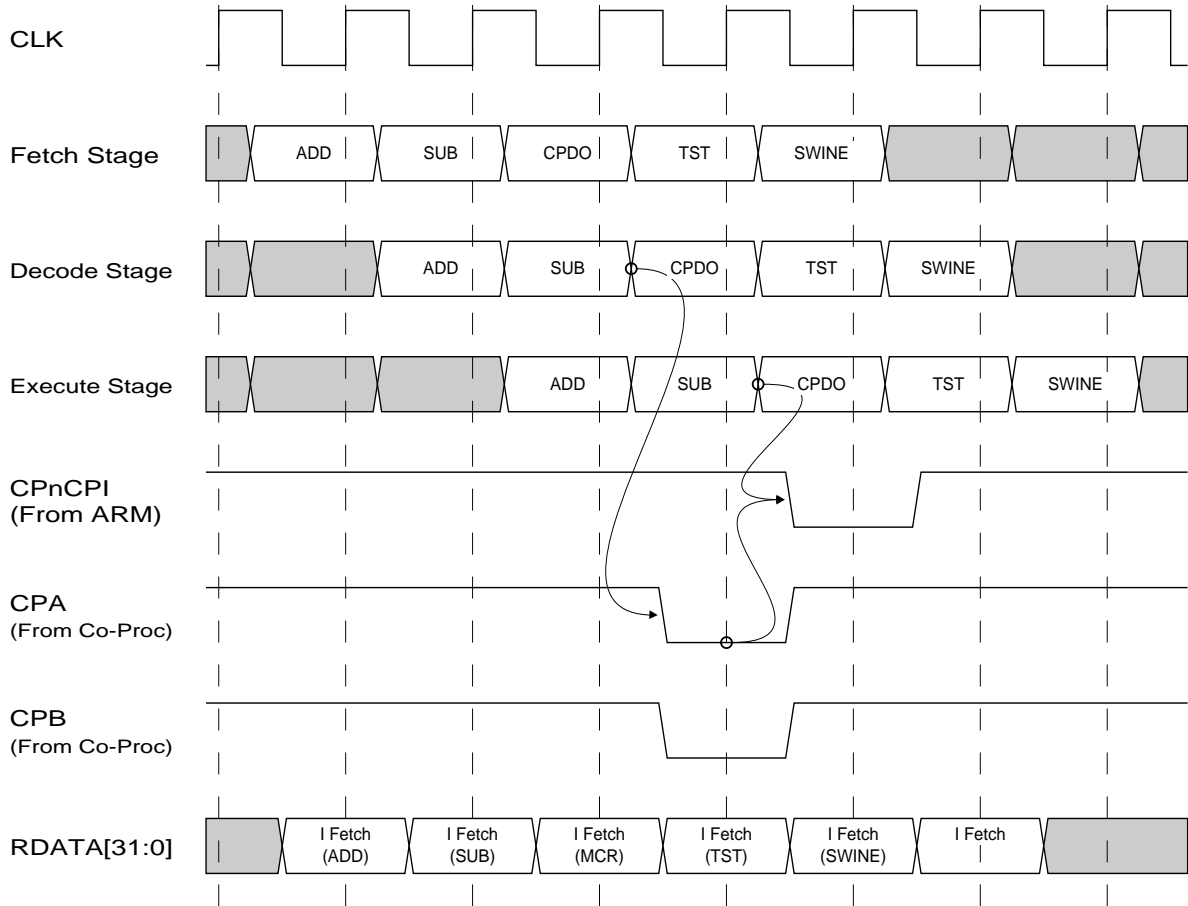


Figure 4-3 Coprocessor data operation sequence

4.4.7 Coprocessor load and store operations

The coprocessor load and store instructions are used to transfer data between a coprocessor and memory. They can be used to transfer either a single word of data, or a number of the coprocessor registers. There is no limit to the number of words of data that can be transferred by a single LDC or STC instruction, but by convention no coprocessor should 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 will increase.

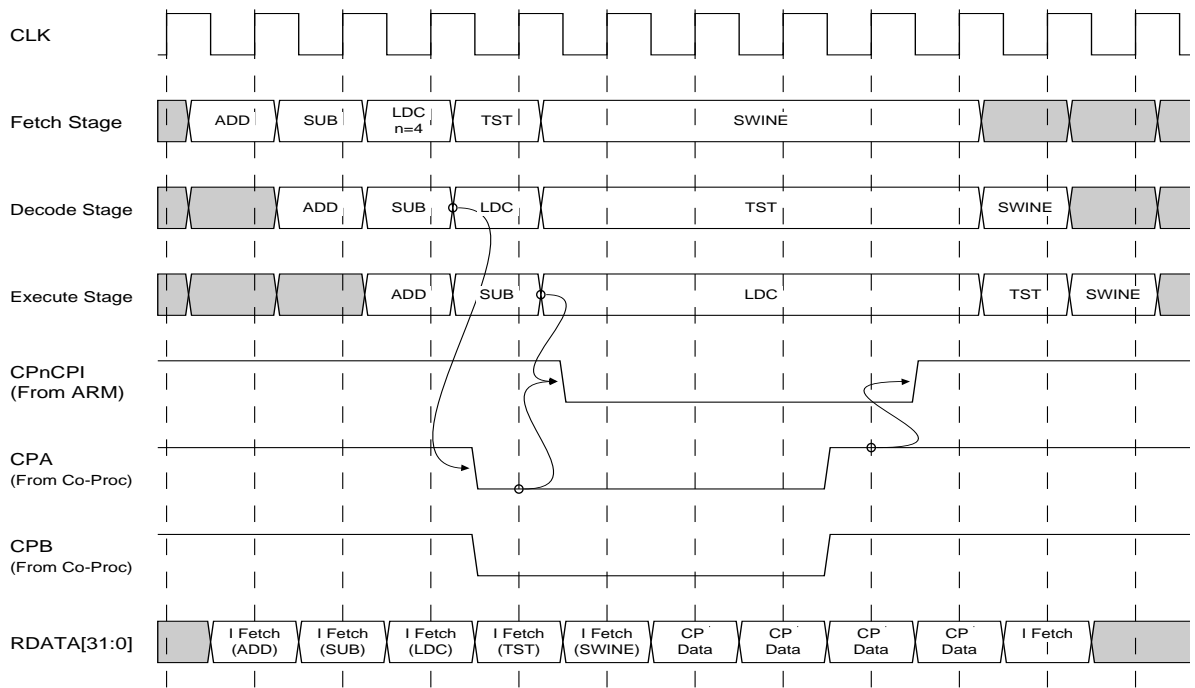


Figure 4-4 Coprocessor load sequence

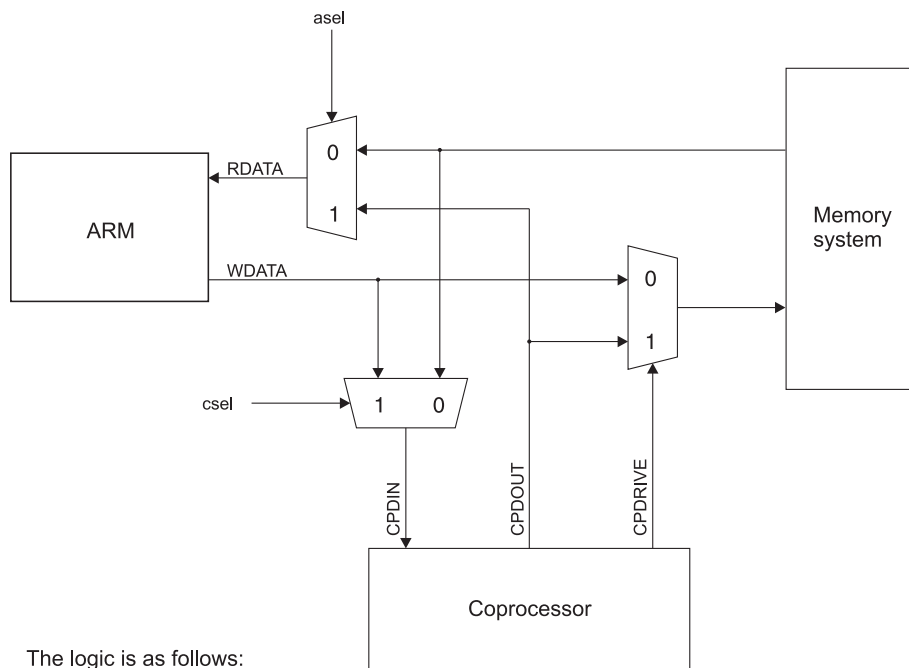
4.5 Connecting coprocessors

A coprocessor in an ARM7TDMI-S system needs to have 32-bit connections to:

- 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 system is shown in Figure 4-5.



The logic is as follows:

```
on RISING CLK
  asel = ((TRANS [1:0] == 01) and (not WRITE))
  csel = ((TRANS [1:0] == 01) and (WRITE))
```

Figure 4-5 Coprocessor connections

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
CPnCPI	Connect this signal to all coprocessors present in the system.
CPA and CPB	The individual CPA and CPB outputs from each coprocessor must be ANDed together, and connected to the CPA and CPB inputs on the ARM7TDMI-S.

You must also multiplex the output data from the coprocessors.

4.6 If you are not using an external coprocessor

If you are implementing a system which 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 will take the undefined instruction trap so that they can be emulated in software if required.

The coprocessor-specific outputs from the ARM7TDMI-S should be left unconnected:

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

4.7 Undefined instructions

The ARM7TDMI-S 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 to take the undefined instruction trap. Any coprocessor instructions that are not accepted by a coprocessor also result in the ARM7TDMI-S taking the undefined instruction trap.

4.8 Privileged instructions

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

Table 4-4 PROT[1] 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 responds with **CPA** and **CPB** HIGH. This causes the ARM7TDMI-S to take the undefined instruction trap.

Chapter 5

Debug Interface

This chapter describes the ARM7TDMI-S debug interface:

- *Overview of the debug interface* on page 5-2
- *Debug systems* on page 5-4
- *Debug interface signals* on page 5-6
- *ARM7TDMI-S core clock domains* on page 5-10
- *Determining the core and system state* on page 5-11.

This chapter also describes the ARM7TDMI-S EmbeddedICE macrocell module:

- *Overview of EmbeddedICE* on page 5-12
- *Disabling EmbeddedICE* on page 5-14
- *The debug communications channel* on page 5-15.

5.1 Overview of the debug interface

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

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

The debug extensions allow the core to be forced into *debug state*. In debug state, the core is stopped, and isolated from the rest of the system. This allows the internal state of the core, and the external state of the system, to be examined while all other system activity continues as normal. When debug has been completed, the ARM7TDMI-S restores the core and system state, and resumes program execution.

5.1.1 Stages of debug

A request on one of the external debug interface signals, or on an internal functional unit known as the *EmbeddedICE macrocell*, forces the ARM7TDMI-S into debug state. The interrupts which activate debug are:

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

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

5.1.2 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. To synchronize 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** (**R**eturned **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 5-1 shows this synchronization:

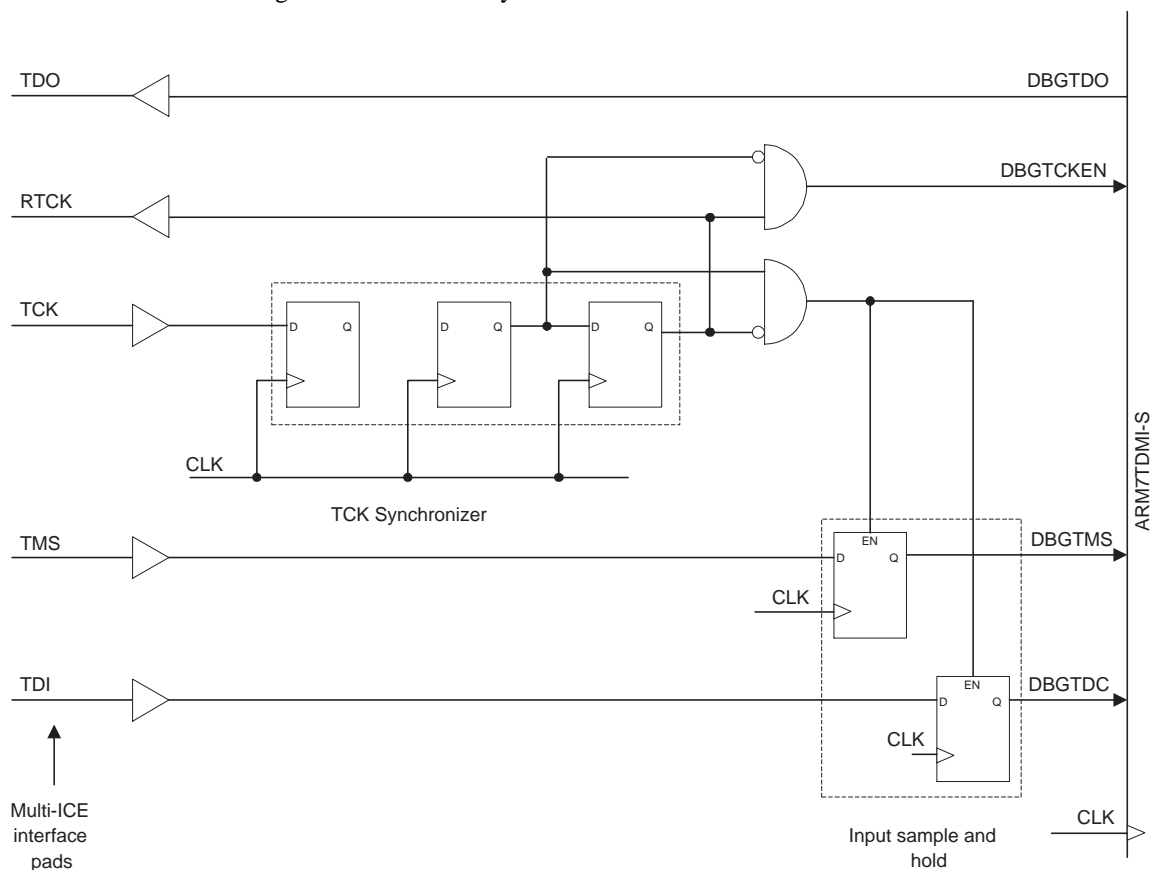


Figure 5-1 Clock synchronization

5.2 Debug systems

The ARM7TDMI-S forms one component of a debug system that interfaces from the high-level debugging performed by the user to the low-level interface supported by the ARM7TDMI-S. Figure 5-2 shows a typical debug system.

