

[查询ARM7TDMI-S供应商](#)

捷多邦，专业PCB打样工厂，24小时加
[急出货](#)

ARM7TDMI-S Microprocessor Core

Technical Manual

Preliminary

LSI LOGIC®



This document is preliminary. As such, it contains data derived from functional simulations and performance estimates. LSI Logic has not verified either the functional descriptions, or the electrical and mechanical specifications using production parts.

This document contains proprietary information of LSI Logic Corporation. The information contained herein is not to be used by or disclosed to third parties without the express written permission of an officer of LSI Logic Corporation.

Document DB14-000127-00, First Edition (March 2000)

This document describes LSI Logic Corporation's ARM7TDMI-S Microprocessor Core and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

To receive product literature, visit us at <http://www.lsillogic.com>.

LSI Logic Corporation reserves the right to make changes to any products herein at any time without notice. LSI Logic does not assume any responsibility or liability arising out of the application or use of any product described herein, except as expressly agreed to in writing by LSI Logic; nor does the purchase or use of a product from LSI Logic convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual property rights of LSI Logic or third parties.

Copyright © 2000 by LSI Logic Corporation. All rights reserved.

TRADEMARK ACKNOWLEDGMENT

LSI Logic logo design, CoreWare and CoreWare logo design, GigaBlaze, G10 and G10 logo design, G11 and G11 logo design, G12 and G12 logo design, Right-First-Time are trademarks or registered trademarks of LSI Logic Corporation. ARM is a registered trademark of Advanced RISC Machines Limited, used under license. All other brand and product names may be trademarks of their respective companies.

EH

Preface

This book is the primary reference and technical manual for the ARM7TDMI-S core. It contains a complete functional description and complete electrical specifications for the ARM7TDMI-S core.

Audience

This document assumes that you have some familiarity with microprocessors and related support devices. The people who benefit from this book are:

- Engineers and managers who are evaluating the processor for possible use in a system
 - Engineers who are designing the processor into a system
-

Organization

This document has the following chapters and appendixes:

- [Chapter 1, Introduction](#), introduces the ARM7TDMI-S core and summarizes the LSI Logic CoreWare program.
- [Chapter 2, Signal Descriptions](#), describes the signals that make up the external interface of the ARM7TDMI-S core.
- [Chapter 3, Programmer's Model](#), describes the operating states and registers of the ARM7TDMI-S core.
- [Chapter 4, Memory Interface](#), provides details on the ARM7TDMI-S memory interface.
- [Chapter 5, Coprocessor Interface](#), describes the ARM7TDMI-S coprocessor interface.

- [Chapter 6, Debug Interface](#), describes the ARM7TDMI-S debug interface.
- [Chapter 7, Instruction Cycle Timing](#), provides the instruction cycle timing for the ARM7TDMI-S core.
- [Appendix A, Differences Between the ARM7TDMI-S and the ARM7TDMI](#), describes the differences between the ARM7TDMI-S and the ARM7TDMI hard macrocell.
- [Appendix B, Detailed Debug Operation](#), provides extensive detail of the debug operation of the ARM7TDMI-S core.

Related Publications

ARM7TDMI Microprocessor Core Technical Manual, Document No. DB14-000058-02

ARM Architecture Reference Manual (ARM DDI 0100).

ARM7TDMI Data Sheet (ARM DDI 0029).

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

Conventions Used in This Manual

The first time a word or phrase is defined in this manual, it is *italicized*.

The word *assert* means to drive a signal true or active. The word *deassert* means to drive a signal false or inactive. Signals that are active LOW contain a lowercase “n.”

Hexadecimal numbers are indicated by the prefix “0x” —for example, 0x32CF. Binary numbers are indicated by the prefix “0b” —for example, 0b0011.0010.1100.1111.

Contents

Preface

Chapter 1	Introduction	
1.1	Introduction	1-1
	1.1.1 General Information	1-1
	1.1.2 LSI Logic's ARM7TDMI-S Implementation	1-2
1.2	ARM7TDMI-S Architecture	1-2
	1.2.1 Instruction Compression	1-2
	1.2.2 The Thumb Instruction Set	1-3
	1.2.3 Pipeline Architecture	1-5
	1.2.4 Memory Accesses	1-6
	1.2.5 Memory Interface	1-7
1.3	ARM7TDMI-S Core Instruction Set Summary	1-7
	1.3.1 ARM Instruction Summary	1-9
	1.3.2 Thumb Instruction Summary	1-18
1.4	CoreWare [®] Program	1-23

Chapter 2	Signal Descriptions	
2.1	Memory Interface	2-3
2.2	Coprocessor Interface	2-6
2.3	Clock and Control Signals	2-7
2.4	Debug Interface	2-9
2.5	ATPG Interface	2-11

Chapter 3	Programmer's Model	
3.1	Processor Operating States	3-1
3.2	Memory Formats	3-2
	3.2.1 Big-Endian Format	3-2

3.2.2	Little-Endian Format	3-2
3.3	Data Types	3-3
3.4	Operating Modes	3-3
3.5	Registers	3-4
3.5.1	The ARM State Register Set	3-4
3.5.2	The Thumb State Register Set	3-6
3.5.3	The Relationship Between ARM and Thumb State Registers	3-7
3.5.4	Accessing High Registers in Thumb State	3-8
3.5.5	Program Status Registers	3-9
3.6	Exceptions	3-11
3.6.1	Entering an Exception	3-12
3.6.2	Leaving an Exception	3-13
3.6.3	Fast Interrupt Request (FIQ)	3-13
3.6.4	Interrupt Request (IRQ)	3-14
3.6.5	Aborts (PABT and DABT)	3-14
3.6.6	Software Interrupt (SWI)	3-15
3.6.7	Undefined Instruction (UDEF)	3-16
3.6.8	Exception Vectors	3-17
3.6.9	Exception Priorities	3-17
3.7	Interrupt Latencies	3-18
3.7.1	Maximum Interrupt Latencies	3-18
3.7.2	Minimum Interrupt Latencies	3-18
3.8	Reset	3-19

Chapter 4

Memory Interface

4.1	About the Memory Interface	4-1
4.2	Bus Cycle Types	4-1
4.2.1	Nonsequential Cycles (N)	4-3
4.2.2	Sequential Cycles (S)	4-4
4.2.3	Internal Cycles	4-5
4.2.4	Merged I-S Cycles	4-6
4.2.5	Coprocessor Register Transfer Cycles	4-6
4.3	Bus Interface Signals	4-7
4.3.1	Addressing Signals	4-7
4.3.2	Data Timed Signals	4-9
4.3.3	Byte and Halfword Accesses	4-10

4.3.4	Write Operations	4-12
4.4	Use of CLKEN to Control Bus Cycles	4-13

Chapter 5

Coprocessor Interface

5.1	About Coprocessors	5-1
5.2	Coprocessor Availability	5-2
5.3	Coprocessor Interface Signals	5-3
5.4	Pipeline Following Signals	5-4
5.5	Coprocessor Interface Handshaking	5-5
5.5.1	The Coprocessor	5-5
5.5.2	Coprocessor Instructions in the ARM7TDMI-S Core	5-5
5.5.3	Coprocessor Signalling	5-6
5.5.4	Consequences of Busy-Waiting	5-7
5.5.5	Coprocessor Register Transfer Instructions	5-8
5.5.6	Coprocessor Data Operations	5-9
5.5.7	Coprocessor Load and Store Operations	5-10
5.6	Connecting Coprocessors	5-10
5.6.1	Connecting a Single Coprocessor	5-11
5.6.2	Connecting Multiple Coprocessors	5-11
5.7	Implementations Without External Coprocessors	5-12
5.8	Undefined Instructions	5-12
5.9	Privileged Instructions	5-13

Chapter 6

Debug Interface

6.1	Overview of the Debug Interface	6-1
6.1.1	Debug Stages	6-2
6.1.2	Clocks	6-2
6.1.3	Debug Interface Signals	6-3
6.2	Debug Systems	6-3
6.2.1	The Debug Host	6-4
6.2.2	The Protocol Converter	6-4
6.2.3	The ARM7TDMI-S Core	6-5
6.3	Entry into Debug State	6-6
6.3.1	Entry into Debug State on Breakpoint	6-7
6.3.2	Entry into Debug State on Watchpoint	6-7
6.3.3	Entry into Debug State on Debug Request	6-8
6.3.4	Action of the ARM7TDMI-S Core in Debug State	6-8

6.4	ARM7TDMI SCore Clock Domains	6-9
6.5	Determining the Core and System State	6-9
6.6	Overview of EmbeddedICE	6-9
6.7	The Debug Communications Channel	6-11
6.7.1	Debug Comms Channel Registers	6-11
6.7.2	Communications via the Comms Channel	6-12

Chapter 7

Instruction Cycle Timing

7.1	Introduction	7-1
7.2	Instruction Cycle Count Summary	7-3
7.3	Instruction Execution Times	7-4
7.3.1	Branch and ARM Branch with Link Instructions	7-4
7.3.2	Thumb Branch with Link Instructions	7-6
7.3.3	Branch and Exchange Instruction	7-7
7.3.4	Data Operations	7-8
7.3.5	Multiply and Multiply Accumulate Operations	7-10
7.3.6	Load Register Instruction	7-12
7.3.7	Store Register Instruction	7-13
7.3.8	Load Multiple Register Operations	7-13
7.3.9	Store Multiple Register Instructions	7-15
7.3.10	Data Swap Operations	7-16
7.3.11	Software Interrupt and Exception Entry	7-17
7.3.12	Coprocessor Data Processing Operation	7-18
7.3.13	Load Coprocessor Register (from Memory to Coprocessor)	7-19
7.3.14	Store Coprocessor Register (from Coprocessor to Memory)	7-21
7.3.15	Coprocessor Register Transfer (Move from Coprocessor to ARM Register)	7-23
7.3.16	Coprocessor Register Transfer (Move from ARM Register to Coprocessor)	7-24
7.3.17	Undefined Instructions and Coprocessor Absent	7-24
7.3.18	Unexecuted Instructions	7-25

Appendix A

Differences Between the ARM7TDMI-S and the ARM7TDMI

A.1	Interface Signals	A-1
A.2	ATPG Scan Interface	A-5

A.3	Timing Parameters	A-5
A.4	ARM7TDMI-S Design Considerations	A-5
A.4.1	Master Clock	A-6
A.4.2	JTAG Interface Timing	A-6
A.4.3	Interrupt Timing	A-6
A.4.4	Address Class Signal Timing	A-6

Appendix B

Detailed Debug Operation

B.1	Scan Chains and JTAG Interface	B-2
B.1.1	Scan Limitations	B-2
B.1.2	TAP State Machine	B-4
B.2	Resetting the TAP Controller	B-6
B.3	Instruction Register	B-6
B.4	Public Instructions	B-7
B.4.1	SCAN_N (0010)	B-7
B.4.2	INTEST (1100)	B-8
B.4.3	IDCODE (1110)	B-8
B.4.4	BYPASS (1111)	B-9
B.4.5	RESTART (0100)	B-9
B.5	Test Data Registers	B-9
B.5.1	Bypass Register	B-10
B.5.2	ARM7TDMI-S Device Identification (ID) Code Register	B-10
B.5.3	Instruction Register	B-10
B.5.4	Scan Path Select Register	B-11
B.5.5	Scan Chains 1 and 2	B-12
B.6	ARM7TDMI-S Core Clock Domains	B-13
B.7	Determining the Core and System State	B-14
B.7.1	Determining the Core State	B-14
B.7.2	Determining System State	B-16
B.7.3	Exit from Debug State	B-17
B.8	Behavior of the Program Counter During Debug	B-19
B.8.1	Breakpoints	B-20
B.8.2	Watchpoints	B-20
B.8.3	Watchpoint with Another Exception	B-20
B.8.4	Debug Request	B-21
B.8.5	System Speed Access	B-22

B.8.6	Summary of Return Address Calculations	B-22
B.9	Priorities and Exceptions	B-23
B.9.1	Breakpoint with Prefetch Abort	B-23
B.9.2	Interrupts	B-23
B.9.3	Data Aborts	B-24
B.10	Scan Interface Timing	B-24
B.11	The Watchpoint Registers	B-24
B.11.1	Programming and Reading Watchpoint Registers	B-25
B.11.2	Using the Watchpoint 0/1 Data and Address Mask Registers	B-26
B.11.3	Watchpoint 0/1 Control and Mask Registers	B-27
B.12	Programming Breakpoints	B-29
B.12.1	Hardware Breakpoints	B-30
B.12.2	Software Breakpoints	B-30
B.13	Programming Watchpoints	B-31
B.14	The Debug Control Register	B-32
B.15	The Debug Status Register	B-33
B.16	Coupling Breakpoints and Watchpoints	B-35
B.16.1	Breakpoint and Watchpoint Coupling Example	B-35
B.16.2	DBGRNG Signal	B-37
B.17	EmbeddedICE Issues	B-38