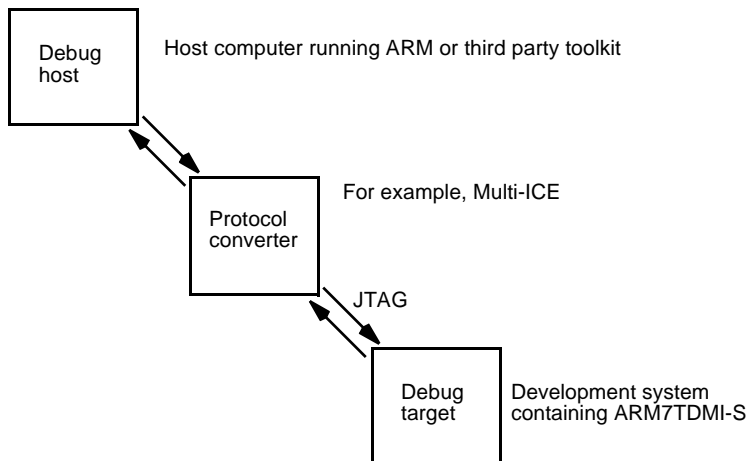


Figure 5-2 Typical debug system

A debug system typically has three parts:

- *The debug host*
- *The protocol converter* on page 5-4
- *The ARM7TDMI-S* on page 5-5.

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

5.2.1 The debug host

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

5.2.2 The protocol converter

An interface, such as an RS232, connects the debug host to the ARM7TDMI-S development system. The messages broadcast over this connection must be converted to the interface signals of the ARM7TDMI-S. The protocol converter performs the conversion.

5.2.3 The ARM7TDMI-S

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

- allow the user to stall the core from program execution
- examine the core internal state
- examine the state of the memory system
- resume program execution.

The major blocks of the ARM7TDMI-S are:

- The ARM7TDMI-S. This is the CPU core, with hardware support for debug.
- The EmbeddedICE macrocell. This is a set of registers and comparators used to generate debug exceptions (such as breakpoints). This unit is described in *Overview of EmbeddedICE* on page 5-12.
- The TAP controller. This controls the action of the scan chains via a JTAG serial interface.

These blocks are shown in Figure 5-3:

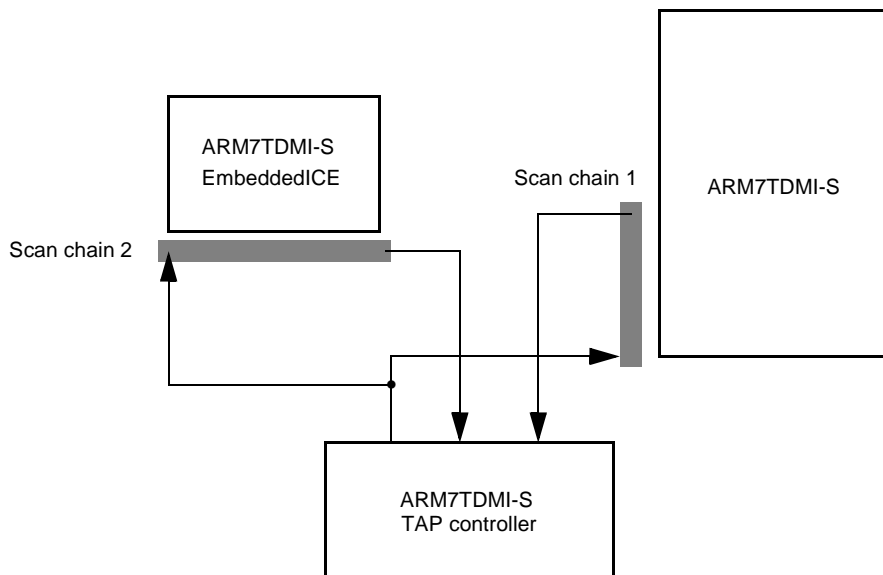


Figure 5-3 ARM7TDMI-S block diagram

The rest of this chapter describes the ARM7TDMI-S hardware debug extensions.

5.3 Debug interface signals

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

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

5.3.1 Entry into debug state

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

You can use EmbeddedICE to program the conditions under which a breakpoint or watchpoint may occur. Alternatively, you can use external logic to monitor the address and data bus, and flag breakpoints and watchpoints via the **DBGBREAK** pin.

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 may be a delay before the ARM7TDMI-S enters debug state. When it enters debug state, the **DBGACK** signal is asserted. The timing for an externally-generated breakpoint is shown in Figure 5-4 on page 5-7.

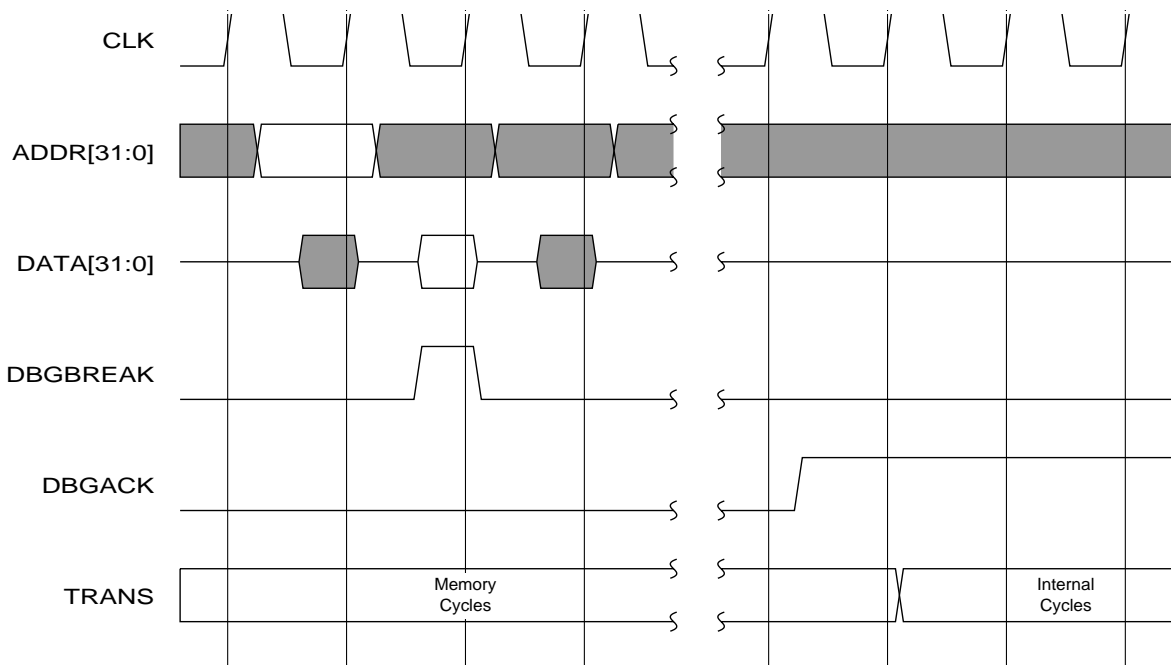


Figure 5-4 Debug state entry

Entry into debug state on breakpoint

The ARM7TDMI-S 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 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. The ARM7TDMI-S enters debug state regardless of whether the condition was met.

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

Entry into debug state on watchpoint

Watchpoints occur on data accesses. A watchpoint is always taken, but the core may not enter debug state immediately. In all cases, the current instruction completes. If the current instruction is a multiword load or store (an LDM or STM), many cycles may elapse before the watchpoint is taken.

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

————— Note —————

Watchpoints are similar to data aborts, the difference being that when a data abort occurs, although the instruction completes, the ARM7TDMI-S prevents all subsequent changes to the ARM7TDMI-S state. This action allows the abort handler to cure the cause of the abort, and the instruction to be re-executed.

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

Entry into debug state on debug request

The ARM7TDMI-S may be forced into debug state on debug request in either of the following ways:

- through EmbeddedICE programming (see *Programming breakpoints on page D-32 and Programming watchpoints on page D-34*)
- by asserting the **DBGREQ** pin.

When the **DBGRO** 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 enters debug state immediately (this is similar to the action of **nIRQ** and **nFIQ**).

Action of the ARM7TDMI-S in debug state

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

Caution

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

The system should not change the **CFGBIGEND** signal during debug. If **CFGBIGEND** changes, the programmer's view of the ARM7TDMI-S changes with the debugger unaware that the core has reset. Make sure, also, that **nRESET** is held stable during debug. When the system applies reset to the ARM7TDMI-S (that is, **nRESET** is driven LOW), the ARM7TDMI-S state changes with the debugger unaware that the core has reset.

5.4 ARM7TDMI-S core clock domains

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

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

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

5.5 Determining the core and system state

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

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

For more details about determining the core state, see *Determining the core and system state* on page D-15.

5.6 Overview of EmbeddedICE

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

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

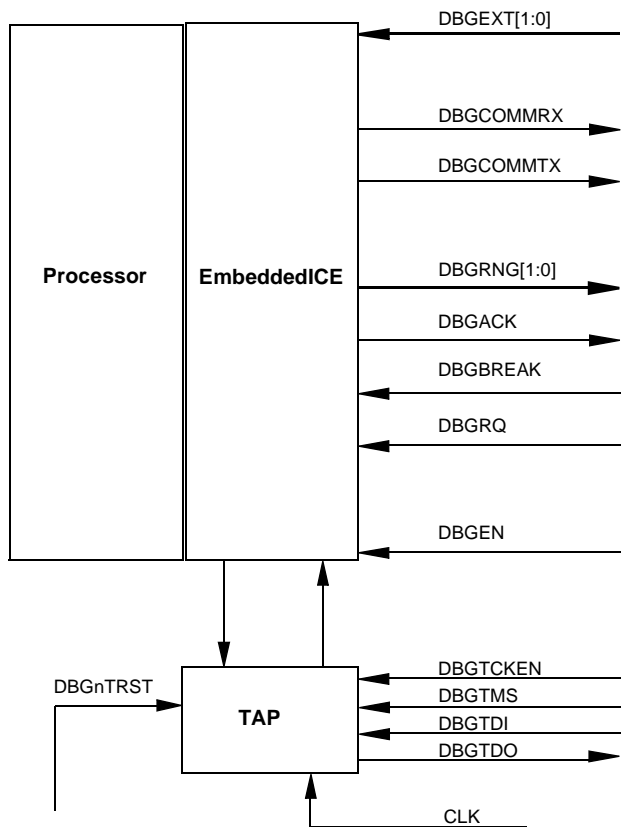


Figure 5-5 The ARM7TDMI-S, TAP controller and EmbeddedICE

The EmbeddedICE macrocell comprises:

- two real-time watchpoint units
- two independent registers, the debug control register and the debug status register.

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

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

———— **Note** —————

You can mask any bit so that its value does not affect the comparison.

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

5.7 Disabling EmbeddedICE

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

———— **Caution** —————

Hard wiring the **DBGEN** input LOW *permanently* disables debug access.

When **DBGEN** is LOW, it inhibits **DBGBREAK** and **DBGREQ** to the core, and **DBGACK** from the ARM7TDMI-S will always be LOW.

5.8 The debug communications channel

The ARM7TDMI-S EmbeddedICE unit contains a communications channel for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The communications channel comprises:

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

These registers are located in fixed locations in the EmbeddedICE unit register map (as shown in Figure D-5 on page D-28) and are accessed from the processor via MCR and MRC instructions to coprocessor 14.

5.8.1 Debug comms channel registers

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

31	30	29	28	27:2	1	0
0	0	1	0		W	R

Figure 5-6 Debug comms control register

The function of each register bit is described below:

- Bits 31:28 contain a fixed pattern that denotes the EmbeddedICE version number (in this case 0001).
- Bits 27:2 are reserved.
- Bit 1 denotes whether the comms data write register is available (from the viewpoint of the processor).
 If, from the point of view of the processor, the comms data write register is free (W=0), new data may be written.
 If the register is not free (W=1), the processor must poll until W=0.
 From the point of view of the debugger, when W=1, some new data has been written that may then be scanned out.

Bit 0 denotes whether there is new data in the comms data read register. If, from the point of view of the processor, $R=1$, there is some new data which may be read using an MRC instruction. From the point of view of the debugger, if $R=0$, the comms data read register is free, and new data may be placed there through the scan chain. If $R=1$, this denotes that data previously placed there through the scan chain has not been collected by the processor, and so the debugger must wait.

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

You should use the following instructions:

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

This returns the debug comms control register into Rd.

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

This writes the value in Rn to the comms data write register.

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

This returns the debug data read register into Rd.

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

5.8.2 Communications via the comms channel

Messages can be sent and received via the comms channel.

Sending a message to the debugger

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

The processor reads the debug comms control register to check status of the W bit.

- If W bit is clear, the comms 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 comms data write register, the W bit is set in the debug comms control register.

The debugger sees both the R and W bits when it polls the debug comms 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 may 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 via 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 via 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 may now be repeated.

Chapter 6

Instruction Cycle Timings

This chapter gives the ARM7TDMI-S instruction cycle timings:

- *Introduction to instruction cycle timings* on page 6-3
- *Instruction cycle count summary* on page 6-5
- *Branch and ARM branch with link* on page 6-7
- *Thumb branch with link* on page 6-8
- *Branch and exchange* on page 6-9
- *Data operations* on page 6-10
- *Multiply and multiply accumulate* on page 6-12
- *Load register* on page 6-14
- *Store register* on page 6-16
- *Load multiple registers* on page 6-17
- *Store multiple registers* on page 6-19
- *Data swap* on page 6-20
- *Software interrupt and exception entry* on page 6-21
- *Coprocessor data processing operation* on page 6-22
- *Load coprocessor register (from memory to coprocessor)* on page 6-23
- *Store coprocessor register (from coprocessor to memory)* on page 6-25

- *Coprocessor register transfer (move from coprocessor to ARM register) on page 6-27*
- *Coprocessor register transfer (move from ARM register to coprocessor) on page 6-28*
- *Undefined instructions and coprocessor absent on page 6-29*
- *Unexecuted instructions on page 6-30.*

6.1 Introduction to 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 as such in the following tables.

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

- Address is **ADDR[31:0]** registered to the cycle to which they apply
- Lock is **LOCK** registered to the cycle to which it applies
- Size is **SIZE[1:0]** registered to the cycle to which they apply
- Write is **WRITE** registered to the cycle to which it applies
- Prot1 and Prot0 are **PROT[1:0]** registered to the cycle to which they apply
- Tbit is **CPTBIT** registered to the cycle to which it applies.

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. They are therefore shown in the tables the cycle after they have been driven by the coprocessor.

Transaction types are shown in Table 6-1.

Table 6-1 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, the cycle counts must be adjusted accordingly.

6.2 Instruction cycle count summary

In the pipelined architecture of the ARM7TDMI-S, while one instruction is being fetched, the previous instruction is being decoded, and the one prior to that is being executed. Table 6-2 shows the number of cycles required by an instruction, once that instruction reaches the execute stage.

The number of cycles for a routine can be calculated from the figures in Table 6-2. These figures assume execution of the instruction, unexecuted instructions take one cycle.

In the table:

- 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 may be zero or more).

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

Table 6-2 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

Table 6-2 Instruction cycle counts (continued)

Instruction	Qualifier	Cycle count
LDR	R15 destination	+N +I +N +2S
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 6-1 on page 6-4.

6.3 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 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 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 6-3 shows the cycle timings, where:

- pc is the address of the branch instruction
 pc' is an address calculated by the ARM7TDMI-S
 (pc') are the contents of that address.

Table 6-3 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.

6.4 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 similarly to the ARM BL instruction over three cycles:
 - During the first cycle, the ARM7TDMI-S calculates the final branch destination while performing a prefetch from the current PC.
 - During the second cycle, the ARM7TDMI-S performs a fetch from the branch destination. The return address is stored in r14.
 - During the third cycle, the ARM7TDMI-S 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 6-4 shows the cycle timings of the complete operation.

Table 6-4 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*.

6.5 Branch and exchange

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

1. During the first cycle, the ARM7TDMI-S 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 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 performs a fetch from the destination +2 or +4 dependent on the new specified state, refilling the instruction pipeline.

Table 6-5 shows the cycle timings.

Table 6-5 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 Tbit 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.

6.6 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 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 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). As the address remains stable through both cycles, the memory manager can merge this internal cycle with the following sequential access.

The PC may be one or more of the register operands. When the PC is the destination, external bus activity may be affected. When the ARM7TDMI-S writes the result to the PC, the contents of the instruction pipeline are invalidated, and the ARM7TDMI-S takes the address for the next instruction prefetch from the ALU rather than the address incrementer. The ARM7TDMI-S refills the instruction pipeline before any further 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 6-6.

Table 6-6 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					
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.

6.7 Multiply and multiply accumulate

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

The cycle timings are shown in Table 6-7 to Table 6-10, in which m is the number of cycles required by the multiplication algorithm (see *Instruction cycle count summary* on page 6-5).

Table 6-7 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 6-8 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 6-9 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 6-10 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.

6.8 Load register

A load register instruction takes a variable number of cycles:

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

The load register cycle timings are shown in Table 6-11, where:

b, h and w are byte, halfword, and word as defined in Table D-5 on page D-30.

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 6-11 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 pc+3i	w/h	0	-	S cycle	1	s
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 pc'+8	w	0	(pc'+4)	S cycle	0	s

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

———— **Note** —————

Destination equals PC is not possible in Thumb state.

6.9 Store register

A store register has two cycles:

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

The store register cycle timings are shown in Table 6-12, 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 6-12 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						

6.10 Load multiple registers

A *load multiple* (LDM) takes four cycles:

1. During the first cycle, the ARM7TDMI-S calculates the address of the first word to be transferred, while performing a prefetch from memory.
2. During the second cycle, the ARM7TDMI-S fetches the first word and performs the base modification.
3. During the third cycle, the ARM7TDMI-S 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 needed 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 moves the last word to its destination register. The last cycle may 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 prevents all register writing after the abort. The ARM7TDMI-S changes the final cycle to restore the modified base register (which the load activity before the abort occurred may have overwritten).

When the PC is in the list of registers to be loaded, the ARM7TDMI-S 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, $\text{POP}\{\text{Rlist}, \text{PC}\}$ equates to an LDM with destination = PC.

The LDM cycle timings are shown in Table 6-13.

Table 6-13 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 6-13 Load multiple registers instruction cycle operations (continued)

Cycle	Address	Size	Write	Data	TRANS[1:0]	Prot0	
1 register dest=pc	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
	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 (n>1)	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
	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 (n>1) incl pc	1	pc+2i	w/h	0	(pc+2i)	N cycle	0
	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	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						

6.11 Store multiple registers

Store multiple (STM) proceeds very much as load multiple, although without the final cycle. There are therefore two cycles:

1. During the first cycle, the ARM7TDMI-S calculates the address of the first word to be stored.
2. During the second cycle, the ARM7TDMI-S 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 6-14.

Table 6-14 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 pc+3i	w	1	R	N cycle	1
n registers (n>1)	1	pc+8	w/h	0	(pc+2i)	N cycle	0
	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++ pc+12	w	1	R'''	N cycle	1

6.12 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 may be a byte or word quantity (b/w).

The ARM7TDMI-S may 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 6-15, where b and w are byte and word as defined in Table D-5 on page D-30.

Table 6-15 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 is driven HIGH for both load and store data cycles to indicate to the memory controller that this is an atomic operation.

6.13 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 constructs the forced address, and a mode change may take place. The ARM7TDMI-S moves the return address to r14 and moves the CPSR to SPSR_svc.
2. During the second cycle, the ARM7TDMI-S 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 6-16, 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 6-16 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								

6.14 Coprocessor data processing operation

A *coprocessor data processing* (CDP) operation is a request from the ARM7TDMI-S 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 busy-waits until **CPB** goes LOW. However, an interrupt may cause the ARM7TDMI-S 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 6-17.

Table 6-17 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.

6.15 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 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 may cause the ARM7TDMI-S 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 6-18.

Table 6-18 Load coprocessor register instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
1 register	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
ready	2	da	w	0	(da)	N cycle	1	1	1	1
		pc+12								
1 register	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
not ready	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	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
(m>1)	2	da	w	0	(da)	S cycle	1	1	0	0
ready	•	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 6-18 Load coprocessor register instruction cycle operations (continued)

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

6.16 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 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 may cause the ARM7TDMI-S 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 6-19.

Table 6-19 Store coprocessor register instruction cycle operations

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnI	CPA	CPB
1 register	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
ready	2	da	w	1	CPdata	N cycle	1	1	1	1
		pc+12								
1 register	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
not ready	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	1	pc+8	w	0	(pc+8)	N cycle	0	0	0	0
(m>1)	2	da	w	1	CPdata	S cycle	1	1	0	0
ready	•	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 6-19 Store coprocessor register instruction cycle operations (continued)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0	CPnl	CPA	CPB
m registers	1	pc+8	w	0	(pc+8)	I cycle	0	0	0	1
(m>1)	2	pc+8	w	0	-	I cycle	1	0	0	1
not ready	•	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.

6.17 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 may cause the ARM7TDMI-S 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 may merge the third cycle with the following prefetch cycle into a merged I-S cycle.

The MRC cycle timings are shown in Table 6-20.

Table 6-20 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.

—————

6.18 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 may cause the ARM7TDMI-S to abandon the coprocessor instruction (see *Consequences of busy-waiting* on page 4-8).

The MCR cycle timings are shown in Table 6-21.

Table 6-21 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.

6.19 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 allows 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 6-22.

Table 6-22 Undefined instruction cycle operations

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

6.20 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 6-23.

Table 6-23 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 7

AC Parameters

This chapter gives the AC timing parameters of the ARM7TDMI-S:

- *Timing diagrams* on page 7-2
- *AC timing parameter definitions* on page 7-7.

7.1 Timing diagrams

The timing diagrams in this section are:

- *Figure 7-1 Timing parameters*
- *Figure 7-2 Coprocessor timing*
- *Figure 7-3 Exception and configuration input timing*
- *Figure 7-4 Debug timing*
- *Figure 7-5 Scan general timing.*

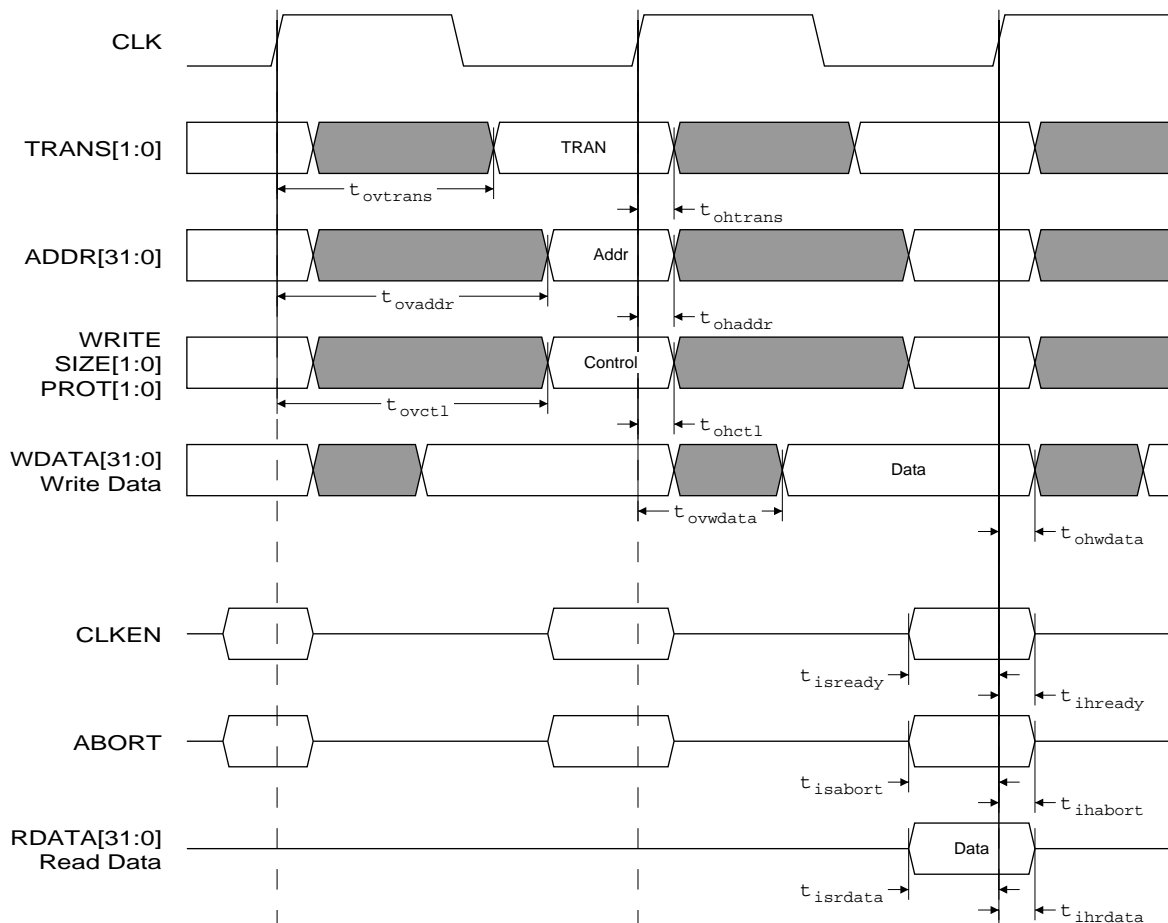


Figure 7-1 Timing parameters

———— **Note** ————

The timing for both read and write data access are superimposed in the figure. The **WRITE** signal conveys whether the access uses the read 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.