Figures

1.1	Processor Block Diagram	1-4
1.2	ARM7TDMI-S Core Diagram	1-5
1.3	ARM7TDMI-S Pipeline	1-6
2.1	ARM7TDMI-S Logic Diagram	2-2
3.1	Big-Endian Addresses of Bytes Within Words	3-2
3.2	Little-Endian Addresses of Bytes Within Words	3-3
3.3	Register Organization in ARM State	3-5
3.4	Register Organization in Thumb State	3-7
3.5	Thumb State to ARM State Register Mappings	3-8
3.6	Program Status Register Format	3-9
4.1	Simple Memory Cycle	4-2
4.2	Nonsequential Memory Cycle	4-3
4.3	Back-to-Back Memory Cycles	4-4
4.4	Sequential Access Cycles	4-5
4.5	Merged I-S Cycles	4-6
4.6	Data Replication	4-13
4.7	Use of CLKEN	4-14
5.1	Coprocessor Busy-Wait Sequence	5-7
5.2	Coprocessor Register Transfer Sequence	5-8
5.3	Coprocessor Data Operation Sequence	5-9
5.4	Coprocessor Load Sequence	5-10
5.5	Coprocessor Connections	5-11
6.1	Clock Synchronization	6-3
6.2	Typical Debug System	6-4
6.3	ARM7TDMI-S Core Block Diagram	6-5
6.4	Debug State Entry	6-6
6.5	The ARM7TDMI S TAP Controller, and EmbeddedICE	6-10
6.6	Debug Comms Control Register	6-11
B.1	ARM7TDMI SScan Chain Arrangements	B-2
B.2	Test Access Port Controller State Transitions	B-5
B.3	Debug Exit Sequence	B-19
B.4	General Scan Timing	B-24
B.5	EmbeddedICE Block Diagram	B-26
B.6	Watchpoint 0/1 Control Value Register	B-27
B.7	Watchpoint 0/1 Control Mask Register	B-27
B.8	Debug Control Register Format	B-32

B.9	Debug Status Register Format	B-33
B.10	Debug Control and Status Register Structure	B-35

Tables

1.1	Key to Tables	1-8
1.2	ARM Instruction Summary	1-9
1.3	Addressing Mode 2	1-13
1.4	Addressing Mode 2 (Privileged)	1-14
1.5	Addressing Mode 3	1-15
1.6	Addressing Mode 4 (Load)	1-15
1.7	Addressing Mode 4 (Store)	1-15
1.8	Addressing Mode 5	1-16
1.9	Oprnd2	1-16
1.10	Fields	1-17
1.11	Condition Fields	1-17
1.12	Thumb Instruction Summary	1-18
3.1	Mode Bit States	3-11
3.2	Exception Entry/Exit	3-12
3.3	Exception Vectors	3-17
4.1	Cycle Types	4-2
4.2	Burst Types	4-5
4.3	Transfer Widths	4-8
4.4	PROT Encoding	4-9
4.5	Significant Address Bits	4-11
4.6	Word Accesses	4-11
4.7	Halfword Accesses	4-11
4.8	Byte Accesses	4-12
5.1	Coprocessor Availability	5-3
5.2	Coprocessor Interface Signals	5-3
5.3	Handshaking Signals	5-5
5.4	Handshake Signal Connections	5-11
5.5	PROT1 Signal Meanings	5-13
7.1	Transaction Types	7-2
7.2	Instruction Cycle Counts	7-3
7.3	Branch Instruction Cycle Operations	7-5
7.4	Thumb Long Branch with Link	7-6
7.5	Branch and Exchange Instruction Cycle Operations	7-7
7.6	Data Operation Instruction Cycle Operations	7-9
7.7	Multiply Instruction Cycle Operations	7-10
7.8	Multiply Accumulate Instruction Cycle Operations	7-10

7.9	Multiply Long Instruction Cycle Operations	7-11
7.10	Multiply Accumulate Long Instruction Cycle Operations	7-11
7.11	Load Register Instruction Cycle Operations	7-12
7.12	Store Register Instruction Cycle Operations	7-13
7.13	Load Multiple Registers Instruction Cycle Operations	7-14
7.14	Store Multiple Registers Instruction Cycle Operations	7-16
7.15	Data Swap Instruction Cycle Operations	7-17
7.16	Software Interrupt Instruction Cycle Operations	7-18
7.17	Coprocessor Data Operation Instruction Cycle Operations	7-19
7.18	Load Coprocessor Register Instruction Cycle Operations	7-20
7.19	Store Coprocessor Register Instruction Cycle Operations	7-22
7.20	Coprocessor Register Transfer (MRC)	7-23
7.21	Coprocessor Register Transfer (MCR)	7-24
7.22	Undefined Instruction Cycle Operations	7-25
7.23	Unexecuted Instruction Cycle Operations	7-25
A.1	ARM7TDMI SSignals and ARM7TDMI Hard Macrocell Equivalents	A-2
B.1	Scan Chain 1 Cells	B-3
B.2	Public Instructions	B-7
B.3	Scan Chain Number Allocation	B-11
B.4	Function and Mapping of EmbeddedICE Registers	B-25

Chapter 1

Introduction

This chapter introduces the core architecture and shows block, core, and functional diagrams. It contains the following sections:

- ◆ [Section 1.1, “Introduction”](#)
 - ◆ [Section 1.2, “ARM7TDMI-S Architecture”](#)
 - ◆ [Section 1.3, “ARM7TDMI-S Core Instruction Set Summary”](#)
 - ◆ [Section 1.4, “CoreWare® Program”](#)
-

1.1 Introduction

This section introduces the overall capabilities of LSI Logic Corporation’s ARM7TDMI-S core implementation and highlights its features.

1.1.1 General Information

The ARM7TDMI-S architecture is a member of the Advanced RISC Machines (ARM) family of general-purpose 32-bit microprocessors, which offer high performance for very low power consumption and price.

The ARM® architecture is based on Reduced Instruction Set Computer (RISC) principles, and the instruction set and related decode mechanism are much simpler than those of microprogrammed Complex Instruction Set Computers. This simplicity results in a high instruction throughput and impressive real time interrupt response from a small and cost effective chip.

Pipelining is employed so that all parts of the processing and memory systems can operate continuously. Typically, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.

The ARM memory interface has been designed to allow the performance potential to be realized without incurring high costs in the memory system. Speed critical control signals are pipelined to allow system control functions to be implemented in standard low-power logic, and these control signals facilitate the exploitation of the fast local access modes offered by industry standard dynamic RAMs.

1.1.2 LSI Logic's ARM7TDMI-S Implementation

The ARM7TDMI-S Microprocessor Core described in this manual is LSI Logic Corporation's proprietary implementation of the ARM7TDMI-S microprocessor. LSI Logic has optimized this synthesizable version to facilitate implementation of complex system-on-a-chip ASICs in LSI Logic's state-of-the-art ASIC flows.

The LSI Logic implementation of the ARM7TDMI-S core is fully hardware and software compatible with the ARM7TDMI-S core. To implement full scan, LSI Logic has added four additional test signals to the core (for a full description of these signals, see [Chapter 2](#), “[Signal Descriptions](#)”).

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 core is an implementation of the ARMv4T architecture. For full details on 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 can manipulate 32-bit integers with single instructions, and address a large address space much more efficiently than a 16-bit architecture. When processing 32-bit data, a 16-bit architecture takes at least two instructions to perform the same task as a single 32-bit instruction.

When a 16-bit architecture has only 16-bit instructions and a 32-bit architecture has only 32-bit instructions, overall the 16-bit architecture

has higher code density and greater than half the performance of the 32-bit architecture.

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

The ARM7TDMI-S architecture and core are illustrated in the following figures:

- ◆ Figure 1.1 shows the ARM7TDMI-S block diagram
- ◆ Figure 1.2 shows the ARM7TDMI-S core block diagram

Figure 1.1 Processor Block Diagram

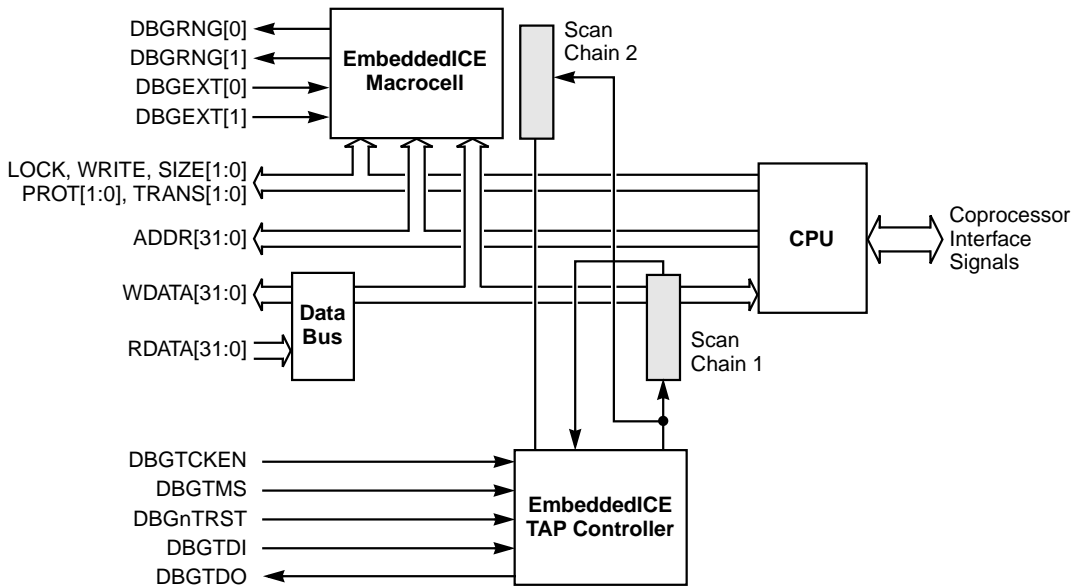
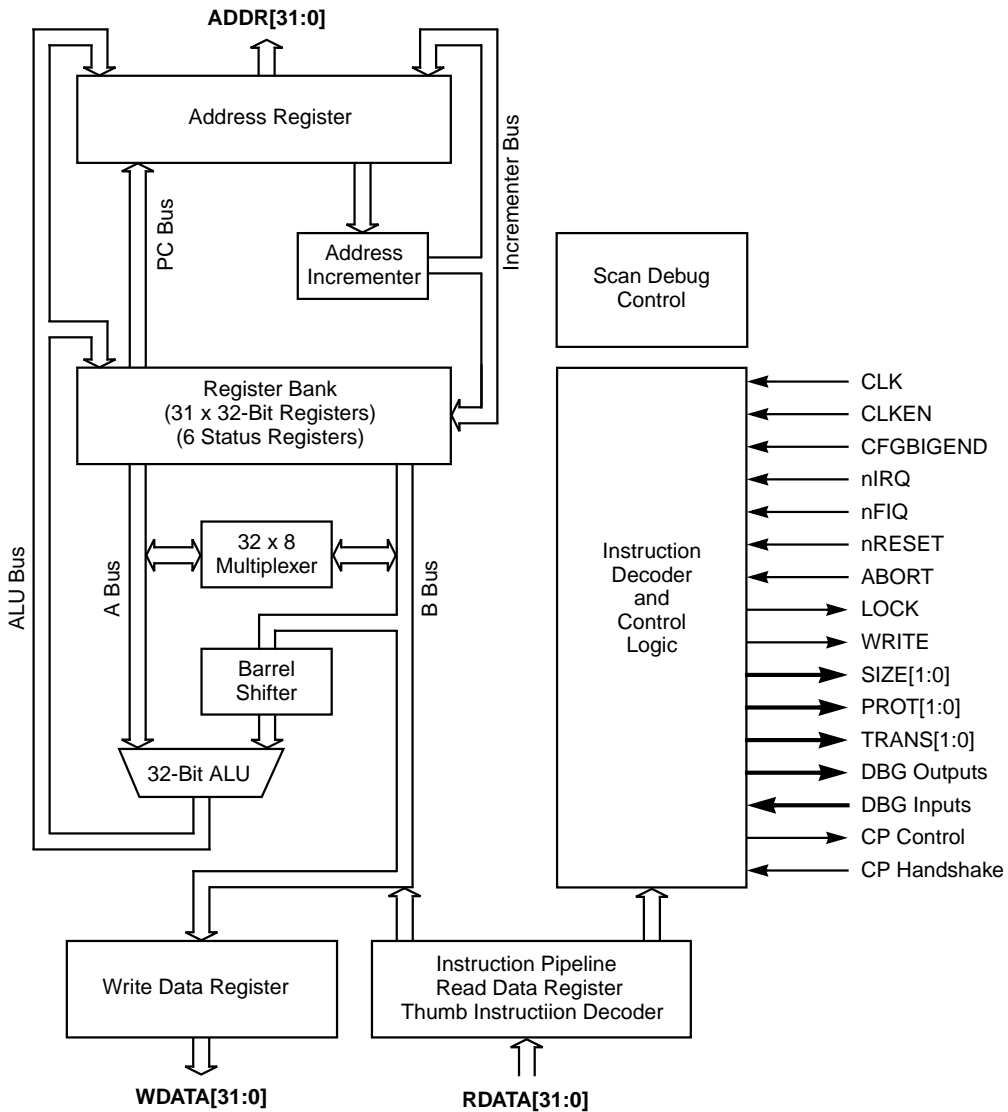