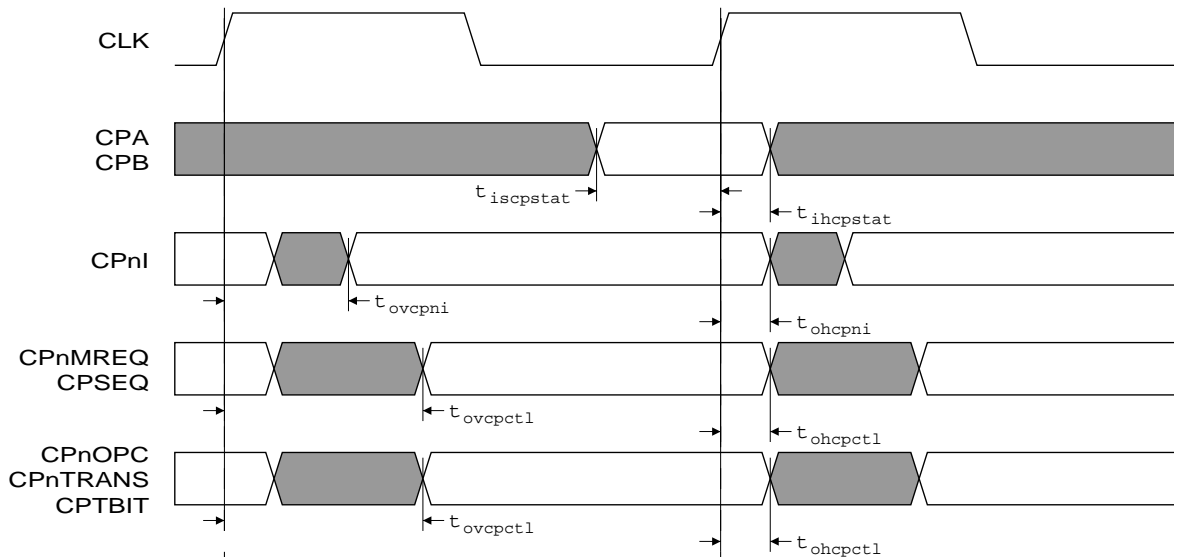


Figure 7-2 Coprocessor timing

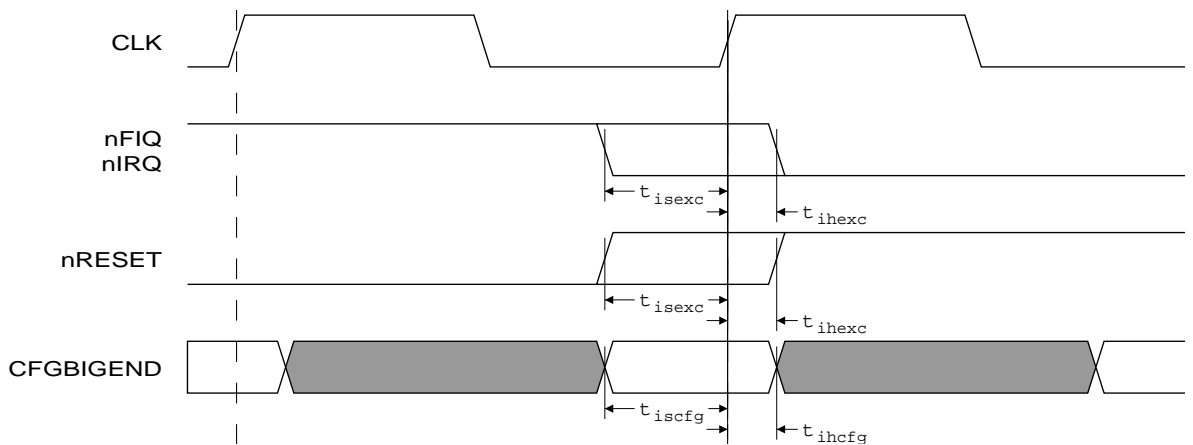


Figure 7-3 Exception and configuration input timing

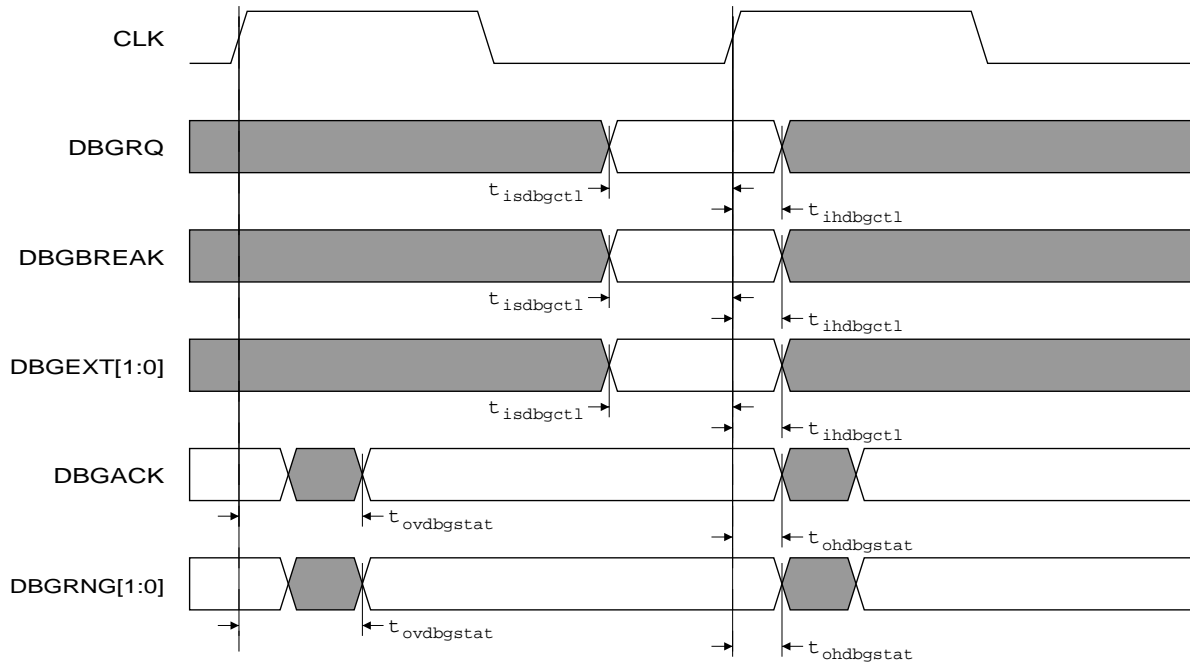


Figure 7-4 Debug timing

Note

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

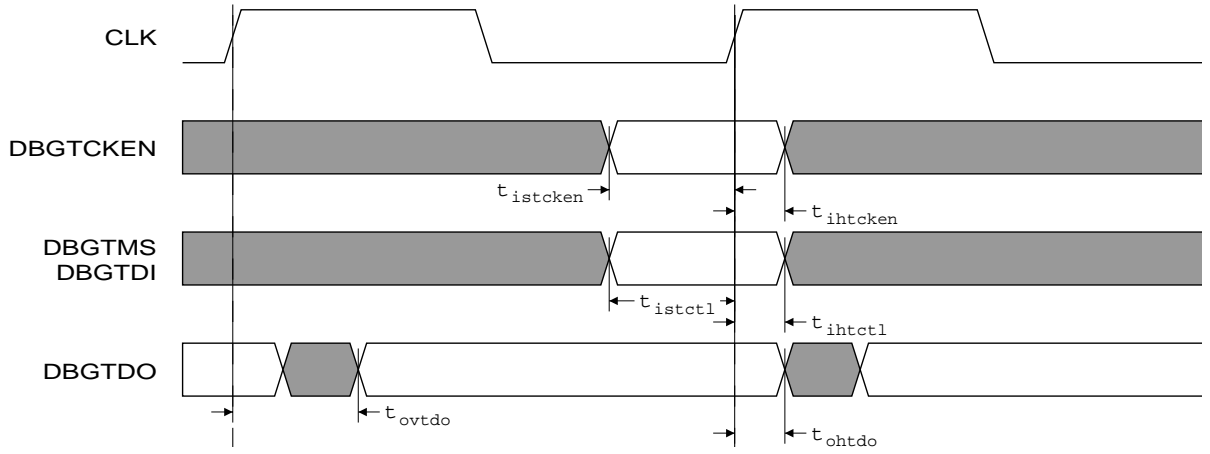


Figure 7-5 Scan general timing

7.2 AC timing parameter definitions

Table 7-1 shows target AC parameters. All figures are expressed as percentages of the **CLK** period at maximum operating frequency.

———— **Note** —————

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

Table 7-1 Provisional AC parameters

Symbol	Parameter	Min	Max
t_{cyc}	CLK cycle time.	100%	
$t_{isready}$	CLKEN input setup to rising CLK .	40%	
$t_{ihready}$	CLKEN input hold from rising CLK .		0%
$t_{isabort}$	ABORT input setup to rising CLK .	15%	
$t_{ishbort}$	ABORT input hold from rising CLK .		0%
$t_{isrdata}$	RDATA input setup to rising CLK .	10%	
t_{ihdata}	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_{ovtran}	Rising CLK to transaction type valid.		50%
t_{ohtran}	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 7-1 Provisional AC parameters (continued)

Symbol	Parameter	Min	Max
$t_{ovcpecl}$	Rising CLK to coprocessor control valid.		80%
$t_{ohcpecl}$	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%
$t_{isdbgstat}$	Debug status inputs setup to rising CLK .	10%	
$t_{ihdbgstat}$	Debug status inputs hold from rising CLK .		0%
$t_{ovdbctrl}$	Rising CLK to debug control valid.		40%
$t_{ohdbctrl}$	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%	

Appendix A

Signal Descriptions

This appendix lists and describes all the ARM7TDMI-S signals.

A.1 Signal descriptions

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

Table A-1 Signal descriptions

Name	Type	Description
ABORT	Input	Memory abort or bus error. This is an input which 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 may be used with a free-running CLK to add synchronous wait-states. Alternatively, the clock may 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 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 which is capable of performing the operation that the ARM7TDMI-S is requesting (by asserting nCPI), takes CPA LOW, set up to the cycle edge that precedes the coprocessor access. When CPA is signalled HIGH, and the coprocessor cycle is executed (as signalled by CPnI signalled 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, indicates this 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.
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.

Table A-1 Signal descriptions (continued)

Name	Type	Description
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 the ARM7TDMI-S is in debug state. It is enabled only when DBGEN is HIGH.
DBGBREAK	Input	EmbeddedICE breakpoint/watchpoint indicator. This signal allows 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 allows extension of the internal breakpoints provided by the EmbeddedICE module. DBGBREAK is enabled only when DBGEN is HIGH. (This signal is analogous to BREAKPT on the hard macrocell.)
DBGCOMMRX	Output	EmbeddedICE 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.)

Table A-1 Signal descriptions (continued)

Name	Type	Description
DBGCOMMTX	Output	EmbeddedICE 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 external input 0, external input 1. These are inputs to the EmbeddedICE macrocell logic in the ARM7TDMI-S which 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.)
DBGRNG[1:0]	Output	EmbeddedICE rangeout. This signal indicates that EmbeddedICE watchpoint register 0/1 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 data in. JTAG test data input. DBGTDI is enabled only when DBGEN is HIGH.
DBGTDO	Output	EmbeddedICE data out. Output from the boundary scan logic. DBGTDO is enabled only when DBGEN is HIGH.
DBGnTDOEN	Output	Not DBGTDO enable. When LOW, this signal denotes that serial data is being driven out on the DBGTDO output. DBGnTDOEN would normally be used as an output enable for a DBGTDO pin in a packaged part.
DBGTMS	Input	EmbeddedICE mode select. JTAG test mode select. DBGTMS is enabled only when DBGEN is HIGH.
DBGnTRST	Input	Not test reset. This is the internally synchronized active-low reset signal for the EmbeddedICE macrocell internal state.

Table A-1 Signal descriptions (continued)

Name	Type	Description
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 lower-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 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.)
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 non-wait 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).

Table A-1 Signal descriptions (continued)

Name	Type	Description
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 non-sequential 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.)
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:

- *Interface signals* on page B-2
- *ATPG scan interface* on page B-7
- *Timing parameters* on page B-8
- *ARM7TDMI-S design considerations* on page B-9.

B.1 Interface signals

The signal names have prefixes which 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 debug support input or output.

Other signals provide the system designer's interface which is primarily memory-mapped. Table B-1 provides the ARM7TDMI-S signals with their ARM7TDMI hard macrocell equivalent signals. The notes to this table are given in *Notes to Table B-1* on page B-5.

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

ARM7TDMI-S signal	Function	ARM7TDMI hard macrocell equivalent	Note
ABORT	1 = memory abort or bus error. 0 = no error.	ABORT	
ADDR[31:0]	32-bit address output bus, available in the cycle preceding the memory cycle.	A[31:0]	1
CFGBIGEND	1 = big-endian configuration. 0 = little-endian configuration.	BIGEND	
CLK	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	2
CLKEN	System memory interface clock enable: 1 = advance the core on rising CLK . 0 = prevent the core advancing on rising CLK .	NWAIT	3
CPA	Coprocessor absent. Tie HIGH when no coprocessor is present.	CPA	4
CPB	Coprocessor busy. Tie HIGH when no coprocessor is present.	CPB	4
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 signals and ARM7TDMI hard macrocell equivalents (continued)

ARM7TDMI-S signal	Function	ARM7TDMI hard macrocell equivalent	Note
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 communication channel receive buffer full output.	COMMRX	
DBGCOMMTX	EmbeddedICE communication channel transmit buffer empty output.	COMMTX	
DBGEN	Debug enable. Tie this signal HIGH in order to be able to use the debug features of the ARM7TDMI.	DBGEN	
DBGEXT[1:0]	EmbeddedICE EXTERN debug qualifiers (tie LOW when not required).	EXTERN0, EXTERN1	
DBGnEXEC	Active LOW condition codes success at execute stage, pipelined in the preceding access.	nEXEC	
DBGnTDOEN	Active LOW TAP controller DBGTDO output qualifier.	nTDOEN	6
DBGnTRST	Active LOW TAP controller reset (asynchronous assertion). Resets the ICEBreaker subsystem.	nTRST	6

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

ARM7TDMI-S signal	Function	ARM7TDMI hard macrocell equivalent	Note
DBG RNG[1:0]	EmbeddedICE rangeout qualifier outputs.	RANGEOUT1, RANGEOUT0	
DBG RQ	External debug request (tie LOW when not required).	DBG RQ	5
DBG TCKEN	Multi-ICE clock input qualifier sampled on the rising edge of CLK . Used to qualify CLK to enable the debug subsystem.		
DBG TDI	Multi-ICE TDI test data input.	TDI	6
DBG TDO	EmbeddedICE TAP controller serial data output.	TDO	6
DBG TMS	Multi-ICE TMS test mode select input.	TMS	6
LOCK	Indicates whether the current address is part of locked access. This signal is generated by execution of a SWP instruction.	LOCK	1
nFIQ	Active LOW fast interrupt request input.	nFIQ	7
nIRQ	Active LOW interrupt request input.	nIRQ	7
nRESET	Active LOW reset input (asynchronous assertion). Resets the processor core subsystem.	nRESET	
PROT[1:0]	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	1,9
RDATA[31:0]	Unidirectional 32-bit input data bus.	DIN[31:0]	8
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 signals and ARM7TDMI hard macrocell equivalents (continued)

ARM7TDMI-S signal	Function	ARM7TDMI hard macrocell equivalent	Note
TCKEN	JTAG interface clock enable: 1 = advance the JTAG logic on rising CLK . 0 = prevent the JTAG logic advancing on rising CLK .		
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	
WRITE	Write access indicator.	nRW	1

Notes to Table B-1

- 1 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.
- 2 **CLK** is a rising edge clock. It is inverted with respect to the **MCLK** signal used on the ARM7TDMI hard macrocell.
- 3 **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**) may still change in a cycle in which **CLKEN** is taken LOW.

You must take this possibility into account when designing a memory system.
- 4 **CPA** and **CPA** are sampled on the rising edge of **CLK**. They may no longer change in the first phase of the next cycle, as is possible with the ARM7TDMI hard macrocell.
- 5 **DBGRQ** must be synchronized externally to the macrocell. It is *not* an asynchronous input as on the ARM7TDMI hard macrocell.

- 6 All JTAG signals are synchronous to **CLK** on the ARM7TDMI-S. There is no asynchronous **TCLK** as on the ARM7TDMI hard macrocell.

An external synchronizing circuit can be used to generate **TCLKEN** when an asynchronous **TCLK** is required.
- 7 **nFIQ** and **nIRQ** are synchronous inputs to the ARM7TDMI-S, and are sampled on the rising edge of **CLK**.

Asynchronous interrupts are not supported.
- 8 The ARM7TDMI-S 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.
- 9 **PROT[0]** is the equivalent of **nOPC**, and **PROT[1]** is the equivalent of **nTRANS** on the ARM7TDMI hard macrocell.

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 7-2.

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 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_{ihdata} , 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_{ohtran} , valid time is t_{ovtran} when **CLKEN** is active.
- **LOCK**, **PROT**, **SIZE**, **WRITE** control output hold time is t_{ohctrl} , valid time is t_{ovctrl} 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, a number of areas require special consideration. These are the:

- *Master clock*
- *JTAG interface timing*
- *Interrupt timing*
- *Address class signal timing.*

B.4.1 Master clock

The master clock to the ARM7TDMI-S, **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 are synchronous to the master clock input, **CLK**. When an external **TCLK** is used, use an external synchronizer to the ARM7TDMI-S.

B.4.3 Interrupt timing

As with all ARM7TDMI-S 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.

B.4.4 Address class signal timing

The address class outputs (**ADDR[31:0]**, **WRITE**, **SIZE[1:0]**, **PROT[1:0]** and **LOCK**) on the ARM7TDMI-S 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 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]** may have moved on to address for the next memory cycle.

For further details, refer to Chapter 3 *Memory Interface*.

Appendix C

Implications of Removing the Debugger or 64-bit Multiply Support

This appendix explains the implications of removing the debugger (EmbeddedICE) or implementing only 32-bit multiply:

- *Implications of removing EmbeddedICE* on page C-2
- *Using MUL32* on page C-3
- *MUL32 instructions* on page C-3
- *MUL32 performance* on page C-5.

C.1 Implications of removing EmbeddedICE

When the EmbeddedICE module is deselected, certain input/outputs on the ARM7TDMI-S macrocell become unconnected. Designers must make sure that no logic is connected to any of the outputs listed below in a design which does not incorporate the EmbeddedICE macrocell.

Inputs which become unconnected:

- **DBGnTRST**
- **DBGTCKEN**
- **DBGTMS**
- **DBGTDI**
- **DBGBREAK**
- **DBGGEN**
- **DBGGRQ**
- **DBGEXT[1:0]**

Outputs which become undriven:

- **DBGACK**
- **DBGBREAK**
- **DBGRNG[1:0]**
- **DBGTDO**
- **DBGnTDOEN**
- **DBGCOMMRX**
- **DBGCOMMTX**

C.2 Using MUL32

The multiplier is an essential component of ARMTDMI-S architecture. Designers must not remove the multiplier altogether, but may wish to substitute the reduced-function multiplier variant, MUL32, where reduced functionality is acceptable.

MUL32 offers:

- 32 x 32 multiplier with 32-bit result
- MUL and MLA opcode support for the ARM instruction set
- MUL opcode support for Thumb instruction set
- minimal gate area implementation
- 2 bits per cycle, with early termination for both positive and negative multiplicands.

MUL32 does not support any 64-bit result opcodes. These are handled by the undefined instruction trap:

- UMULL, UMLAL, SMULL, and SMLAL all trap
- a software trap handler could provide long signed/unsigned instruction emulation.

C.2.1 MUL32 instructions

The MUL32 multiplier supports a subset of the full ARM architecture v4T multiply instruction set. The supported instructions are given here.

MUL{<cond>}{S} Rd, Rm, Rs (ARM)

32-bit register x 32-bit register multiplication with 32-bit result executes only when the condition codes specified {CC} are met.

$Rd := Rm * Rs$

Restrictions on registers are:

- the destination register, Rd, must not be the same as the multiplier register, Rm
- neither Rd or Rm can be r15.

The CPSR flags are optionally updated when the set condition codes {S} instruction bit is set.

MLA{<cond>}{S} Rd, Rm, Rs, Rn (ARM)

32-bit register x 32-bit register multiplication with 32-bit result accumulate executes only when the condition codes {CC} specified are met:

$Rd := Rn + Rm * Rs$

Restrictions on registers are:

- the destination register, Rd, must not be the same as the multiplier register, Rm.
- neither Rd or Rm may be r15.

The CPSR flags are optionally updated when the set condition codes {S} instruction bit is set.

MUL Rd, Ra (Thumb)

32-bit register x 32-bit register multiplication with 32-bit result, setting condition codes:

$Rd := Ra * Rd$

maps to ARM 32-bit multiply:

MULS Rd, Ra, Rd

where Rd and Ra are general-purpose registers in the range r0-r7.

CPSR Flags

The flags are updated only when the set condition codes {S} instruction bit is set in the instruction.

The Z flag is optionally set only when the 32-bit result is zero.

The N flag is optionally set when bit[31] of the result is set.

Preserve the C and V flags in these instructions.

The *ARM Architecture Reference Manual* specifies:

- C flag as UNPREDICTABLE
- preserved values as highly desirable for test and modeling.

Instruction cycle times

MUL takes a data-dependent number of cycles to complete. These appear as internal cycles to the bus interface (indicating non-memory accesses).

The minimum number of cycles is 2. The maximum is 17.

Detailed cycle counts are described in *Instruction cycle count summary* on page 6-5.

C.2.2 MUL32 performance

Table C-1 shows the performance of MUL32.

Table C-1 MUL32 performance

Multiplier (Rs) operand	Cycle count
bits[31:1] all zero or all one	2
bits[31:3] all zero or all one	3
bits[31:5] all zero or all one	4
bits[31:7] all zero or all one	5
bits[31:9] all zero or all one	6
bits[31:11] all zero or all one	7
bits[31:13] all zero or all one	8
bits[31:15] all zero or all one	9
bits[31:17] all zero or all one	10
bits[31:19] all zero or all one	11
bits[31:21] all zero or all one	12
bits[31:23] all zero or all one	13
bits[31:25] all zero or all one	14
bits[31:27] all zero or all one	15
bits[31:29] all zero or all one	16
otherwise	17

Appendix D

Debug in Depth

This appendix describes in further detail the debug features of the ARM7TDMI-S, and includes additional information about the EmbeddedICE macrocell:

- *Scan chains and JTAG interface* on page D-3
- *Scan limitations* on page D-3
- *Resetting the TAP controller* on page D-5
- *Instruction register* on page D-6
- *Public instructions* on page D-7
- *Test data registers* on page D-10
- *ARM7TDMI-S core clock domains* on page D-14
- *Determining the core and system state* on page D-15
- *Behavior of the program counter during debug* on page D-21
- *Priorities and exceptions* on page D-24
- *Scan interface timing* on page D-25
- *The watchpoint registers* on page D-27
- *Programming breakpoints* on page D-32
- *Programming watchpoints* on page D-34
- *The debug control register* on page D-35
- *The debug status register* on page D-36

- *Coupling breakpoints and watchpoints* on page D-38
- *Disabling EmbeddedICE* on page D-40
- *EmbeddedICE timing* on page D-41.

D.1 Scan chains and JTAG interface

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

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

D.1.1 Scan limitations

The two scan paths are referred to as scan chain 1 and scan chain 2, and are shown in Figure D-1.

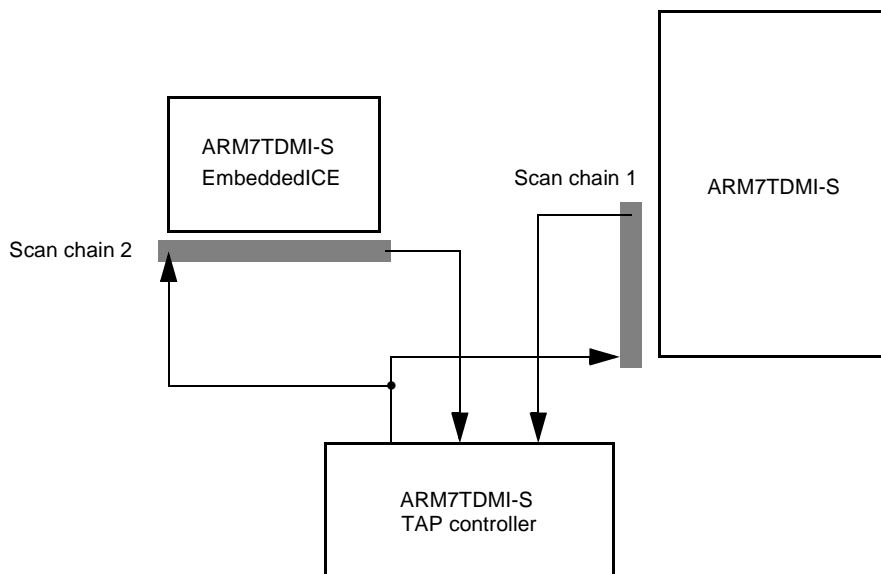


Figure D-1 ARM7TDMI-S scan chain arrangements

Scan chain 1

Scan chain 1 provides serial access to the core data bus **D[31:0]**, 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 allows access to the EmbeddedICE registers. Refer to *Test data registers* on page D-10 for details.

D.1.2 TAP state machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. Figure D-2 shows the state transitions that occur in the TAP controller. The state numbers shown in the diagram are output from the ARM7TDMI-S on the **TAPSM[3:0]** bits.

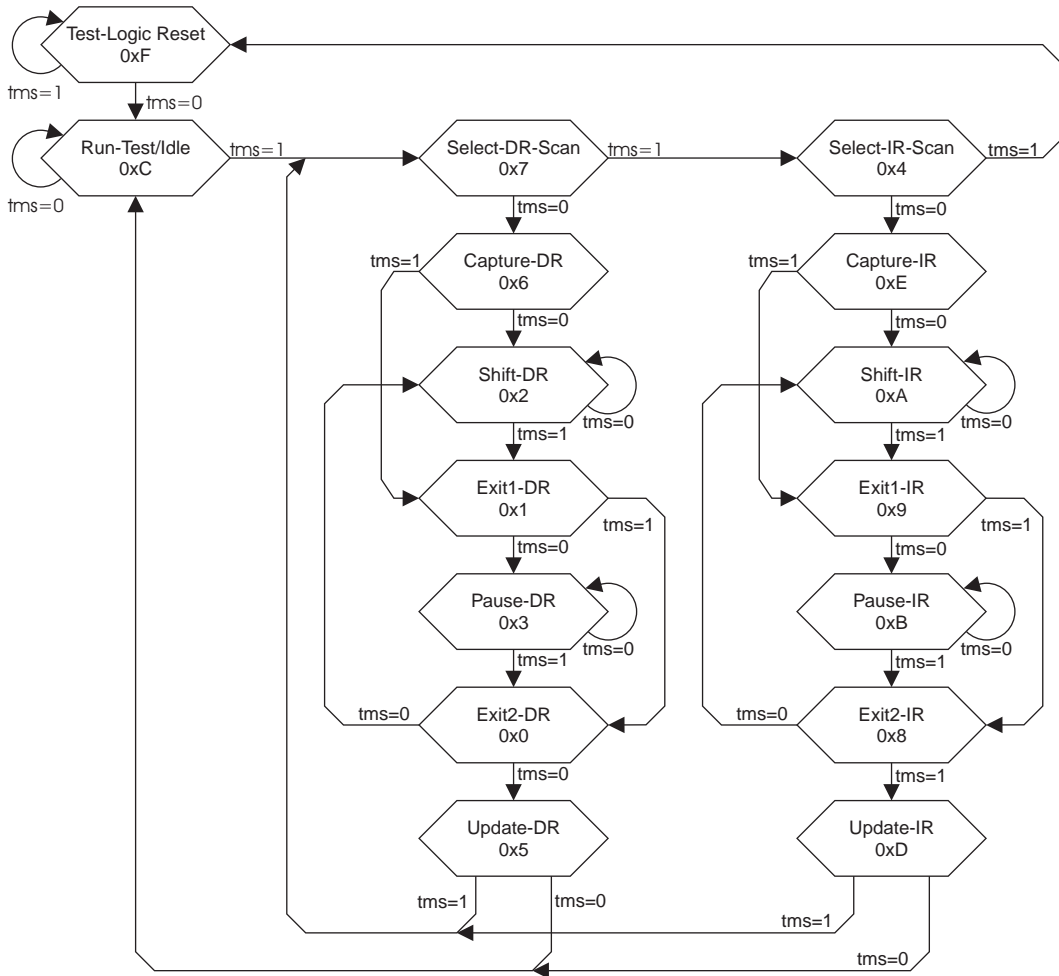


Figure D-2 Test access port controller state transitions

D.2 Resetting the TAP controller

The boundary-scan interface includes a state machine controller, 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, the **DBGnTRST** input may be tied permanently LOW.

———— **Note** —————

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

The action of reset is as follows:

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

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

D.3 Instruction register

The instruction register is 4 bits in length.

There is no parity bit.

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

D.4 Public instructions

Table D-1 gives the public instructions.

Table D-1 Public instructions

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

In the following descriptions, the ARM7TDMI-S samples **DBGTDI** and **DBGTMS** on the rising edge of **CLK** with **DBGTCKEN HIGH**.

D.4.1 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.

D.4.2 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 via the **DBGTDO** pin, while new test data is shifted in via the **DBGTDI** pin.

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

D.4.3 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 allows 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 D-10 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 via the **DBGTDO** pin, while data is shifted into the ID register via the **DBGTDI** pin.
- In the UPDATE-DR state, the ID register is unaffected.

D.4.4 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 via **DBGTDI**, and shifted out via **DBGTDO** after a delay of one **TCK** 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.

D.4.5 RESTART (0100)

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

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

D.5 Test data registers

There are five test data registers which may connect between **DBGTDI** and **DBGTDO**:

- bypass register
- id code register
- instruction register
- scan path select register
- scan chain 1
- scan chain 2.

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

D.5.1 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.

D.5.2 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 register is as follows:



The default device identification code is 0x0f1f0f0f.

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.

D.5.3 Instruction register

Purpose	Changes the current TAP instruction.
Length	4 bits.
Operating mode	<p>In the SHIFT-IR state, the instruction register is selected as the serial path between DBGTDI and DBGTDO.</p> <p>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).</p> <p>During the UPDATE-IR state, the value in the instruction register becomes the current instruction.</p> <p>On reset, IDCODE becomes the current instruction.</p>

D.5.4 Scan path select register

Purpose	Changes the current active scan chain.
Length	4 bits.
Operating mode	<p>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.</p> <p>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 further instructions such as INTEST then apply to that scan chain.</p> <p>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.</p>

Table D-2 shows the scan chain number allocation.

Table D-2 Scan chain number allocation

Scan chain number	Function
0	Reserved*
1	Debug
2	EmbeddedICE programming
3	Reserved*
4	Reserved*
8	Reserved*

Note

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

D.5.5 Scan chains 1 and 2

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

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 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 allows a known value to be scanned into the **DBGBREAK** input.
- While debugging, the value placed in the 33rd bit determines whether the ARM7TDMI-S synchronizes back to system speed before executing the instruction. See *System speed access on page D-23* for further details.
- After the ARM7TDMI-S 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 allows access to the EmbeddedICE 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 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). Refer to Figure D-5 on page D-28 for further details.

D.6 ARM7TDMI-S core clock domains

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

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

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

D.7 Determining the core and system state

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

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

D.7.1 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, the simplest course of action, when shifting scan chain 1, is to repeat the instruction. For example, the encoding for BX R0 is 0x4700, so when 0x47004700 shifts into scan chain 1, the debugger does not have to keep track of the half of the bus on which the processor expects to read the data.

The sequences of ARM instructions below can be used to determine the processor's state.