Figure 1.2 ARM7TDMI-S Core Diagram

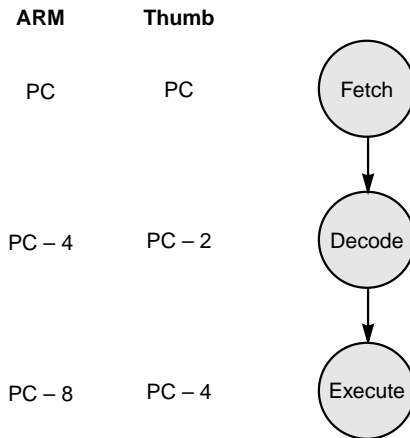


1.2.3 Pipeline Architecture

The ARM7TDMI-S core implements a three-stage pipeline (Instruction Fetch, Decode, and Execute) that increases the speed of the flow of instructions to the processor. Several operations can take place

simultaneously, and the processing and memory systems can operate continuously. Figure 1.3 shows the ARM7TDMI-S three-stage pipeline.

Figure 1.3 ARM7TDMI-S Pipeline



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

The execution of a single ARM7TDMI-S instruction consists of the following pipeline stages:

1. Instruction Fetch (IF) – The core fetches the instruction from memory.
2. Decode (D) – The core decodes the instruction and determines which registers are needed for this operation. If necessary, the core decompresses a 16-bit Thumb instruction into a 32-bit ARM instruction.
3. Execute (X) – The core reads from the necessary register bank, executes all shift and ALU operations, and writes results to the appropriate register bank.

1.2.4 Memory Accesses

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.

1.2.5 Memory Interface

The ARM7TDMI-S memory interface allows 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 cycles:

- ◆ idle cycles
- ◆ nonsequential cycles
- ◆ sequential cycles
- ◆ coprocessor register transfer cycles

1.3 ARM7TDMI-S Core Instruction Set Summary

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

- ◆ Section 1.3.1, “ARM Instruction Summary,” page 1-9
- ◆ Section 1.3.2, “Thumb Instruction Summary,” page 1-18

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

Symbol	Description
{cond}	Refer to Table 1.11 on page 1-17
<Oprnd2>	Refer to Table 1.9 on page 1-16
{field}	Refer to Table 1.10 on page 1-17
S	Sets condition codes (optional)
B	Byte operation (optional)
H	Halfword operation (optional)
T	Forces address translation. Cannot be used with pre-indexed addresses
<a_mode2>	Refer to Table 1.3 on page 1-13
<a_mode2P>	Refer to Table 1.4 on page 1-14
<a_mode3>	Refer to Table 1.5 on page 1-15
<a_mode4L>	Refer to Table 1.6 on page 1-15
<a_mode4S>	Refer to Table 1.7 on page 1-15
<a_mode5>	Refer to Table 1.8 on page 1-16
#32bit_Imm	A 32-bit constant, formed by right-rotating an 8-bit value by an even number of bits
<reglist>	A comma-separated list of registers, enclosed in braces ({ and })

1.3.1 ARM Instruction Summary

Table 1.2 provides the ARM instruction set summary.

Table 1.2 ARM Instruction Summary

Operation		Assembler
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>	

Table 1.2 ARM Instruction Summary (Cont.)

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
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 (Cont.)

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>
	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>^	

Table 1.2 ARM Instruction Summary (Cont.)

Operation		Assembler
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>
	Increment before	STM{cond}IB Rd{!}, <reglist>{^}
	Increment after	STM{cond}IA Rd{!}, <reglist>{^}
	Decrement before	STM{cond}DB Rd{!}, <reglist>{^}
	Decrement after	STM{cond}DA Rd{!}, <reglist>{^}
	Stack operations	STM{cond}<a_mode4S> Rd{!}, <reglist>
	User registers	STM{cond}<a_mode4S> Rd{!}, <reglist>^
Swap	Word	SWP{cond} Rd, Rm, [Rn]
	Byte	SWP{cond}B Rd, Rm, [Rn]
Coprocessors	Data operations	CDP{cond} p<cpnum>, <op1>, CRd, CRn, CRm, <op2>
	Move to ARM reg from coprocessor	MRC{cond} p<cpnum>, <op1>, Rd, CRn, CRm, <op2>
	Move to coprocessor 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

Table 1.3 summarizes Addressing Mode 2.

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]

Table 1.4 summarizes Addressing Mode 2 (privileged).

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]

Table 1.5 summarizes Addressing Mode 3.

Table 1.5 Addressing Mode 3

Addressing Mode 3 - Signed Byte and Halfword Data Transfers	
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

Table 1.6 summarizes Addressing Mode 4 (load).

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)

Table 1.9 summarizes Oprnd2.

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

Table 1.10 summarizes the fields.

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)

Table 1.11 summarizes condition fields.

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

Table 1.11 Condition Fields

Condition Field {cond}	
Suffix	Description
LS	Unsigned lower or same
GE	Greater or equal
LT	Less than
GT	Greater than
LE	Less than or equal
AL	Always

1.3.2 Thumb Instruction Summary

Table 1.12 summarizes the Thumb instruction set.

Table 1.12 Thumb Instruction Summary

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

Table 1.12 Thumb Instruction Summary (Cont.)

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

Table 1.12 Thumb Instruction Summary (Cont.)

Operation		Assembler
Logical	AND	AND Rd, Rs
	EOR	EOR Rd, Rs
	OR	ORR Rd, Rs
	Bit clear	BIC Rd, Rs
	Move NOT	MVN Rd, Rs
	Test bits	TST Rd, Rs
Shift/Rotate	Logical shift left	LSL Rd, Rs, #5bit_shift_imm LSL Rd, Rs
	Logical shift right	LSR Rd, Rs, #5bit_shift_imm LSR Rd, Rs
	Arithmetic shift right	ASR Rd, Rs, #5bit_shift_imm ASR Rd, Rs
	Rotate right	ROR Rd, Rs

Table 1.12 Thumb Instruction Summary (Cont.)

Operation		Assembler
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

Table 1.12 Thumb Instruction Summary (Cont.)

Operation		Assembler
Load	With immediate offset	
	word	LDR Rd, [Rb, #7bit_offset]
	halfword	LDRH Rd, [Rb, #6bit_offset]
	byte	LDRB Rd, [Rb, #5bit_offset]
	With register offset	
	word	LDR Rd, [Rb, Ro]
	halfword	LDRH Rd, [Rb, Ro]
	signed halfword	LDRSH Rd, [Rb, Ro]
	byte	LDRB Rd, [Rb, Ro]
	signed byte	LDRSB Rd, [Rb, Ro]
	PC-relative	LDR Rd, [PC, #10bit_Offset]
	SP-relative	LDR Rd, [SP, #10bit_Offset]
	Address	
	using PC	ADD Rd, PC, #10bit_Offset
	using SP	ADD Rd, SP, #10bit_Offset
	Multiple	LDMIA Rb!, <reglist>

Table 1.12 Thumb Instruction Summary (Cont.)

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

1.4 CoreWare® Program

An LSI Logic core is a fully defined, optimized, and reusable block of logic. It supports industry-standard functions and has predefined timing and layout. A core is also an encrypted RTL simulation model for a wide range of VHDL and Verilog simulators.

The CoreWare library contains an extensive set of complex cores for the communications, consumer, and computer markets. The library consists of high-speed interconnect functions such as the GigaBlaze™ G10® Core, MIPS embedded microprocessors, MPEG-2 decoders, a PCI core, and many more.

The library also includes megafunctions or building blocks, which provide useful functions for developing a system on a chip. Through the CoreWare program, you can create a system on a chip uniquely suited to your applications.

Each core has an associated set of deliverables, including:

- ◆ RTL simulation models for both Verilog and VHDL environments
- ◆ A System Verification Environment (SVE) for RTL-based simulation
- ◆ Synthesis and timing shells
- ◆ Netlists for full timing simulation
- ◆ Complete documentation
- ◆ LSI Logic ToolKit support

LSI Logic's ToolKit provides seamless connectivity between products from leading electronic design automation (EDA) vendors and LSI Logic's manufacturing environment. Standard interfaces for formats and languages such as VHDL, Verilog, Waveform Generation Language (WGL), Physical Design Exchange Format (PDEF), and Standard Delay Format (SDF) allow a wide range of tools to interoperate within the LSI ToolKit environment. In addition to design capabilities, full scan Automatic Test Pattern Generation (ATPG) tools and LSI Logic's specialized test solutions can be combined to provide high-fault coverage test programs that assure a fully functional design.

Because your design requirements are unique, LSI Logic is flexible in working with you to develop your system-on-a-chip CoreWare design. Three different work relationships are available:

- ◆ You provide LSI Logic with a detailed specification and LSI Logic performs all design work.
- ◆ You design some functions while LSI Logic provides you with the cores and megafunctions, and LSI Logic completes the integration.
- ◆ You perform the entire design and integration, and LSI Logic provides the core and associated deliverables.

Whatever the work relationship, LSI Logic's advanced CoreWare methodology and ASIC process technologies consistently produce Right-First-Time™ silicon.

Chapter 2

Signal Descriptions

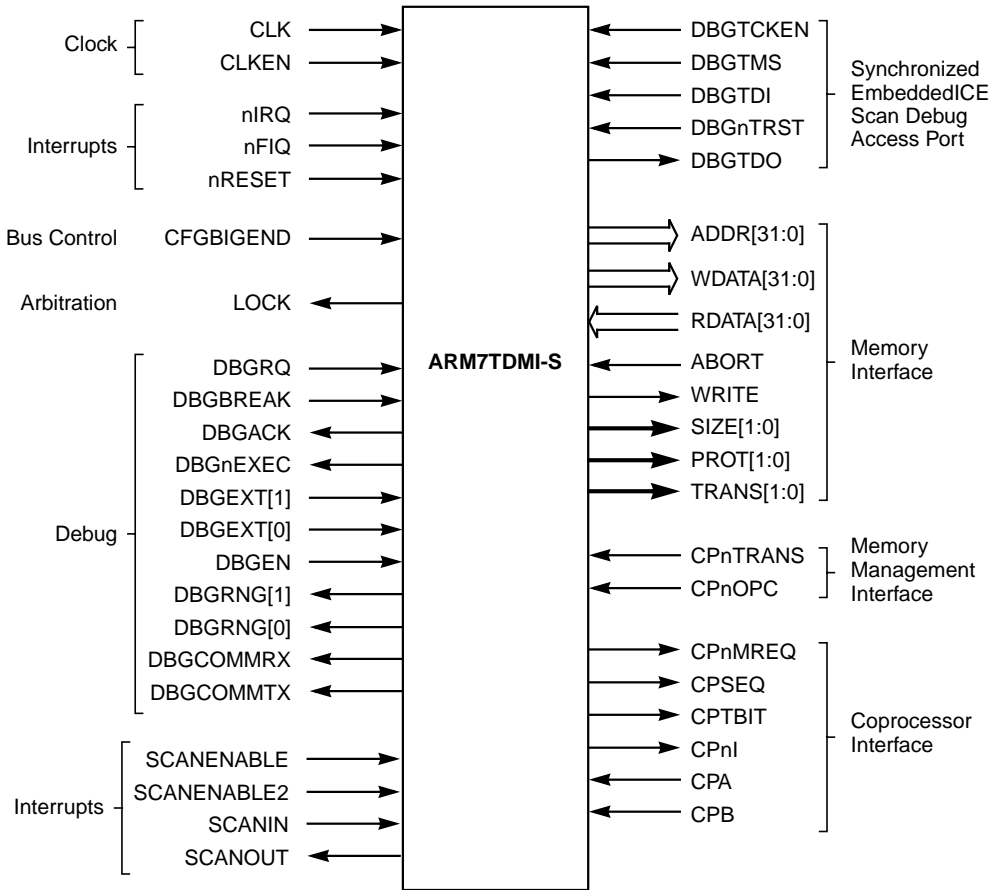
This chapter describes the signals of the ARM7TDMI-S core. The descriptions are categorized according to the interface. The signal descriptions are listed alphabetically within each interface. Active-LOW signal names contain a lowercase “n.”