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

```
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 above use of r0 as the base register for the STM is only for illustration, and you can use any register.

After you have determined the values in the current bank of registers, you may 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 may occur.

The debugger must restore the original mode before exiting debug state. For example, if the debugger had been requested to return the state of the user mode registers and FIQ mode registers, and debug state was entered in supervisor mode, the instruction sequence could 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 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, except TEQP
- all load, store, load multiple, and store multiple instructions
- MSR and MRS.

D.7.2 Determining system state

In order 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 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 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]** in order to determine whether a system speed instruction has completed. In order to access memory, the ARM7TDMI-S 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 can have the bus in the next cycle. If the bus is not available, the ARM7TDMI-S may 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 to control debugging, and so the state of **TRANS[1:0]**, and **DBGACK** can be determined by reading the EmbeddedICE status register. Refer to *The debug status register on page D-36* 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 may have the 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*, below.

When the ARM7TDMI-S 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.

D.7.3 Exit from debug state

Leaving debug state involves:

- restoring the ARM7TDMI-S 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 *Behavior of the program counter during debug* on page D-21 for details on calculating the branch.

Bit 33 of scan chain 1 forces the ARM7TDMI-S 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 then resumes normal operation, fetching instructions from memory. This delay, until the state machine is in the RUN-TEST/IDLE state, allows 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.

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

As a result of the debugging process, more memory accesses occur than would be expected normally. **DBGACK** can inhibit any system peripheral that may be sensitive to the number of memory accesses.

For example, a peripheral that simply counts the number of memory cycles should return the same answer after a program has been run both with and without debugging. Figure D-3 shows the behavior of the ARM7TDMI-S on exit from the debug state.

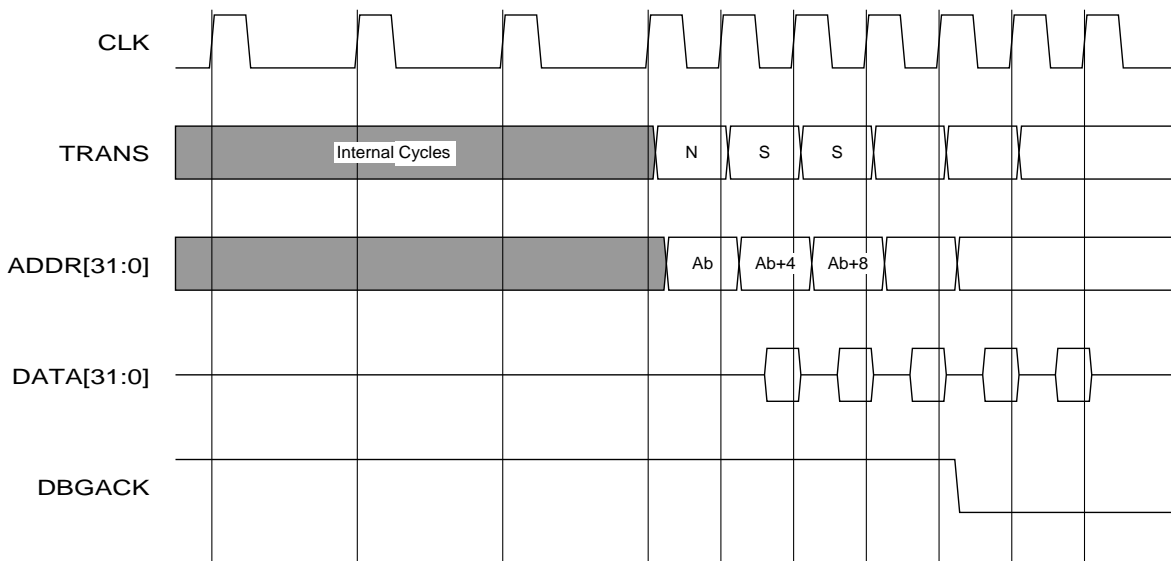


Figure D-3 Debug exit sequence

Figure D-2 on page D-4 shows that the final memory access occurs in the cycle after **DBGACK** goes HIGH. This is the point at which the cycle counter should be disabled. Figure D-3 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 temporarily drops out of debug state, and so **DBGACK** can go LOW. If there are peripherals that are sensitive to the number of memory accesses, they must be led to believe that the

ARM7TDMI-S is still in debug state. You can do this by programming the EmbeddedICE control register to force the value on **DBGACK** to be HIGH. See *The debug status register on page D-36* for more details.

D.8 Behavior of the program counter during debug

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

- a breakpoint
- a watchpoint
- a watchpoint when another exception occurs
- a debug request
- a system speed access.

D.8.1 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 normal way to exit from debug state after a breakpoint is to remove the breakpoint, and branch back to the previously-breakpointed address.

For example, if the ARM7TDMI-S 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 EFFFFFF9; B -7 (2's complement)
```

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

D.8.2 Watchpoints

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

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 should return to the next instruction.

D.8.3 Watchpoint with another exception

If a watchpointed access simultaneously causes a data abort, the ARM7TDMI-S 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 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, the user should be 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 has occurred on a watchpointed access and ten instructions had been executed to determine this eventuality. You could 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 will be refetched and executed. This triggers the watchpoint again, and the ARM7TDMI-S will reenter debug state.

D.8.4 Debug request

Entry into debug state via 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 that the user has invoked a debug request, and decides 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.

D.8.5 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 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 hence 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.

D.8.6 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.

D.9 Priorities and exceptions

When a breakpoint or a debug request occurs, the normal flow of the program is interrupted. Debug therefore can be treated as another type of exception. The interaction of the debugger with other exceptions is described in *Behavior of the program counter during debug* on page D-21. This section covers the priorities.

D.9.1 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 may be data-dependent), the ARM7TDMI-S enters debug state.

The prefetch abort, therefore, takes higher priority than the breakpoint.

D.9.2 Interrupts

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

If an interrupt is pending during the instruction prior to entering debug state, the ARM7TDMI-S enters debug state in the mode of the interrupt. On entry to debug state, the debugger cannot assume that the ARM7TDMI-S is in the mode expected by the user's program. The ARM7TDMI-S 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 does remember that an interrupt has occurred.

D.9.3 Data aborts

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

D.10 Scan interface timing

Figure D-4 provides general scan timing information.

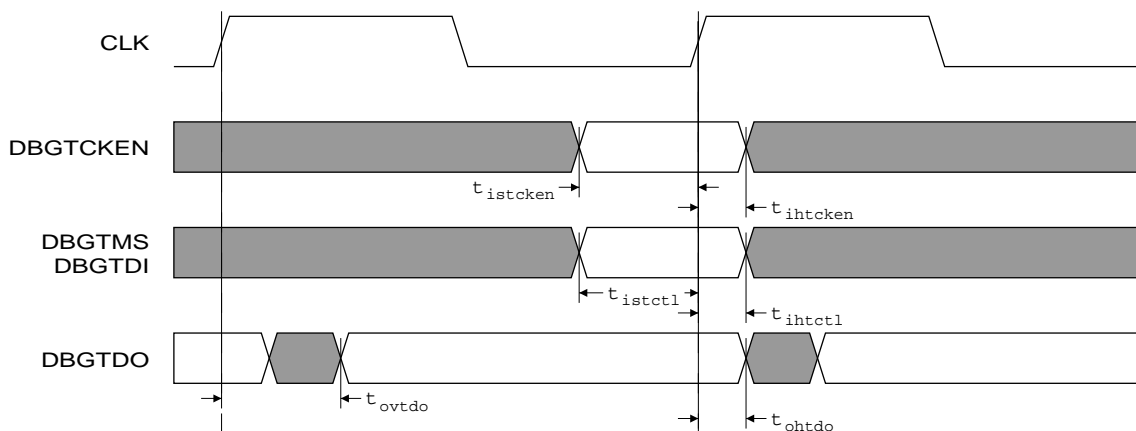


Figure D-4 General scan timing

D.10.1 Scan chain 1 cells

The ARM7TDMI-S provides data for scan chain 1 cells as shown in Table D-3.

Table D-3 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
8	DATA[7]	Input/output
9	DATA[8]	Input/output
10	DATA[9]	Input/output
11	DATA[10]	Input/output

Table D-3 Scan chain 1 cells (continued)

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

D.11 The watchpoint registers

The two watchpoint units, known as Watchpoint 0 and Watchpoint 1, each contain three pairs of registers:

- address value and address mask
- data value and data mask
- control value and control mask.

Each register is independently programmable, and has a unique address. The function and mapping of the registers is shown in Table D-4.

Table D-4 Function and mapping of EmbeddedICE registers

Address	Width	Function
00000	3	Debug control
00001	5	Debug status
00100	6	Debug comms control register
00101	32	Debug comms data register
01000	32	Watchpoint 0 address value
01001	32	Watchpoint 0 address mask
01010	32	Watchpoint 0 data value
01011	32	Watchpoint 0 data mask
01100	9	Watchpoint 0 control value
01101	8	Watchpoint 0 control mask
10000	32	Watchpoint 1 address value
10001	32	Watchpoint 1 address mask
10010	32	Watchpoint 1 data value
10011	32	Watchpoint 1 data mask
10100	9	Watchpoint 1 control value
10101	8	Watchpoint 1 control mask

D.11.1 Programming and reading watchpoint registers

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

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

This setup is shown in Figure D-5.

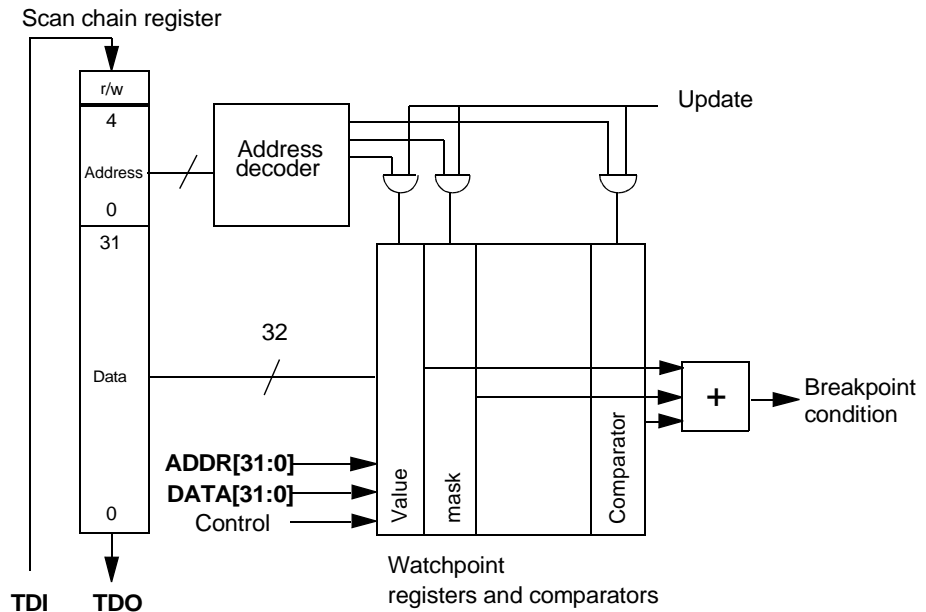


Figure D-5 EmbeddedICE block diagram

The data to be written is shifted into the 32-bit data field. The address of the register is shifted into the 5-bit address field. A 1 is shifted into the read/write bit.

A register is read by shifting its address into the address field and by shifting a 0 into the read/write bit. The 32-bit data field is ignored.

The register addresses are shown in Table D-4 on page D-27.

———— **Note** ————

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

D.11.2 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.

D.11.3 The control registers

The control value and control mask registers are mapped identically in the lower eight bits, as shown in Figure D-6.

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

Figure D-6 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 D-5 on page D-30.

Table D-5 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 that allows the watchpoint to be dependent upon some external condition. The **DBGEXT** input for Watchpoint 0 is labelled **DBGEXT[0]**. The **DBGEXT** input for Watchpoint 1 is labelled **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 EmbeddedICE, the **CHAINOUT** output of Watchpoint 1 is connected to the **CHAIN** input of Watchpoint 0. The **CHAINOUT** output is derived from a register. The address/control field comparator drives the write enable for the register. The input to the register is the value of the data field comparator. The **CHAINOUT** register is cleared when the control value register is written or when **nTRST** is LOW.

RANGE can be connected to the range output of another watchpoint register. In the ARM7TDMI-S EmbeddedICE, the **DBGRNG** output of Watchpoint 1 is connected to the **RANGE** input of Watchpoint 0. Connection allows 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.

D.12 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.
- *Software breakpoints* (see page D-33) monitor a particular bit pattern being fetched from any address. One EmbeddedICE watchpoint can therefore be used to support any number of software breakpoints.

Software breakpoints can normally be set only in RAM because a special bit pattern chosen to cause a software breakpoint has to replace the instruction.

D.12.1 Hardware breakpoints

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

1. Program its address value register with the address of the instruction to be breakpointed.
2. For an ARM-state breakpoint, program bits [1:0] of the address mask register to 11. For a breakpoint in Thumb state, program bits [1:0] of the address mask register to 01.
3. Program the data value register only when you require a data-dependent breakpoint, that is only when you need to match the actual instruction code fetched as well as the address. If the data value is not required, program the data mask register to 0xffffffff (all bits to 1). Otherwise program it to 0x00000000.
4. Program the control value register with **PROT[0]** = 0.
5. Program the control mask register with **PROT[0]** = 0.
6. When you need 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 2.

D.12.2 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 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 wish 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 —————

There is no need to program the address value register.

Setting the breakpoint

To set the software breakpoint:

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

Clearing the breakpoint

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

D.13 Programming watchpoints

To make a watchpoint unit cause watchpoints (on data accesses):

1. Program its address value register with the address of the data access to be watchpointed.
2. Program the address mask register to 0x00000000.
3. Program the data value register only if you require a data-dependent watchpoint, that is, only if you need to match the actual data value read or written as well as the address. If the data value is irrelevant, program the data mask register to 0xffffffff (all bits set to 1). Otherwise program the data mask register to 0x00000000.
4. Program the control value register with **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 may 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 wish 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 instance, you can provide simple range breakpoints by setting one or more of the address mask bits.

D.14 The debug control register

The debug control register is 3 bits wide. Writing control bits occurs during a register write access (with the read/write bit HIGH). Reading control bits occurs during a register read access (with the read/write bit LOW).

Figure D-7 shows the function of each bit in this register.

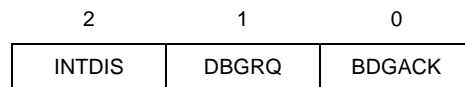


Figure D-7 Debug control register format

Bit 2 If bit 2 (**INTDIS**) is asserted, the interrupt signals to the processor are inhibited. So, both IRQ and FIQ are disabled when the processor is in debug state (**DBGACK** =1), or when **INTDIS** is forced.

Table D-6 shows interrupt signal control.

Table D-6 Interrupt signal control

DBGACK	INTDIS	Interrupts
0	0	permitted
1	x	inhibited
x	1	inhibited

Bits 1:0 These bits allow the values on **DBGRQ** and **DBGACK** to be forced.

As shown in Figure D-9 on page D-37, the value stored in bit 1 of the control register is synchronized and then ORed with the external **DBGRQ** before being applied to the processor.

In the case of **DBGACK**, the value of **DBGACK** from the core is ORed with the value held in bit 0 to generate the external value of **DBGACK** seen at the periphery of the ARM7TDMI-S. This allows the debug system to signal to the rest of the system that the core is still being debugged even when system-speed accesses are being performed (in which case the internal **DBGACK** signal from the core is LOW).

D.15 The debug status register

The debug status register is 5 bits wide. If it is accessed for a write (with the read/write bit set HIGH), the status bits are written. If it is accessed for a read (with the read/write bit LOW), the status bits are read. The format of the debug status register is shown in Figure D-8.

4	3	2	1	0
TBIT	TRANS[1]	IFEN	DBGRQ	DBGACK

Figure D-8 Debug status register format

The function of each bit in this register is as follows:

- Bit 4 allows **TBIT** to be read. This enables the debugger to determine the processor state, and therefore which instructions to execute.
- Bit 3 allows the state of the **TRANS[1]** signal from the core to be read. This state allows the debugger to determine whether a memory access from the debug state has completed.
- Bit 2 allows the state of the core interrupt enable signal (**IFEN**) to be read.
- Bits 1:0 allow 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 D-9 on page D-37.

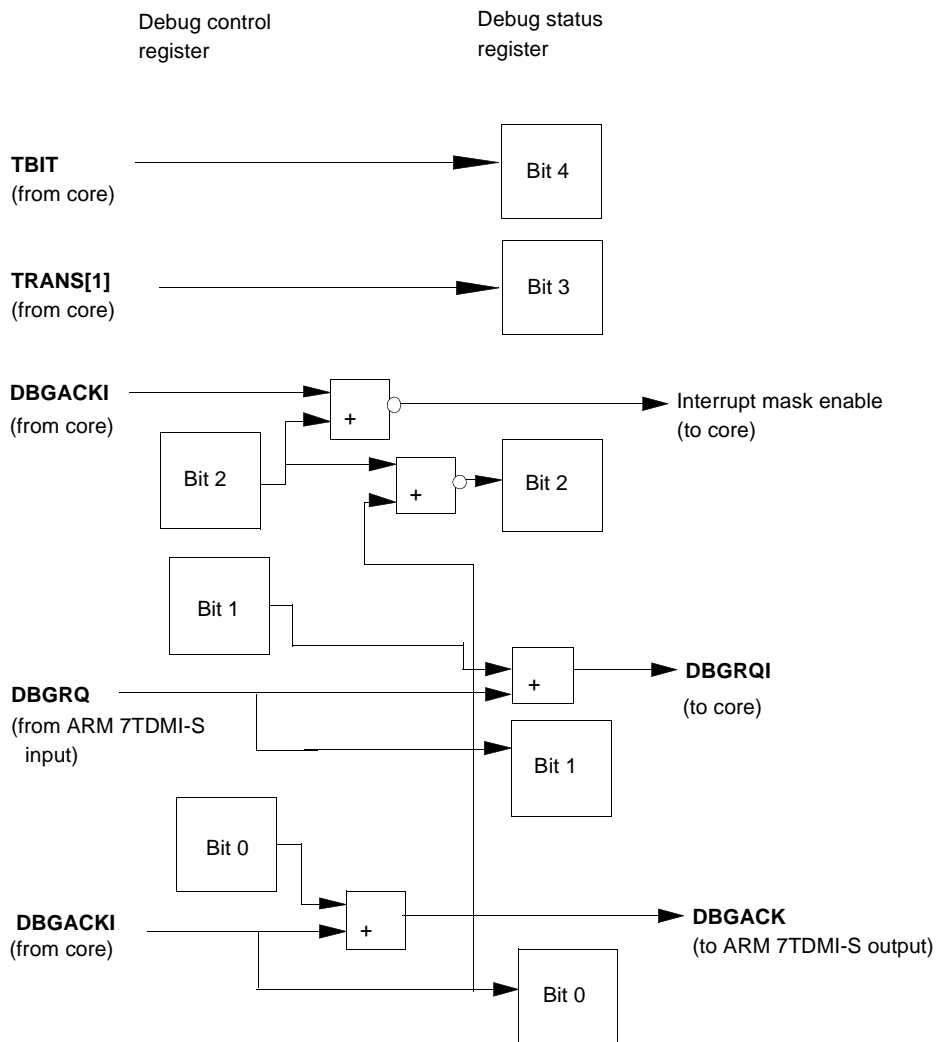


Figure D-9 Debug control and status register structure

D.16 Coupling breakpoints and watchpoints

Watchpoint units 1 and 0 can be coupled 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.

D.16.1 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
 $Dv[31:0]$ 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
 $Cv[8:0]$ 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, other watchpoint registers, and the **DBGEXT** signal.

CHAINOUT signal

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

$$\text{WHEN } ((\{Av[31:0], Cv[4:0]\} \text{ XNOR } \{A[31:0], C[4:0]\}) \text{ OR } \{Am[31:0], Cm[4:0]\} == 0xFFFFFFFF)$$

$$\text{CHAINOUT} = (((\{Dv[31:0], Cv[6:4]\} \text{ XNOR } \{D[31:0], C[7:5]\}) \text{ OR } \{Dm[31:0], Cm[7:5]\}) == 0x7FFFFFFF)$$

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

———— Note ————

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

Take, for example, the request by a debugger to breakpoint on the instruction at location **YYY** when running process **XXX** in a multiprocess system. If the current process ID is stored in memory, you can implement the above function with a watchpoint and breakpoint chained together. The watchpoint address points to a known memory location containing the current process ID, the watchpoint data points to the required process ID, and the **ENABLE** bit is set to off.

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

D.16.2 DBGRNG signal

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

$$\begin{aligned} \text{DBGRNG} = & (((\{A_v[31:0], C_v[4:0]\} \text{ XNOR } \{A[31:0], C[4:0]\}) \text{ OR} \\ & \{A_m[31:0], C_m[4:0]\}) == 0\text{xFFFFFFFF}) \text{ AND} \\ & (((\{D_v[31:0], C_v[7:5]\} \text{ XNOR } \{D[31:0], C[7:5]\}) \text{ OR} \\ & D_m[31:0], C_m[7:5]\}) == 0\text{x7FFFFFFFF}) \end{aligned}$$

The **DBGRNG** output of watchpoint register 1 provides the **RANGE** input to watchpoint register 0. This **RANGE** input allows two breakpoints to be coupled 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.

D.17 Disabling EmbeddedICE

You can disable EmbeddedICE by wiring the **DBGEN** input LOW.

When **DBGEN** is LOW:

- **DBGBREAK** and **DBGGRQ** are forced LOW to the core
- **DBGACK** is forced LOW from the ARM7TDMI-S
- interrupts pass through to the processor uninhibited.

D.18 EmbeddedICE timing

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

Refer to Chapter 7 *AC Parameters* for details of the required setup and hold times for these signals.

Index

The items in this index are listed in alphabetic order, with symbols and numerics appearing at the end. The references given are to page numbers.

A

ABORT A-2, B-8

Abort

- data 2-19, 5-8, D-24
- exception 2-19
- handler 2-20, 5-8
- mode 2-7
- prefetch 2-19, 2-20, D-24
- vector D-22

Aborted watchpoint D-23

AC

- timing diagrams 7-2-7-6
- timing parameters 7-7-7-8

Access

- memory D-17, D-19
- non-memory C-4
- system speed D-21, D-35
- watchpointed D-22, D-24

ADD instruction 2-13

ADDR 3-10, A-2

Address class signals
timing B-9

Address mask register D-27, D-29

Address value register D-27

ARM state 1-4

- register set 2-8, 2-11

ARM7TDMI-S

- architecture 1-4
- block diagram 1-6
- core diagram 1-7
- functional diagram 1-8
- overview 1-2
- porting considerations B-9
- signals compared to
ARM7TDMI B-2

ATPG scan interface B-7

B

Banked registers 2-8, D-16

Barrel shifter 6-10

Big-endian format 2-4

BL instruction 2-8, 6-7, 6-8

Boundary-scan

- chain cells D-5
- interface D-5

Branch and exchange. *See* BX instruction

Branch instruction 6-7

Branch with link. *See* BL instruction

Breakpoints 5-6, 5-7, 5-8, 5-13, D-13,
D-18, D-21

- data-dependent D-32
- entering debug state D-21
- externally-generated 5-6
- generating D-34
- hardware D-32

- programming D-32
 - software D-32, D-33
 - with prefetch abort D-24
- Bus interface
- cycle types 3-4
 - signals 3-3
- BX instruction 2-3, 6-9
- BYPASS instruction D-9
- Bypass register D-9, D-10
- ## C
- CAPTURE-DR
- cycle D-13
 - state D-7, D-8, D-9, D-10, D-11
- CAPTURE-IR state D-6, D-11
- CDP 4-10
- CFGBIGEND 5-9, **A-2**
- CHAIN D-30–D-39
- CHAINOUT D-30, D-38–D-39
- CLK 4-4, 4-5, 5-6, 5-10, **A-2**, B-8, B-9, D-5, D-7, D-10, D-14, D-41
- CLKEN **3-17**, 4-4, 4-5, 5-10, **A-2**, B-8, D-14, D-17, D-18
- Clock
- domains 5-10
 - maximum skew 7-7
 - system 5-2
 - test 5-2
- CMP instruction 2-13
- Code density 1-4, 1-5
- Condition code flags 2-14
- Configuration input timing 7-4
- Control mask register D-27, D-29
- Control value register D-27, D-29
- Coprocessors
- absence of external 4-14
 - absent 6-29
 - availability 4-3
 - busy-waiting 4-8
 - connecting 4-12–4-13
 - data operations 4-10
 - data processing operations 6-22
 - expansion interface signals B-2, B-8
 - handshaking 4-6
 - interface 4-1
 - interface signals 4-4
 - load and store operations 4-11
 - load coprocessor register 6-23
 - register transfer 6-27, 6-28
 - register transfer instructions 4-9, 5-16
 - signaling 4-7
 - store coprocessor register 6-25
 - timing 7-4
- CPA 4-2, 4-4, 4-6, 4-7, 4-14, 6-22, 6-23, 6-25, 6-29, **A-2**
- CPB 4-2, 4-4, 4-6, 4-7, 4-14, 6-22, 6-23, 6-25, 6-29, **A-2**
- CPnCPI 4-2, 4-4, 4-5, 4-6
- CPnI **A-2**
- CPnMREQ 4-4, 4-5, 4-14, **A-3**
- CPnOPC 4-4, 4-5, 4-14, **A-3**
- CPnTRANS 4-4, 4-14, 4-16, **A-3**
- CPSEQ 4-4, 4-14, **A-3**
- CPSR 2-8, 2-12, 2-14, 2-15, 2-18, 2-20, 6-21
- F flag 2-19
 - format 2-14
 - mode D-22
- CPTBIT 2-15, 3-12, 4-4, 4-5, 4-14, 6-3, **A-3**
- Current program status register. *See* CPSR
- ## D
- Data
- abort 2-19, 5-8, D-24
 - byte 2-6
 - halfword 2-6
 - multiplexing 4-13
 - operations 6-10
 - types 2-6
 - word 2-6
- Data mask register D-27, D-29
- Data swap instruction 6-20
- Data value register D-27
- DBGACK 5-6, 5-14, **A-3**, D-17, D-18, D-19, D-20, D-35, D-36, D-40
- DBGBREAK 5-6, 5-14, 7-5, **A-3**, D-3, D-13, D-17, D-21, D-40
- DBGCOMMRX **A-3**
- DBGCOMMTX **A-4**
- DBGGEN 5-14, **A-4**, D-40
- DBGEXT **A-4**, D-30, D-32, D-33, D-34, D-41
- DBGnEXEC **A-4**
- DBGnTDOEN **A-4**
- DBGnTRST **A-4**, D-5
- DBGnRNG **A-4**, D-30, D-39
- DBGnRQ 5-6, 5-8, 5-9, 5-14, **A-4**, D-35, D-36, D-40
- DBGnTCKEN 5-10, **A-4**, B-8, D-5, D-7, D-10, D-14, D-17
- DBGnTDI **A-4**, B-8, D-7, D-8, D-9, D-10, D-11, D-13
- DBGnTDO **A-4**, D-5, D-7, D-8, D-9, D-10, D-11, D-13
- DBGTMS **A-4**, B-8, D-7
- Debug
- actions 5-9
 - breakpoints 5-7
 - comms channel 5-15, 5-16–5-17
 - entry from ARM state D-15
 - entry from Thumb state D-15
 - expansion signals B-8
 - extensions 5-2
 - hardware extensions 5-5
 - host 5-4
 - interface 5-1, 5-2
 - interface signals 5-2, 5-6
 - message transfer 5-16–5-17
 - Multi-ICE 5-2
 - request 5-7, 5-8, D-21, D-22
 - state 5-2
 - state, entry from breakpoint D-21
 - state, exiting from D-18
 - state, pipeline behavior on exit D-19
 - state, processor restart on exit D-9
 - support 5-12
 - systems 5-4
 - target 5-4
 - timing 7-5
 - watchpoints 5-8
- Debug comms control register 5-15
- Debug comms data read register 5-15
- Debug comms data write register 5-15
- Debug control register 5-12, D-35
- Debug status register 5-12, D-36
- Determining
- core state 5-11
 - system state 5-11
- Device identification code D-8, D-10
- Disabling EmbeddedICE 5-14, D-40
- ## E
- EmbeddedICE 5-1, 5-5
- breakpoints, coupling with watchpoints D-38
 - breakpoints, hardware D-32
 - breakpoints, programming D-32
 - breakpoints, software D-32, D-33
 - control registers D-29
 - debug communications channel 5-15
 - debug status register 5-11, D-15
 - disabling 5-14, D-40
 - implications of module removal C-2
 - macrocell 1-6, 5-12
 - operation 5-12
 - overview 5-12
 - programming 5-8, D-3
 - register D-13
 - registers, accessing D-4