This chapter contains the following sections:

- [Section 2.1, “Memory Interface”](#)
- [Section 2.2, “Coprocessor Interface”](#)
- [Section 2.3, “Clock and Control Signals”](#)
- [Section 2.4, “Debug Interface”](#)
- [Section 2.5, “ATPG Interface”](#)

[Figure 2.1](#) is the logic diagram of the ARM7TDMI-S core.

Figure 2.1 ARM7TDMI-S Logic Diagram



2.1 Memory Interface

This section describes the signals that make up the interface between the ARM7TDMI-S core and memory.

ABORT	Input Memory Abort or Bus Error I The memory system uses this input to signal the processor that a requested access is disallowed.
ADDR[31:0]	Processor Address Bus O ADDR[31:0] is the 32-bit processor address bus.
CFGBIGEND	Big Endian Configuration I When this signal is HIGH, the processor uses big-endian byte ordering. When CFGBIGEND is LOW, memory is addressed as little endian. CFGBIGEND is a static configuration signal. Once its value is registered during reset, it is not changed. CFGBIGEND is analogous to BIGEND on the hard macrocell.
CPTBIT	ARM/Thumb Instruction O When HIGH, this signal indicates to a coprocessor that the processor is executing the Thumb instruction set. When CPTBIT is LOW, the processor is executing the ARM instruction set.
LOCK	Locked Transaction Operation O 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 Type and Access **O**

These output signals to the memory system indicate whether the output is code or data, and whether access is user mode or privileged.

PROT[1:0]	Encoding
x0 ¹	Opcode fetch
x1	Data access
0x	User mode access
1x	Supervisor or privileged mode access

1. x is a don't care.

RDATA[31:0] Read Data Input Bus **I**

RDATA[31:0] is the read data bus used to transfer instructions and data between the processor and memory. The processor samples data on this bus at the end of the clock cycle during read accesses (when WRITE is LOW).

This signal is analogous to DIN[31:0] on the hard macro-cell.

SIZE[1:0] Memory Access Width **O**

These output signals indicate to the external memory system the required width of the current transfer (word, halfword, or byte).

SIZE[1:0]	Encoding
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 macro-cell.

TRANS[1:0] Next Transaction Type ○
TRANS indicates the next transaction type.

TRANS[1:0]	Description
00	Address only (internal operation cycle)
01	Coprocessor
10	Memory access at non sequential address
11	Memory access at sequential burst address

TRANS1 is analogous to inverted nMREQ, and TRANS0 signal is analogous to SEQ on the hard macrocell. TRANS[1:0] is analogous to BTRAN[1:0] on the AMBA system bus.

WDATA[31:0] Write Data Output Bus ○
WDATA[31:0] is the write data bus, used to transfer data from the processor to the memory or coprocessor system. Write data is set up prior 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 Write/Read Access ○
When HIGH, WRITE indicates a processor write cycle. When LOW, WRITE indicates a processor read cycle. This signal is analogous to nRW on the hard macrocell.

2.2 Coprocessor Interface

This section describes the signals between the ARM7TDMI-S and an external coprocessor.

CPA	Coprocessor Absent Handshake I
	The ARM7TDMI S asserts CPnI LOW to request a coprocessor operation. A coprocessor that can perform the operation drives CPA LOW, with respect to the cycle edge that precedes the coprocessor access. When CPA is driven HIGH, and the coprocessor cycle is executed (as signalled by CPnI LOW), the ARM7TDMI S aborts the coprocessor handshake and takes an undefined instruction trap. While CPA remains LOW, the ARM7TDMI S remains in a wait state until CPB goes LOW, and then completes the coprocessor instruction.
CPB	Coprocessor Busy Handshake I
	When a coprocessor can perform a requested ARM7TDMI-S operation (CPnI is asserted) but not immediately, it drives CPB HIGH. When the coprocessor is ready to start, it takes CPB LOW, before the start of the coprocessor instruction execution cycle.
CPnI	Not Coprocessor Instruction O
	When the ARM7TDMI S core executes a coprocessor instruction, it drives this output LOW and waits for a response from the coprocessor. The action taken depends on the coprocessor's response on the CPA and CPB inputs.
CPnMREQ	Not Memory Request O
	When LOW, this signal indicates that the processor requires a memory access during the next transaction. This signal is analogous to nMREQ on the hard macrocell.
CPnOPC	Not Opcode Fetch O
	When LOW, this signal indicates that the processor is fetching an instruction from memory. When CPnOPC is 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	Sequential Address	O
	This output goes 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, four greater in ARM state, or two greater when fetching opcodes in Thumb state.	
	This signal is analogous to SEQ on the hard macrocell.	
CPnTRANS	Not Memory Translate	O
	When LOW, this signal indicates that the processor is in user mode. It can be used to indicate 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.	

2.3 Clock and Control Signals

This section describes the clock, interrupt, and reset signals.

CLK	Clock Input	I
	All ARM7TDMI S memory accesses and internal operations are clocked with respect to CLK. All outputs change from the rising edge of CLK, and all inputs are sampled on the rising edge of CLK.	
	The CLKEN input can be used with a free running CLK to add synchronous wait states. Alternatively, the clock can be stretched indefinitely in either phase to allow access to slow peripherals or memory, or to put the system into a low power state.	
	CLK is also used for serial scan chain debug operation with the EmbeddedICE tool chain.	
	This signal is analogous to inverted MCLK on the hard macrocell.	
CLKEN	Wait State Control	I
	When accessing slow peripherals, driving CLKEN LOW forces the ARM7TDMI S to wait for an integer number of CLK cycles. When the CLKEN control is not used, it must be tied HIGH.	
	This signal is analogous to nWAIT on the hard macrocell.	

nFIQ	<p>Fast Interrupt Request</p> <p>This active-LOW signal is a high priority synchronous interrupt request to the processor. If the appropriate enable in the processor is active when this signal is driven LOW, the processor is interrupted.</p> <p>nFIQ is level sensitive and must be held LOW until a suitable interrupt acknowledge response is received from the processor.</p> <p>This signal is analogous to nFIQ on the hard macrocell when ISYNC is HIGH.</p>	I
nIRQ	<p>Interrupt Request</p> <p>This active-LOW signal is a low priority synchronous interrupt request to the processor. If the appropriate enable in the processor is active when this signal is taken LOW, the processor is interrupted.</p> <p>nIRQ is level sensitive and must be held LOW until a suitable interrupt acknowledge response is received from the processor.</p> <p>This signal is analogous to nIRQ on the hard macrocell when ISYNC is HIGH.</p>	I
nRESET	<p>Reset</p> <p>Assertion of this input signal forces the processor to terminate the current instruction, and subsequently enter the reset vector in supervisor mode. nRESET 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.</p>	I

2.4 Debug Interface

This section describes the ARM7TDMI-S debug signals. The processor is in debug state when DBGEN is HIGH.

- DBGACK** **Debug Acknowledge** **O**
When HIGH, this signal indicates the ARM7TDMI S is in debug state. It is enabled only when DBGEN is HIGH.
- DBGBREAK** **EmbeddedICE Breakpoint/Watchpoint Indicator** **I**
This signal allows external hardware to halt 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 Spipeline. When the memory access is for data, the ARM7TDMI S enters debug state after the current instruction completes execution, which 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** **EmbeddedICE Communications Channel Receive** **O**
When HIGH, this signal indicates that the comms channel receive buffer is full. DBGCOMMRX is enabled only when DBGEN is HIGH.
This signal is analogous to COMMRX on the hard macrocell.
- DBGCOMMTX** **EmbeddedICE Communications Channel Transmit** **O**
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** **Debug Enable** **I**
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 | Not Executed | O |
| | When HIGH, this signal indicates that the instruction in the execution unit is not being executed. For example, it has failed its condition code check. | |
| DBGEXT[1:0] | EmbeddedICE External Inputs 0 and 1 | I |
| | These inputs to the EmbeddedICE macrocell logic in the ARM7TDMI S allow breakpoints and/or watchpoints to depend 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] | EmbeddedICE Rangeout | O |
| | DBGRNG[1:0] indicate that EmbeddedICE Watchpoint register 0/1 has matched the conditions currently present on the address, data, and control buses. These outputs are independent of the state of the watchpoint enable control bit. DBGRNG[1:0] are enabled only when DBGEN is HIGH. | |
| | These outputs are analogous to RANGE[1:0] on the hard macrocell. | |
| DBGRQ | Debug Request | I |
| | Assertion HIGH of this internally synchronized input signal indicates a request to the processor to enter debug state. DBGRQ is enabled only when DBGEN is HIGH. | |
| DBGTCKEN | Test Clock Enable | I |
| | DBGTCKEN is enabled only when DBGEN is HIGH. | |
| DBGTDI | EmbeddedICE Data In | I |
| | DBGTDI is the JTAG test data input. DBGTDI is enabled only when DBGEN is HIGH. | |
| DBGTDO | EmbeddedICE Data Out | O |
| | DBGTDO is the output from the boundary scan logic. DBGTDO is enabled only when DBGEN is HIGH. | |
| DBGnTDOEN | Not DBGTDO Enable | O |
| | When LOW, this signal denotes that serial data is being driven out on the DBGTDO output. DBGnTDOEN is nor- | |

mally used as an output enable for a DBGTDO pin in a packaged part.

DBGTMS	EmbeddedICE Mode Select	I
	DBGTMS is the JTAG test mode select. DBGTMS is enabled only when DBGEN is HIGH.	
DBGnTRST	Not Test Reset	I
	DBGnTRST is the internally synchronized active LOW reset signal for the EmbeddedICE macrocell internal state.	

2.5 ATPG Interface

The signals that make up the ATPG interface are used for automatic test pattern generation.

SCANENABLE	Scan Test Path Enable	I
	This input is LOW for normal system configuration and HIGH during scan testing.	
SCANENABLE2	Scan Test Path Enable 2	I
	This input is LOW for normal system configuration and HIGH during scan testing.	
SCANIN	Scan Test Path Serial Input	I
	Serial input is fed into SCANIN when SCANENABLE and SCANENABLE2 are HIGH.	
SCANOUT	Scan Test Path Serial Output	O
	This output is active when SCANENABLE and SCANENABLE2 are HIGH.	

Chapter 3

Programmer's Model

This chapter describes the two operating states, registers, and exceptions of the ARM7TDMI-S core. It contains the following sections:

- [Section 3.1, "Processor Operating States"](#)
- [Section 3.2, "Memory Formats"](#)
- [Section 3.3, "Data Types"](#)
- [Section 3.4, "Operating Modes"](#)
- [Section 3.5, "Registers"](#)
- [Section 3.6, "Exceptions"](#)
- [Section 3.7, "Interrupt Latencies"](#)
- [Section 3.8, "Reset"](#)

3.1 Processor Operating States

From the programmer's point of view, the ARM7TDMI-S core can be in one of two states:

- *ARM state* executes 32-bit, word-aligned ARM instructions.
- *Thumb state* operates with 16-bit, halfword-aligned Thumb instructions. In this state, the PC uses bit 1 to select between alternate halfwords.

Note: Transition between these two states does not affect the processor mode or the contents of the registers.

The operating state of the ARM7TDMI-S core can be switched between ARM state and Thumb state using the `BX` instruction. Refer to the *ARM Architecture Manual* for more information.

All exception handling is done 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.

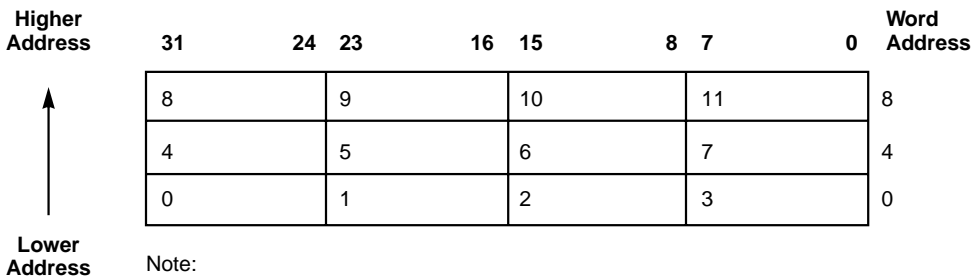
3.2 Memory Formats

The ARM7TDMI-S core views memory as a linear collection of bytes numbered upwards from zero. Bytes 0 to 3 hold the first stored word, bytes 4 to 7 the second, and so on. The ARM7TDMI-S core can treat words in memory as being stored either in *big-endian* or *little-endian* format.

3.2.1 Big-Endian Format

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

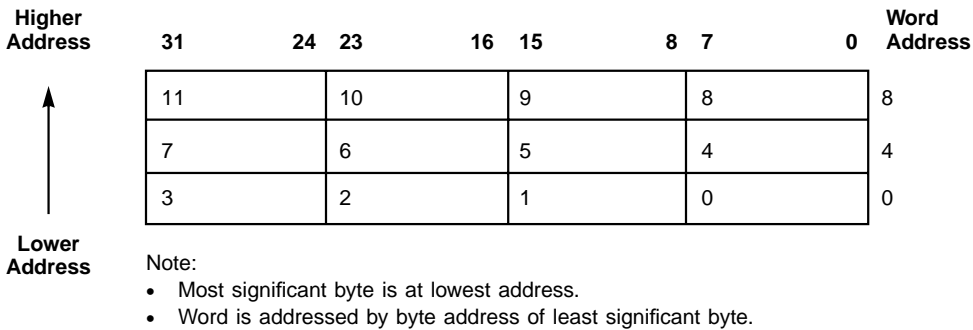
Figure 3.1 Big-Endian Addresses of Bytes Within Words



3.2.2 Little-Endian Format

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

Figure 3.2 Little-Endian Addresses of Bytes Within Words



3.3 Data Types

The ARM7TDMI-S supports byte (8 bit), halfword (16 bit), and word (32 bit) data types.

You must align these data types as follows:

- Words must be aligned to four-byte boundaries
- Halfwords must be aligned to two-byte boundaries
- Bytes can be placed on any byte boundary

3.4 Operating Modes

The ARM7TDMI-S supports seven modes of operation:

- User (usr): The normal ARM program execution state
- FIQ (fiq): Supports a data transfer or channel process
- IRQ (irq): Used for general-purpose interrupt handling
- Supervisor (svc): Protected mode for the operating system
- Abort mode (abt): Entered after a data or instruction prefetch abort
- System (sys): A privileged user mode for the operating system
- Undefined (und): Entered when an undefined instruction is executed

Mode changes are either made under software control or brought about by external interrupts or exception processing. Most application programs execute in User mode. The nonuser modes—known as *privileged modes*—are entered in order to service interrupts or exceptions, or to access protected resources.

3.5 Registers

The ARM7TDMI-S has a total of 37 registers (31 general-purpose 32-bit registers and six 32-bit status registers). These registers are not all accessible at the same time. The processor state and operating mode dictate which registers are available to the programmer.
















3.5.1 The ARM State Register Set

In ARM state, 16 general registers and one or two status registers are visible at any one time. In privileged (nonuser) modes, mode-specific banked registers are visible. [Figure 3.3](#) shows which registers are available in each mode: the banked registers are marked with a shaded triangle.






The ARM-state register set contains 16 directly accessible registers: R0 to R15. An additional register, the Current Program Status Register (CPSR), contains condition code flags and the current mode bits. Registers R0 through R13 are general-purpose registers that hold either data or address values. Registers R14, R15, and the CPSR have special functions as described in the subsections below. Refer to [Section 3.5.5, “Program Status Registers,”](#) for more information on the Program Status Registers.

Figure 3.3 Register Organization in ARM State

ARM State General Registers and Program Counter

System & 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

3.5.1.1 Link Register (R14)

R14 is used as the subroutine link register. It receives a copy of R15 when a Branch with Link (BL) instruction is executed. At all other times, R14 is 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.

3.5.1.2 Program Counter (R15)

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

3.5.1.3 Saved Program Status Register (SPSR)

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

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. User, IRQ, Supervisor, Abort and Undefined each have two banked registers mapped to R13 and R14, allowing each of these modes to have a private stack pointer and link registers.

3.5.2 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 Program Counter (PC),
- a stack pointer register (SP),
- a link register (LR),
- and the Current Program Status register (CPSR).

There are banked Stack Pointers, Link registers and Saved Process Status registers (SPSRs) for each privileged mode, as shown in [Figure 3.4](#).

Figure 3.4 Register Organization in Thumb State

Thumb State General Registers and Program Counter

System & 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

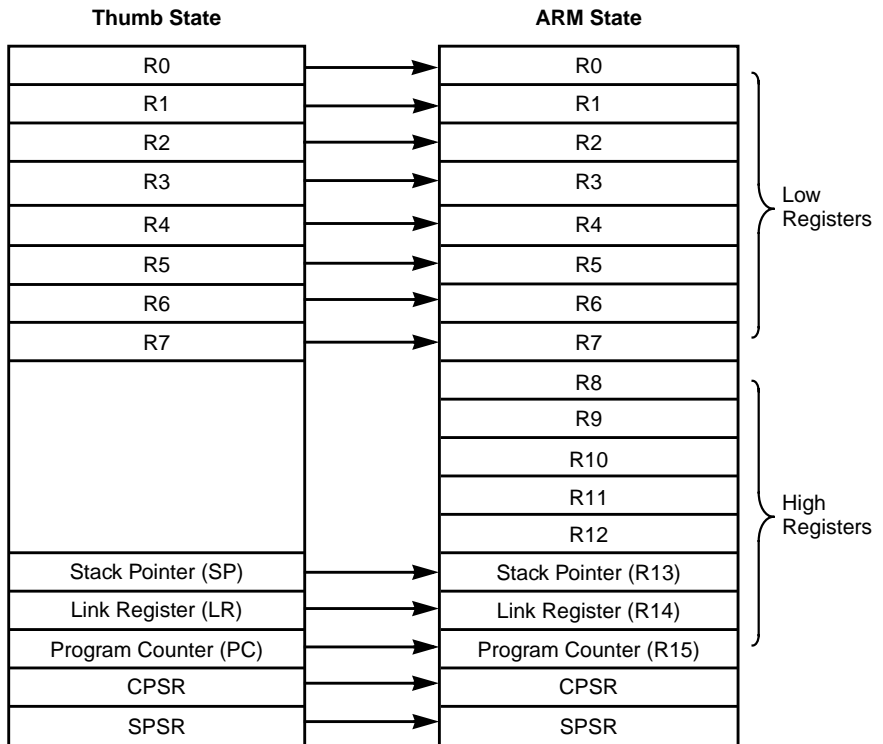
3.5.3 The Relationship Between ARM and Thumb State Registers

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 into ARM state R13.
- Thumb state LR maps into ARM state R14.
- The Thumb state Program Counter maps onto the ARM state Program Counter (R15).

These relationships are shown in [Figure 3.5](#).

Figure 3.5 Thumb State to ARM State Register Mappings



Note that R0 – R7 are referred to as the *Low* registers, and R8 – R15 are referred to as the *High* registers.

3.5.4 Accessing High Registers in Thumb State

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

You can use special variants of the MOV instruction to transfer a value from a Low register (R0 – R7) to a High register, and from a High register to a Low register. High register values can also be compared against or added to Low register values with the CMP and ADD instructions.

Refer to the *ARM Architecture Reference Manual* for more information.

3.5.5 Program Status Registers

The ARM7TDMI-S core contains a Current Program Status register (CPSR) and five Saved Program Status registers (SPSRs) for exception handlers to use. These registers:

- Hold information about the most recently performed ALU operation
- Control the enabling and disabling of interrupts
- Set the processor operating mode

The arrangement of bits in the Program Status Registers is shown in [Figure 3.6](#).

Note: Use a read-modify-write strategy when changing the CPSR to maintain compatibility with future processors.

Figure 3.6 Program Status Register Format



Condition Code Flags	[31:28]
N: Negative/Less Than	31
Z: Zero	30
C: Carry/Borrow/Extend	29
V: Overflow	28

The N, Z, C and V bits are the condition code flags. These can be changed as a result of arithmetic and logical operations. These flags also can be set using MSR and LDM instructions. The ARM7TDMI-S tests these flags to determine whether to execute an instruction or not.

In ARM state, all instructions can be executed conditionally. In Thumb state, only the Branch instruction is capable of conditional execution.

Refer to the ARM Architecture Reference Manual for more information on conditional execution.

Reserved	Reserved Bits	[27:8]
	Bits [27:8] in the Program Status registers are reserved. When changing a PSR's flag or control bits, you must ensure that these unused bits are not altered. Do not rely	

on them containing specific values, because in future processors they might read as ones or zeros.

Control **Control Bits** **[7:0]**

The bottom eight bits of a PSR are known collectively as the control bits. These bits change when an exception arises. If the processor is operating in a privileged mode, software can manipulate them.

T: Operating State **7**

This bit reflects the processor operating state. When this bit is set, the processor is executing in Thumb state; otherwise it is executing in ARM state. This condition is reflected on the CPTBIT external signal.

Note: Never use an MSR instruction to force a change to the state of the T bit; otherwise the processor will enter an unpredictable state.

I and F: Interrupt Disable Bits **[6:5]**

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

M[4:0]: The Mode Bits **[4:0]**

M4, M3, M2, M1, and M0 are the mode bits. These bits determine the processor's operating mode, as shown in [Table 3.1](#). Not all combinations of the mode bits define a valid processor mode. Only use those combinations explicitly described below. If any illegal value is programmed into the mode bits, M[4:0], then the processor will enter an unrecoverable state, requiring a reset. [Table 3.1](#) lists the mode bit states and the accessible state registers for each mode.

Table 3.1 Mode Bit States

M[4:0]	Mode	Accessible Thumb State Registers	Accessible ARM State Registers
0b10000	User	R7..R0, LR, SP, PC, CPSR	R14..R0, PC, CPSR
0b10001	FIQ	R7..R0, LR_fiq, SP_fiq, PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
0b10010	IRQ	R7..R0, LR_irq, SP_irq, PC, CPSR, SPSR_irq	R12..R0, R14_irq..R13_irq, PC, CPSR, SPSR_irq
0b10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc..R13_svc, PC, CPSR, SPSR_svc
0b10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt..R13_abt, PC, CPSR, SPSR_abt
0b11011	Undefined	R7..R0, LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und..R13_und, PC, CPSR
0b11111	System	R7..R0, LR, SP, PC, CPSR	R14..R0, PC, CPSR

3.6 Exceptions

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

If multiple exceptions arise at the same time, they are dealt with in a fixed order shown in [Section 3.6.9, “Exception Priorities.”](#)

[Table 3.2](#) summarizes the PC value preserved in the relevant R14 register on exception entry, and the recommended instruction for exiting

the exception handler. More details on actions taken on entering and exiting an exception are provided in the following subsections.

Table 3.2 Exception Entry/Exit

	Return Instruction	Previous State		Notes
		ARM R14_x	Thumb R14_x	
BL	MOV PC, R14	PC + 4	PC + 2	1
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	1
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	1
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4	2
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4	2
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	1
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8	3
RESET	NA	–	–	4

1. Where PC is the address of the BL/SWI/Undefined Instruction fetch that had the prefetch abort.
2. Where PC is the address of the instruction that did not get executed because the FIQ or IRQ took priority.
3. Where PC is the address of the Load or Store instruction that generated the data abort.
4. The value saved in R14_svc upon reset is unpredictable.