- TAP controller 1-6
- timing D-41
- watchpoint registers D-27–D-31
- watchpoints, coupling with breakpoints D-38
- EmbeddedICE control register D-20
- EmbeddedICE debug control register D-35
- EmbeddedICE debug status register D-36
- EmbeddedICE status register D-17
- Exception
 - at watchpoint D-21
- Exception entry
 - ARM state 2-18
 - Thumb state 2-18
- Exceptions 2-17–2-23
 - abort 2-19
 - action on entry 2-18
 - action on exit 2-18
 - data abort 2-23
 - FIQ 2-19
 - IRQ 2-19
 - priorities 2-23
 - vectors 2-22
- External coprocessors 4-14
- F**
- F bit 2-25
 - FIQ disable 2-15
- F flag, CPSR 2-19
- FIQ 4-8
 - exception 2-19
 - mode 2-7
- FIQ mode
 - defined 2-19
 - registers D-16
 - See also* Interrupts
- Flags, condition code 2-14
- Formats
 - big-endian 2-4
 - little-endian 2-4
- H**
- Hardware breakpoints D-32
- High registers
 - accessing from Thumb state 2-13
 - described 2-13
- I**
- I bit 2-25
 - IRQ disable 2-15
- ID register D-5, D-10
- IDCODE instruction D-5, D-8, D-10, D-11
- Identification register. *See* ID register
- Instruction cycle
 - operations 6-1
 - timings 6-1
- Instruction pipeline 1-2
- Instruction register D-6, D-8, D-9, D-10
- Instruction set
 - ARM 1-4
 - ARM, summary 1-9–1-16
 - Thumb 1-4
 - Thumb, summary 1-17–1-19
- Instructions
 - ADD 2-13
 - BL 2-8, 6-7
 - BL (ARM) 6-8
 - BL (Thumb) 6-8
 - branch 6-7
 - breakpointed 5-7, 5-8, D-18
 - BX 2-3, 6-9
 - BYPASS D-9
 - CDP 4-10
 - CMP 2-13
 - conditional 5-7
 - coprocessor load and store 4-11
 - coprocessor register transfer 4-9, 5-16
 - cycle counts 6-5
 - cycle timings 6-1–6-30
 - data swap 6-20
 - IDCODE D-8
 - LDC 4-11
 - LDM 2-20, 5-8, 6-17, 6-19
 - LDR 2-20, 6-14, 6-20
 - LDRT 6-14
 - length 2-5, 6-3
 - long branch with link (Thumb) 6-8
 - MCR 4-9
 - MLA 6-12
 - MOV 2-13
 - MRC 4-9, 5-16
 - MUL 6-12, C-4
 - privileged 4-16
 - SCAN_N D-7, D-11
 - STC 4-11
 - STM 2-20, 5-8, 6-19
 - STR 2-20, 6-16, 6-20
 - STRT 6-16
 - SWI 6-21
 - SWP 2-20
 - system speed D-17, D-23
 - undefined 6-29
 - undefined, handling 4-15
 - unexecuted 6-30
 - width 6-3
- Integer multiplication 6-12
- Interface
 - ATPG scan B-7
 - boundary-scan D-5
 - coprocessor 4-1
 - debug 5-2
 - JTAG D-3
 - memory 1-3, 3-2
 - signals B-2
- Interrupts D-24
 - disable bits 2-15
 - disable flags 2-18
 - disabling 2-14
 - enabling 2-14
 - latency 2-23, 2-24
 - latency calculation 2-24
- INTEST
 - instruction D-8, D-11
 - mode D-13
- IRQ 4-8
 - exception 2-19
 - mode 2-7
- IRQ disable
 - I bit 2-15
- IRQ mode
 - definition 2-19
 - See also* Interrupts
- J**
- JTAG
 - interface 5-2, 5-5, D-3
 - state machine D-4
- JTAG instructions
 - BYPASS D-9
 - IDCODE D-5, D-8, D-10, D-11
 - INTEST D-8, D-11
 - public D-7
 - RESTART D-9, D-17, D-18
 - SCAN_N D-7, D-11
 - SCAN_N TAP D-13
 - SCAN_N TAP controller D-13
 - TAP D-11
- L**
- LDC 4-11
- LDM 5-8
 - instruction cycle operations 6-17
- LDM instruction 2-20, 6-17, 6-19
 - cycle operation 6-17
- LDR
 - instruction 2-20, 6-14, 6-20

Index

LDRT instruction 6-14
Link bit 6-7
Link register 2-8, 2-9
Little-endian format 2-4
Load coprocessor register 6-23
Load multiple registers. *See* LDM
Load Multiple. *See* LDM
Load register. *See* LDR
LOCK 6-20, **A-5**
Low registers 2-13
LR 2-8, 2-11, 2-12

M

MCR 4-9
Memory
 access cycles 2-19
 access from debug state D-17, D-19
 big-endian format 2-4
 byte and halfword accesses 3-14
 formats 2-4
 interface 1-3, 3-2
 little-endian format 2-4
Minimum interrupt latencies 2-24
MLA
 instruction 6-12
 opcode support C-3
MLAL instruction 6-13
Mode bits 2-15
Modes
 abort 2-7, D-22, D-23, D-24
 FIQ 2-7
 FIQ, defined 2-19
 IRQ 2-7
 IRQ, defined 2-19
 operating 2-7
 privileged 2-7, 2-15, 2-19
 PSR bit values 2-15
 supervisor 2-7, 2-21
 system 2-7, D-18
 undefined 2-7
 user 2-7
MOV instruction 2-13
MRC instruction 4-9, 5-16
MUL
 instruction 6-12, C-4
 instruction cycle operations 6-12
 opcode support C-3
MUL32 C-3–C-5
 instruction cycle times C-4
 performance C-5
MULL instruction 6-13
Multi-ICE 5-2
Multiplier C-3
 reduced-function variant C-3
 removing C-3

Multiply accumulate. *See* MLA
Multiply long. *See* MULL
Multiply. *See* MUL
Multiply-accumulate long. *See* MLAL

N

nCPI 4-8
nFIQ 5-9, **A-5**
nIRQ 5-9, **A-5**
nRESET 2-25, 4-4, 4-5, 5-9, **A-5**

O

Operating modes 2-7
Operating state
 ARM 2-3
 CPTBIT 2-15
 switching 2-3
 T bit 2-15
 Thumb 2-3

P

PC 2-3, 2-8, 2-11, 2-12, 2-18, 2-20, 2-24
Pipeline 1-2
 follower 4-5
Porting considerations
 ARM7TDMI-S B-9
Prefetch
 abort D-24
Prefetch abort 2-19, 2-20, D-24
Privileged instructions 4-16
Privileged modes 2-7, 2-15, 2-19
Processor state, determining D-15
Program counter. *See* PC
Program status registers. *See* PSR
Programming EmbeddedICE 5-8
PROT 3-11, 6-3, **A-5**, D-30, D-32,
 D-33, D-34
Protocol converter 5-4
PSR 2-14, 6-10
 control bits 2-15, 2-16
 format 2-14
 mode bit values 2-15
 reserved bits 2-16
Public instructions D-7

R

RANGE D-30, D-32, D-33, D-34,

D-38, D-39

Range checking D-30
RDATA 3-13, 4-4, 4-5, **A-5**, B-8
Register set
 ARM state 2-11
 Thumb state 2-11
Register transfer, coprocessor 6-27,
 6-28
Registers
 address mask D-27, D-29
 address value D-27
 ARM state 2-8
 banked 2-8
 bypass D-10
 debug comms control 5-15
 FIQ mode D-16
 general-purpose 2-8
 high registers 2-13
 ID D-10
 instruction D-8
 link 2-9
 status 2-8
 Thumb 2-11
 user mode D-16
 watchpoint D-27, D-28
Registers, debug
 bypass D-9, D-10
 comms control 5-15
 comms data read 5-15
 comms data write 5-15
 control 5-12, D-35
 control mask D-27, D-29
 control value D-27, D-29
 data mask D-27, D-29
 data value D-27
 EmbeddedICE D-13
 EmbeddedICE control D-20, D-29
 EmbeddedICE debug status 5-11,
 D-15
 EmbeddedICE status D-17
 EmbeddedICE, accessing D-4
 ID D-5, D-10
 instruction D-6, D-8, D-9, D-10
 scan path select D-7, D-10, D-11
 status 5-12, D-36
 test data D-10
Reserved bits, PSR 2-16
Reset 2-25
 TAP controller D-5

RESTART instruction D-9, D-17, D-18
 Restart on exit from debug D-9
 Return address, calculation D-23
 RTCK 5-2
 RUN-TEST/IDLE state D-9, D-18

S

Saved program status register. *See* SPSR

Scan

- general timing 7-6
- input cells D-8
- interface timing D-25
- limitations D-3
- output cells D-8
- path D-3, D-7
- test D-10

Scan cells D-8, D-9, D-12

Scan chains D-3–D-4, D-7, D-8, D-11,
 D-25–D-31

- number allocation D-12
- scan chain 1 D-3, D-10, D-12,
 D-13, D-15, D-17, D-21,
 D-25
- scan chain 2 D-3, D-4, D-10, D-12,
 D-13, D-28

Scan path select register D-7, D-10,
 D-11

SCAN_N D-7
 instruction D-7, D-11

SCAN_N TAP
 instruction D-13

SCANENABLE **A-5**, B-7

SCANIN **A-5**, B-7

SCANOUT **A-6**, B-7

Serial interface, JTAG 5-2, 5-5

SHIFT-DR

- cycle D-13
- state D-5, D-7, D-8, D-9, D-10,
 D-11

SHIFT-IR state D-11

Signal types 3-3, 4-4

- address class 3-10
- clocking and clock control 3-17
- data timed 3-13

Signals A-2–A-6

- ABORT 3-13, **A-2**, B-8
- ADDR 3-10, **A-2**
- address class, timing B-9
- bus interface 3-3
- CFGBIGEND 5-9, **A-2**
- CHAIN D-30–D-39
- CHAINOUT D-30, D-38–D-39
- CLK 4-4, 4-5, 5-6, 5-10, **A-2**, B-8,
 D-5, D-7, D-10, D-14,
 D-41

- CLKEN **3-17**, 4-4, 4-5, 5-10, **A-2**,
 B-8, D-14, D-17, D-18
- clock and clock control 4-4
- coprocessor interface 4-4
- CPA 4-2, 4-6, 6-22, 6-23, 6-25,
 6-29, **A-2**
- CPB 4-2, 4-6, 6-22, 6-23, 6-25,
 6-29, **A-2**

CPnCPI 4-2

CPnI **A-2**

CPnMREQ 4-4, **A-3**

CPnOPC 4-4, **A-3**

CPnTRANS 4-4, 4-16, **A-3**

CPSEQ 4-4, 4-14, **A-3**

CPTBIT 2-15, 3-12, 4-4, 4-5, 4-14,
 6-3, **A-3**

DBGACK 5-6, 5-14, **A-3**, D-17,
 D-18, D-19, D-20, D-35,
 D-36, D-40

DBGBREAK 5-6, 5-14, 7-5, **A-3**,
 D-3, D-13, D-17, D-21,
 D-40

DBGCOMMRX **A-3**

DBGCOMMTX **A-4**

DBGGEN 5-14, **A-4**, D-40

DBGEXT **A-4**, D-30, D-32, D-33,
 D-34, D-41

DBGnEXEC **A-4**

DBGnTDOEN **A-4**

DBGnTRST **A-4**, D-5

DBG RNG **A-4**, D-30, D-39

DBG RQ 5-6, 5-8, 5-9, 5-14, **A-4**,
 D-35, D-36, D-40

DBG TCKEN 5-10, **A-4**, B-8, D-5,
 D-7, D-10, D-14, D-17

DBG TDI **A-4**, B-8, D-7, D-8

DBG TDO **A-4**, D-5, D-7, D-8

DBG TMS **A-4**, B-8, D-7

debug interface 5-2, 5-6

ENABLE D-31

FIQ 4-8

interface B-2

IRQ 4-8

LOCK 3-12, 6-20, **A-5**

nCPI 4-8

nFIQ 5-9, **A-5**

nIRQ 5-9, **A-5**

nRESET 2-25, 4-4, 4-5, 5-9, **A-5**

PROT 3-11, 6-3, **A-5**, D-30, D-32,
 D-33, D-34

PROT, meanings 4-16

RANGE D-30, D-32, D-33, D-34,
 D-38, D-39

RDATA 3-13, 4-4, 4-5, **A-5**, B-8

RTCK 5-2

SCANENABLE **A-5**, B-7

SCANIN **A-5**, B-7

SCANOUT **A-6**, B-7

SIZE 3-11, 6-3, **A-6**, D-34

TCK 5-2, D-9

TRANS 5-9, 6-3, **A-6**, D-17, D-36

WDATA 3-13, 4-4, **A-6**, B-8

WRITE 3-10, 6-3, 6-23, 6-25, **A-6**,
 D-34

Single-step core operation D-8

SIZE 3-11, 6-3, **A-6**, D-34

Software breakpoints D-32, D-33
 clearing D-33
 setting D-33

Software interrupts. *See* SWI

SP 2-9, 2-11, 2-12

SPSR 2-8, 2-11, 2-12, 2-14, 2-18, D-22
 format 2-14

Stack pointer. *See* SP

State

- ARM 1-4
- debug 5-2
- Thumb 1-4

States

- CAPTURE-DR D-7, D-8, D-9,
 D-10, D-11
- CAPTURE-IR D-6, D-11
- core D-15
- processor operating 2-3
- RUN-TEST/IDLE D-9, D-18
- SHIFT-DR D-7, D-8, D-9, D-10,
 D-11
- SHIFT-IR D-11
- system D-15, D-17
- TAP D-12
- TAP controller 5-2
- UPDATE-DR D-7, D-8, D-9,
 D-11, D-12
- UPDATE-IR D-11

Status registers 2-8

STC 4-11

STM 5-8

- at debug speed D-17
- instruction 2-20, 6-19
- instruction cycle operations 6-19

Store coprocessor register 6-25

Store multiple registers. *See* STM

Store register. *See* STR

STR

- instruction 2-20, 6-16, 6-20
- instruction cycle operations 6-16

STRT instruction 6-16

Summary

- instruction set 1-9–1-19

Supervisor mode 2-7, 2-21

SWI 2-21

- instruction 6-21
- instruction cycle operations 6-21

SWP instruction 2-20

System

- state D-17

System speed

- access D-35
- instruction D-17, D-23

Index

System state, determining 5-11, D-17

T

T bit 2-15, 2-18, 2-25

TAP

instruction D-11
state D-12

TAP controller 5-5, 5-12, D-3, D-4,
D-5, D-9, D-12, D-13, D-17,
D-18

reset D-5
states 5-2

TAP state machine D-4

TBIT D-36

TCK 5-2, D-9

Test

clock 5-2
data D-8
INTEST D-13

Test Access Port *See* TAP

Test data D-9
registers D-10

Test mode D-8

Thumb

BL instruction 6-8
code 1-5
instruction set 1-4
long branch with link
instruction 6-8
operating state 2-3
registers 2-11
state 1-4

Timing

configuration input 7-4
coprocessor 7-4
debug 7-5
EmbeddedICE D-41
exception input 7-4
interrupts B-9
parameters B-8
scan 7-6
scan interface D-25

TRANS 5-9, 6-3, A-6, D-17, D-36

U

Undefined instructions 6-29
handling 4-15
instruction cycle operations 6-29
trap 2-21, 6-29

Undefined mode 2-7

Unexecuted instructions 6-30

Unused instruction codes D-9

UPDATE-DR

cycle D-13
state D-7, D-8, D-9, D-11, D-12

UPDATE-IR state D-11

User mode 2-7

W

Watchpointed

access D-22, D-24
memory access D-22

Watchpoints 5-6, 5-8, 5-12, 5-13, D-13,
D-21

aborted D-23
data-dependent D-34
entering debug state from D-21
externally-generated 5-6
generating D-34
programming D-34
register D-28
registers D-27
units D-27
with exception D-21, D-23

WDATA 3-13, 4-4, A-6, B-8

port 7-3

WRITE 6-3, 6-23, 6-25, A-6, D-34