3.6.1 Entering an Exception

When handling an exception, the ARM7TDMI-S core:

1. Preserves the address of the next instruction in the appropriate Link register.

If the exception has been entered from ARM state, then the address of the next instruction is copied into the Link register (current PC + 4 or PC + 8, depending on the exception). If the exception has been entered from Thumb state, then the value the ARM7TDMI-S core writes into the Link register is the current PC offset by a value that causes the program to resume from the correct place on return from the exception. Thus the exception handler need not determine the state when entering an exception. For example, in the case of a Software Interrupt (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 can also set the interrupt disable flags to prevent otherwise unmanageable nestings of exceptions.

Note: Exceptions are always handled in ARM state. If the processor is in Thumb state when an exception occurs, it automatically switches into ARM state when the PC is loaded with the exception vector address.

3.6.2 Leaving an Exception

When handling of an exception is completed, the exception handler:

1. Moves the Link Register minus an offset where appropriate to the PC. (The offset varies depending on the type of exception.)
2. Copies the SPSR back to the CPSR.
3. Clears the interrupt disable flags, if they were set on entry.

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

3.6.3 Fast Interrupt Request (FIQ)

The FIQ (Fast Interrupt Request) 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).

Driving `nFIQ` LOW externally generates an FIQ exception.

Irrespective of whether the exception was entered from ARM or Thumb state, an FIQ handler executes the following instruction to leave the interrupt:

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

To disable an FIQ exception from a privileged mode, set the CPSR's 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.

3.6.4 Interrupt Request (IRQ)

The IRQ (Interrupt Request) exception is a normal interrupt caused by a LOW level on the nIRQ input. An IRQ exception has a lower priority than an FIQ exception. It is masked out when an FIQ sequence is entered. To disable an IRQ exception at any time, set the I bit in the CPSR from a privileged (non-user) mode.

Irrespective of whether the exception was entered from ARM or Thumb state, an IRQ handler executes the following instruction to return from the interrupt:

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

3.6.5 Aborts (PABT and DABT)

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

There are two types of aborts:

- a prefetch abort (PABT) occurs during an instruction prefetch
- a data abort (DABT) occurs during a data access

3.6.5.1 Prefetch Aborts

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 a branch occurs while it is in the pipeline—the abort does not take place.

3.6.5.2 Data Abort

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

1. Single data transfer instructions (*LDR*, *STR*) write back modified base registers: the Abort handler must be aware of this.
2. The swap instruction (*SWP*) is aborted as though it had not been executed. The abort must occur on the read access of the *SWP* instruction.
3. 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 (it has the base in the transfer list), the ARM7TDMI-S prevents the overwriting. All register overwriting is prevented 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 state (ARM or Thumb) at the point of entry:

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

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

3.6.6 Software Interrupt (SWI)

The software interrupt instruction (*SWI*) is used to enter Supervisor mode, usually to request a particular supervisor function. An SWI handler returns by executing the following instruction, irrespective of the state (ARM or Thumb):

```
MOV PC, R14_svc
```

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

3.6.7 Undefined Instruction (UDEF)

When the ARM7TDMI-S encounters an instruction that neither it nor any system coprocessor 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.

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

```
MOVS PC,R14_und
```

This action restores the CPSR and returns to the instruction following the undefined instruction.

Refer to the *ARM Architecture Manual* for more on undefined instructions.

3.6.8 Exception Vectors

Table 3.3 lists the exception vector addresses. In the table, I and F represent the previous value.

Table 3.3 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

3.6.9 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 software interrupt (lowest priority)

Some exceptions cannot occur simultaneously:

- Undefined instructions and software interrupts are mutually exclusive, because they each correspond to particular (non-overlapping) decodings of the current instruction.
- If a data abort occurs at the same time as a FIQ, and FIQs are enabled (the CPSR's F flag is cleared), the ARM7TDMI-S enters the

data abort handler and then immediately proceeds to the FIQ vector. A normal return from the FIQ causes the data abort handler to resume execution. Data aborts must have higher priorities than FIQs to ensure that the transfer error does not escape detection. Add the time for this exception entry to worst-case FIQ latency calculations in a system that uses aborts to support virtual memory.

3.7 Interrupt Latencies

This section calculates the minimum and maximum interrupt latencies for the ARM7TDMI-S.

3.7.1 Maximum Interrupt Latencies

When FIQs are enabled, the worst-case latency for FIQ is the 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 $T_{\text{syncmax}} + T_{\text{ldm}} + T_{\text{exc}} + T_{\text{fiq}} = 27$ processor cycles, just under 0.7 microseconds in a system that uses a continuous 40 MHz processor clock. At the end of this time, the ARM7TDMI-S executes the instruction at 0x1C.

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

3.7.2 Minimum Interrupt Latencies

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

3.8 Reset

When the nRESET signal goes LOW, the ARM7TDMI-S abandons the executing instruction. When nRESET goes HIGH again, the ARM7TDMI-S:

1. Forces M[4:0] to 0b10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR's T bit.
2. Forces the PC to fetch the next instruction from address 0x00.
3. Reverts to the ARM state and execution resumes.

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

Chapter 4

Memory Interface

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

- [Section 4.1, “About the Memory Interface”](#)
 - [Section 4.2, “Bus Cycle Types”](#)
 - [Section 4.3, “Bus Interface Signals”](#)
 - [Section 4.4, “Use of CLKEN to Control Bus Cycles”](#)
-

4.1 About the Memory Interface

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

The ARM7TDMI-S core supports four basic types of memory cycles:

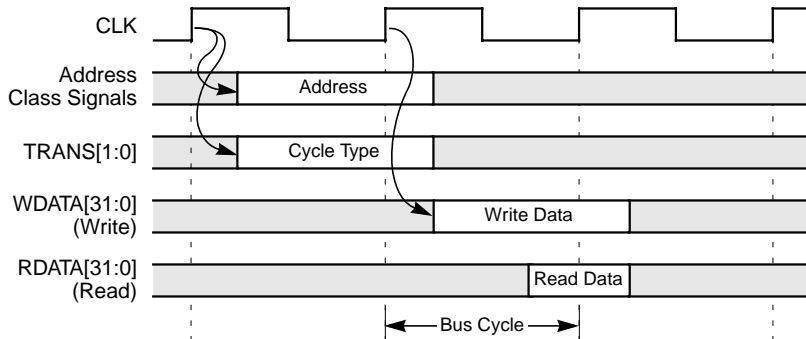
- nonsequential
 - sequential
 - internal
 - coprocessor register transfer
-

4.2 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 procedure gives the maximum time for a memory cycle to decode the address, and respond to the access request.

Figure 4.1 shows a single memory cycle.

Figure 4.1 Simple Memory Cycle



The ARM7TDMI-S bus interface can perform four different types of memory cycles as indicated by the state of the TRANS[1:0] signals. Table 4.1 shows the encoding of the TRANS[1:0] signals, which indicates the memory cycle types. An ARM7TDMI-S memory controller must commit only to a memory access on an N cycle or an S cycle.

Table 4.1 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

The ARM7TDMI-S core has four basic types of memory cycles:

- a nonsequential cycle (N), during which the ARM7TDMI-S core requests a transfer to or from an address that is unrelated to the address used in the preceding cycle
- a sequential cycle (S), during which the ARM7TDMI-S core requests a transfer to or from an address that is either one word or one halfword greater than the address used in the preceding cycle

- an internal cycle (I), 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 (C), 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.

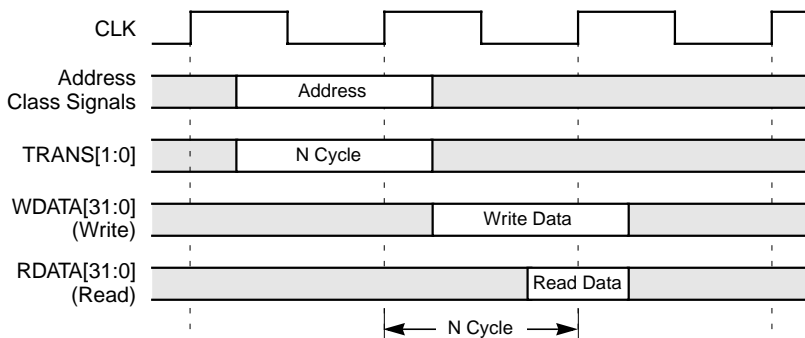
these cycles are described in more detail in the following subsections.

4.2.1 Nonsequential Cycles (N)

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

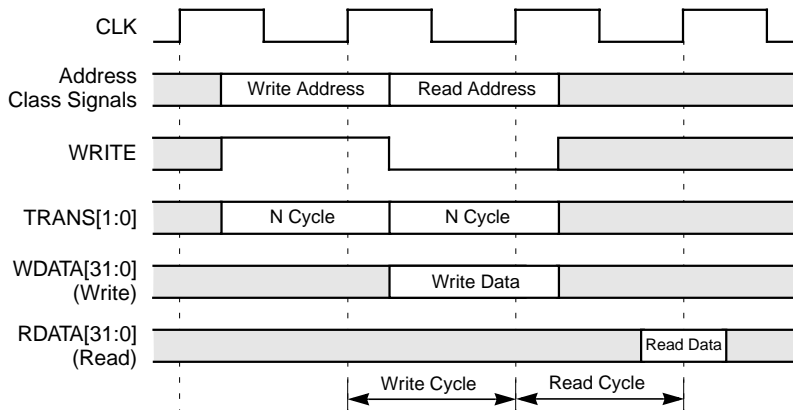
The address class signals (ADDR, WRITE, SIZE, PROT, LOCK) and TRANS[1:0] = N cycle are driven on the bus. At the end of the next bus cycle, the data is transferred between the CPU and the memory, as illustrated in [Figure 4.2](#).

Figure 4.2 Nonsequential Memory Cycle



The ARM7TDMI-S core can perform back-to-back, nonsequential memory cycles. These cycles happen, for example, when an STR instruction is executed, as shown in [Figure 4.2](#). If you are designing a memory controller for the ARM7TDMI-S core, and your memory system cannot cope with this case, use the CLKEN signal to extend the bus cycle to allow sufficient cycles for the memory system to respond. See [Section 4.4, "Use of CLKEN to Control Bus Cycles,"](#) for more information.

Figure 4.3 Back-to-Back Memory Cycles



4.2.2 Sequential Cycles (S)

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

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

- for a burst of word accesses, the address is incremented by four bytes
- for a burst of halfword access, the address is incremented by two bytes

Byte access bursts are not possible.

A burst always starts with an N cycle or a merged I-S cycle (see [Section 4.2.4, "Merged I-S Cycles"](#)), and continues with S cycles. A burst is made up of transfers of the same type. The ADDR[31:0] signal increments during the burst. A burst does not affect the other address class signals.

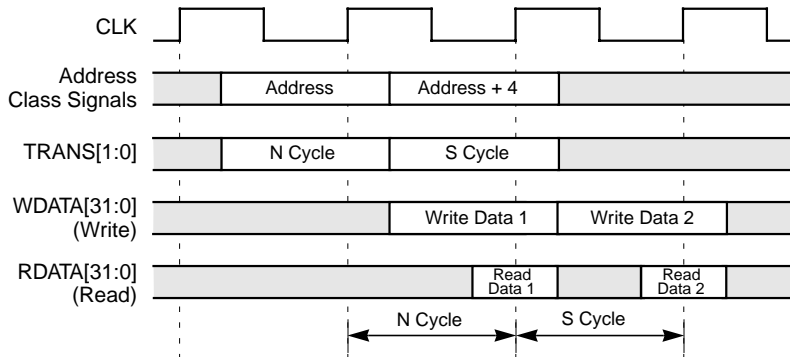
The types of bursts are shown in [Table 4.2](#).

Table 4.2 Burst Types

Burst Type	Address Increment	Cause
Word read	4 bytes	ARM7TDMI Scode 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 [Section 4.3, “Bus Interface Signals.”](#) [Figure 4.4](#) shows an example of a burst access.

Figure 4.4 Sequential Access Cycles



4.2.3 Internal Cycles

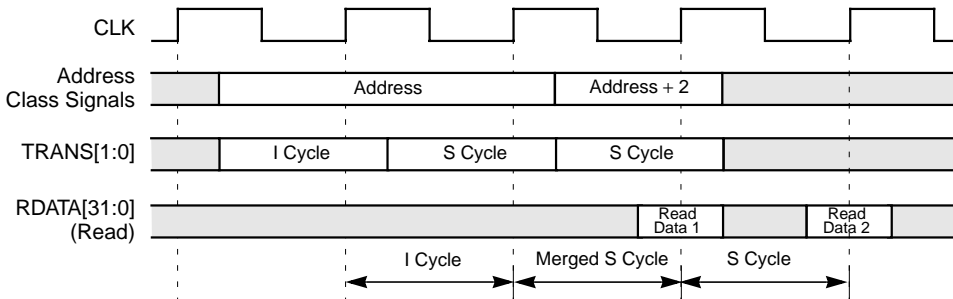
During an internal cycle, the ARM7TDMI-S core does not require a memory access, because an internal function is being performed, and no useful prefetching can be performed at the same time.

Where possible the ARM7TDMI-S core broadcasts the address for the next access, so that decoding can start, but the memory controller must not commit to a memory access. This operation is further described in [Section 4.2.4, “Merged I-S Cycles.”](#)

4.2.4 Merged I-S Cycles

Where possible, the ARM7TDMI-S core performs an optimization on the bus to allow extra time for memory decode. When optimization occurs, the address of the next memory cycle is driven during an internal cycle on this bus, which 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, which commits to the access, and the memory controller must initiate the memory access. [Figure 4.5](#) shows this case.

Figure 4.5 Merged I-S Cycles



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 might 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.

4.2.5 Coprocessor Register Transfer Cycles

During a coprocessor register transfer cycle, the ARM7TDMI-S core 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 5, “Coprocessor Interface.”](#)

4.3 Bus Interface Signals

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

- clocking and clock control signals
 - CLK
 - CLKEN
 - nRESET
- address class signals
 - ADDR[31:0]
 - WRITE
 - SIZE[1:0]
 - PROT[1:0]
 - LOCK
- memory request signals (TRANS[1:0])
- data timed signals
 - 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 described in [Section 4.4, “Use of CLKEN to Control Bus Cycles.”](#) The other sections in this chapter describe a simple system in which CLKEN is permanently HIGH.

4.3.1 Addressing Signals

The address class signals are described in more detail below.

4.3.1.1 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 4 GBytes of linear addressing space. When a word access is signalled, the memory system ignores the bottom two bits, ADDR[1:0]; when a halfword access is signalled, the memory system ignores the bottom bit, ADDR0.

4.3.1.2 WRITE

WRITE specifies the direction of the transfer. A HIGH on WRITE indicates an ARM7TDMI-S write cycle; a LOW indicates an ARM7TDMI-S read cycle. 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.

4.3.1.3 SIZE[1:0]

The SIZE[1:0] bus encodes the size of the transfer. The ARM7TDMI-S core can transfer word, halfword, and byte quantities. [Table 4.3](#) shows the encoding of SIZE[1:0].

Table 4.3 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 core 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 individ-

ual byte write capability is not provided, use of these capabilities might not be possible.

4.3.1.4 PROT[1:0]

The PROT[1:0] bus encodes information about the transfer. A memory management unit (MMU) uses this signal to determine whether an access is from a privileged mode, and whether it is an opcode or a data fetch. Therefore these signals can be used to implement an access permission scheme. [Table 4.4](#) shows the encoding of PROT[1:0].

Table 4.4 PROT Encoding

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

4.3.1.5 LOCK

Assertion of LOCK indicates to an arbiter that an atomic operation is being performed on the bus. LOCK 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.

4.3.1.6 CPTBIT

CPTBIT indicates whether the ARM7TDMI-S core is in ARM state (CPTBIT = LOW) or Thumb state (CPTBIT = HIGH).

4.3.2 Data Timed Signals

The data timed signals are described below.

4.3.2.1 WDATA[31:0]

WDATA[31:0] is the write data bus. All data written out from the ARM7TDMI-S core is driven on this bus. Data transfers from the ARM7TDMI-S core to a coprocessor also use this bus during C cycles.

In normal circumstances, a memory system must sample the WDATA[31:0] bus on the rising edge of CLK at the end of a write bus cycle. The value on WDATA[31:0] is valid only during write cycles.

4.3.2.2 RDATA[31:0]

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

4.3.2.3 ABORT

Assertion of 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 core to take the data abort trap. If ABORT 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.

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

For more details about aborts, see [Section 3.6.5, “Aborts \(PABT and DABT\),” page 3-14](#).

4.3.3 Byte and Halfword Accesses

The ARM7TDMI-S core indicates the size of a transfer using the SIZE[1:0] signals. These signals are encoded in [Table 4.3](#).

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 ARM7TDMI-S core always produces a byte address. However, the memory system should ignore the bottom bits of the address. The significant address bits are listed in [Table 4.5](#).

Table 4.5 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 core 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 [Section 3.2, “Memory Formats,” page 3-2](#) for more information.

[Table 4.6](#) shows the fields extracted by the ARM7TDMI-S core.

Table 4.6 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 and 16-bit memory systems to the ARM7TDMI-S core, make sure that the data is presented to the correct byte lanes on the ARM7TDMI-S core as shown in [Table 4.7](#) and [Table 4.8](#) below.

Table 4.7 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]

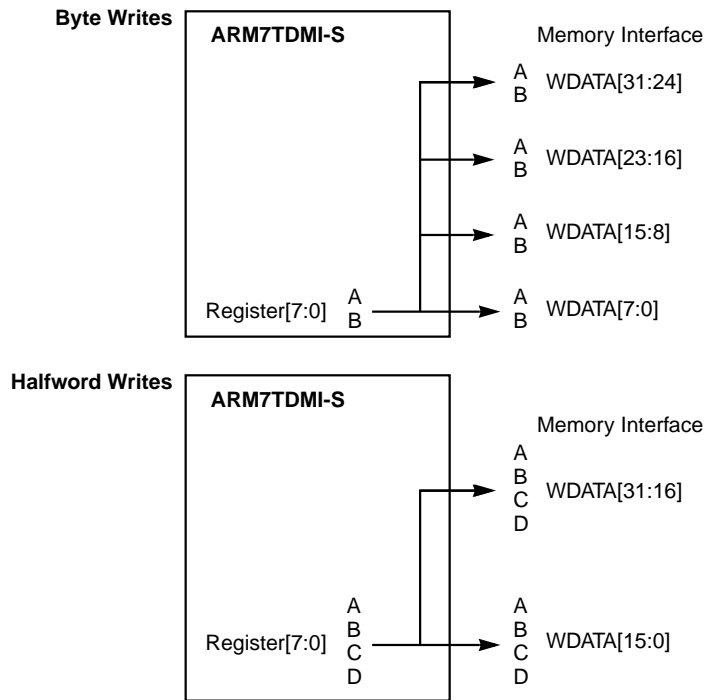
Table 4.8 Byte Accesses

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]

4.3.4 Write Operations

When the ARM7TDMI-S core performs a byte or halfword write, the data being written is replicated across the bus, as illustrated in [Figure 4.6](#). The memory system can use the most convenient copy of the data. A writable memory system must be able to perform a write to any single byte in the memory system. The ARM C Compiler and the Debug tool chain require this capability.

Figure 4.6 Data Replication



4.4 Use of CLKEN to Control Bus Cycles

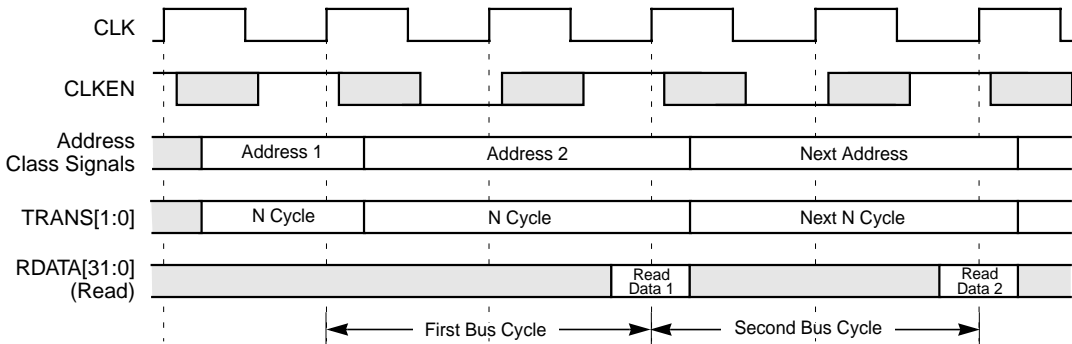
The pipelined nature of the ARM7TDMI-S bus interface means that there is a distinction between clock cycles and bus cycles. CLKEN can be used to stretch a bus cycle, so that the cycle lasts for multiple clock cycles. The CLKEN input extends the timing of bus cycles in increments of complete CLK cycles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

- 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 time might be more than one CLK cycle as illustrated in [Figure 4.7](#), 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 4.7 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 sampling ensures that the state of the memory controller is not accidentally updated during a bus cycle.

Important

Note: *Because of the clock gating structure in the ARM7TDMI-S core, both DBGTCKEN and CLKEN must be held LOW in order for the entire processor core's clock to stop. Otherwise you will not get the power benefit.*

Chapter 5

Coprocessor Interface

This chapter describes the ARM7TDMI-S coprocessor interface:

- [Section 5.1, “About Coprocessors”](#)
 - [Section 5.2, “Coprocessor Availability”](#)
 - [Section 5.3, “Coprocessor Interface Signals”](#)
 - [Section 5.4, “Pipeline Following Signals”](#)
 - [Section 5.5, “Coprocessor Interface Handshaking”](#)
 - [Section 5.6, “Connecting Coprocessors”](#)
 - [Section 5.7, “Implementations Without External Coprocessors”](#)
 - [Section 5.8, “Undefined Instructions”](#)
 - [Section 5.9, “Privileged Instructions”](#)
-

5.1 About Coprocessors

The ARM7TDMI-S instruction set allows additional specialized 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 connects to the same data bus as the ARM7TDMI-S processor in the system, and tracks the pipeline in the ARM7TDMI-S

processor. Thus 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 core:

1. Evaluates the condition codes to determine whether the coprocessor will execute the instruction, and signals this case to any coprocessors in the system (using CPnI).
2. Generates any addresses that the instruction requires, 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 on the CPA and CPB signals whether it can accept the instruction.
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.

5.2 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 core 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. ARM has also reserved other coprocessor numbers.

[Table 5.1](#) lists the coprocessor availability.

Table 5.1 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 to info@arm.com with “coprocessor” in the subject line for up-to-date information on which coprocessor numbers have been allocated.

5.3 Coprocessor Interface Signals

The signals that interface the ARM7TDMI-S core to a coprocessor are grouped into four categories as shown in [Table 5.2](#).

Table 5.2 Coprocessor Interface Signals

Category	Signals
Clock and Control	CLK, CLKEN, nRESET
Pipeline Following	CPnMREQ, CPSEQ, CPnTRANS, CPnOPC, CPTBIT
Handshake	CPnI, CPA, CPB
Data	WDATA[31:0], RDATA[31:0]

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

5.4 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 CLK and CLKEN, and to the ARM7TDMI-S input data bus, RDATA[31:0], over which instructions are fetched.

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

- At reset (nRESET LOW), the pipeline must either be marked as invalid, or filled with instructions that will not decode 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.
- Advance the pipeline 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 CPnI that they have entered the execute stage, 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. 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.

5.5 Coprocessor Interface Handshaking

This section describes the operation of the coprocessor and ARM7TDMI-S core during execution of a coprocessor operation, and describes the signals that are affected.

The ARM7TDMI-S core and any coprocessors in the system use the signals in [Table 5.3](#) for handshaking. These signals are explained in more detail in [Section 5.5.3, “Coprocessor Signalling.”](#)

Table 5.3 Handshaking Signals

Signal	Direction	Meaning
CPnI	ARM7TDMI-S to coprocessor	Not coprocessor instruction
CPA	Coprocessor to ARM7TDMI-S	Coprocessor absent
CPB	Coprocessor to ARM7TDMI-S	Coprocessor busy

5.5.1 The Coprocessor

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

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

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

5.5.2 Coprocessor Instructions in the ARM7TDMI-S Core

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 might 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 can accept the instruction.

If all these requirements are met, the ARM7TDMI-S core drives CPnI LOW, thereby committing the coprocessor to the execution of the coprocessor instruction.

5.5.3 Coprocessor Signalling

The coprocessor signals are described below:

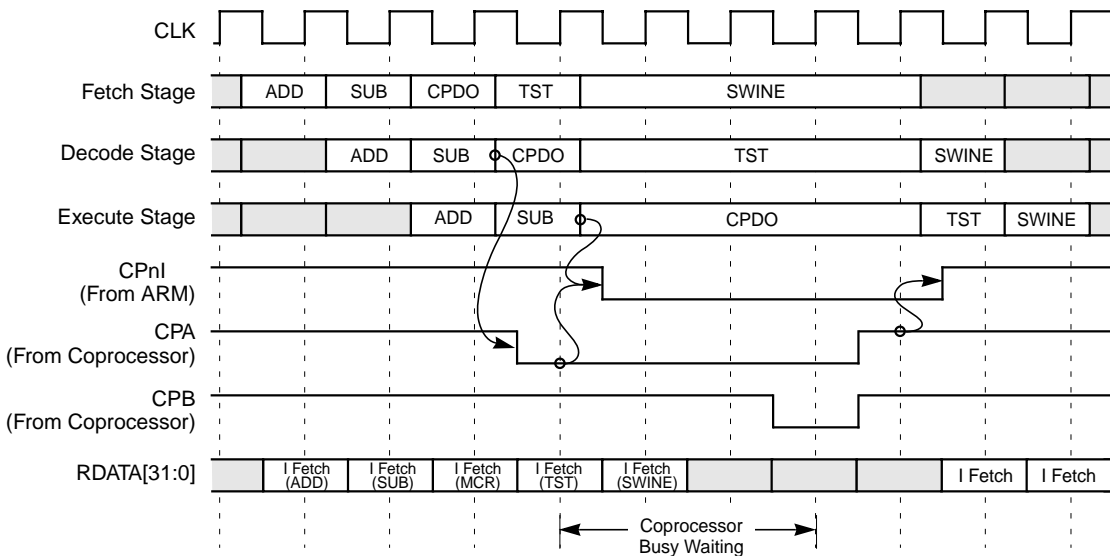
- Coprocessor absent
If a coprocessor cannot accept the instruction currently in decode, it leaves CPA and CPB both HIGH.
- Coprocessor present
If a coprocessor can accept an instruction and can start that instruction immediately, it drives both CPA and CPB LOW.
- Coprocessor busy (busy wait)
If a coprocessor can accept an instruction, but is currently unable to process that request, it asserts busy-wait to stall the ARM7TDMI-S core. A busy-wait is indicated when CPA is driven LOW, but CPB is driven HIGH. When the coprocessor is ready to start executing the instruction, it drives CPB LOW.

5.5.4 Consequences of Busy-Waiting

Figure 5.1 shows the waveforms for a coprocessor busy-wait. A busy-waited coprocessor instruction can be interrupted. If a valid FIQ or IRQ occurs (the appropriate bit is set in the CSPR register), the ARM7TDMI-S core abandons the coprocessor instruction, and drives nCPI HIGH. A coprocessor that can busy-wait must monitor nCPI to detect this condition. When the ARM7TDMI-S core abandons a coprocessor instruction, the coprocessor also abandons the instruction, and continues tracking the ARM7TDMI-S pipeline.

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

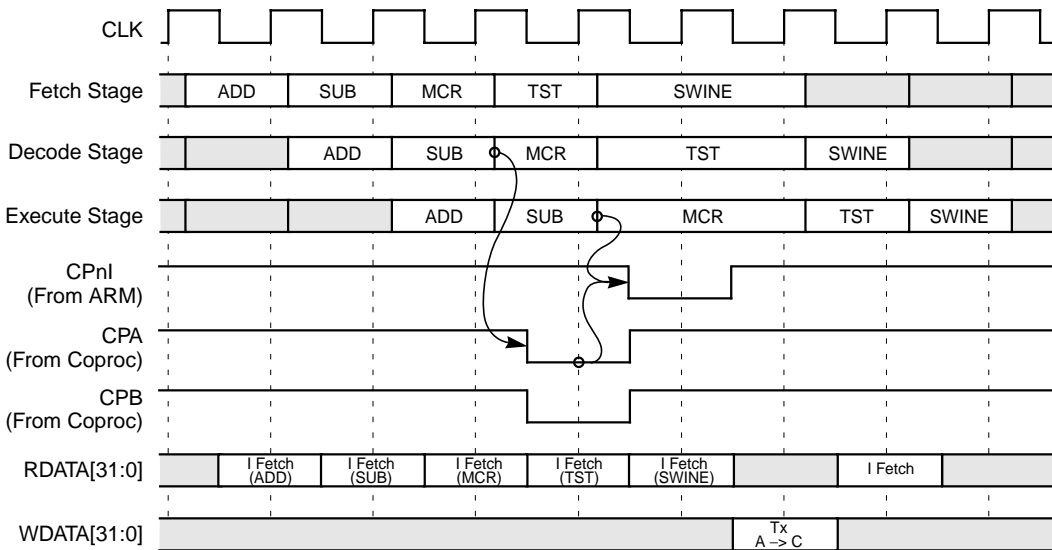
Figure 5.1 Coprocessor Busy-Wait Sequence



5.5.5 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. [Figure 5.2](#) shows an example sequence for a coprocessor register transfer.

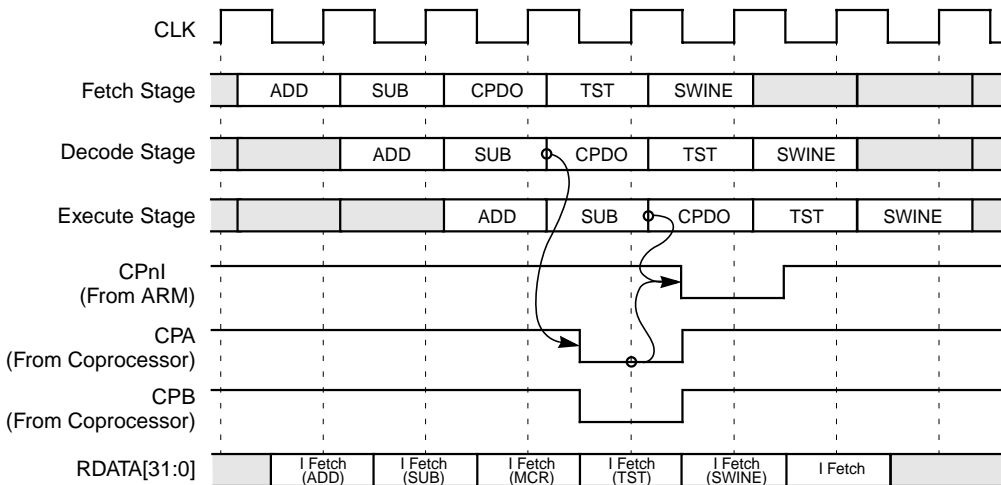
Figure 5.2 Coprocessor Register Transfer Sequence



5.5.6 Coprocessor Data Operations

Coprocessor data operations, CDP instructions, perform processing operations on the data held in the coprocessor register bank. No information is transferred between the ARM7TDMI-S core and the coprocessor as a result of this operation. [Figure 5.3](#) shows an example sequence.

Figure 5.3 Coprocessor Data Operation Sequence

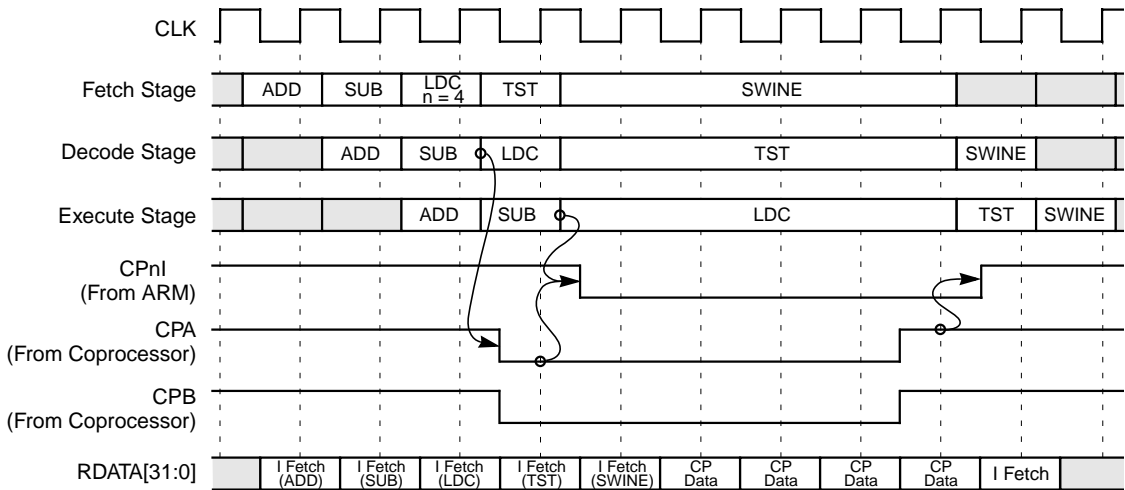


5.5.7 Coprocessor Load and Store Operations

The coprocessor load and store instructions transfer data between a coprocessor and memory. They can 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 a single LDC or STC instruction can transfer, but by convention no coprocessor should transfer more than 16 words of data in a single instruction. [Figure 5.4](#) shows an example sequence.

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

Figure 5.4 Coprocessor Load Sequence



5.6 Connecting Coprocessors

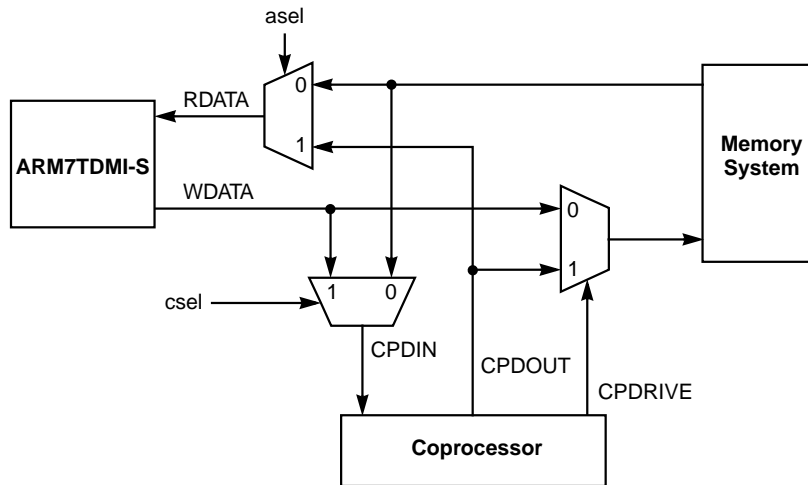
A coprocessor in an ARM7TDMI-S system needs to have 32-bit connections to:

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

5.6.1 Connecting a Single Coprocessor

Figure 5.5 shows an example of how to connect a coprocessor into an ARM7TDMI-S system.

Figure 5.5 Coprocessor Connections



5.6.2 Connecting Multiple Coprocessors

If you have multiple coprocessors in your system, connect the handshake signals as shown in Table 5.4. You must also multiplex the output data from the coprocessors.

Table 5.4 Handshake Signal Connections

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

5.7 Implementations Without External Coprocessors

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

Leave these coprocessor-specific outputs from the ARM7TDMI-S core unconnected:

- CPnMREQ
- CPSEQ
- CPnTRANS
- CPnOPC
- CPTBIT

5.8 Undefined Instructions

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

5.9 Privileged Instructions

The output signal CPnTRANS allows the implementation of coprocessors or coprocessor instructions that can only be accessed from privileged modes. [Table 5.5](#) lists the signal meanings.

Table 5.5 PROT1 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, which causes the ARM7TDMI-S core to take the undefined instruction trap.

Chapter 6

Debug Interface

This chapter describes the ARM7TDMI-S debug interface and the ARM7TDMI-S EmbeddedICE macrocell module:

- [Section 6.1, “Overview of the Debug Interface”](#)
- [Section 6.2, “Debug Systems”](#)
- [Section 6.3, “Entry into Debug State”](#)
- [Section 6.4, “ARM7TDMI SCore Clock Domains”](#)
- [Section 6.5, “Determining the Core and System State”](#)
- [Section 6.6, “Overview of EmbeddedICE”](#)
- [Section 6.7, “The Debug Communications Channel”](#)

6.1 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 core contains hardware extensions for advanced debugging features, which make it easier to develop application software, operating systems, and the hardware itself.

The debug extensions allow the core to be forced into the debug state. In debug state, the core is stopped and isolated from the rest of the system. This isolation 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 is completed, the ARM7TDMI-S core restores the core and system state, and resumes program execution.

6.1.1 Debug Stages

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 core into debug state. The interrupts that activate debug are:

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

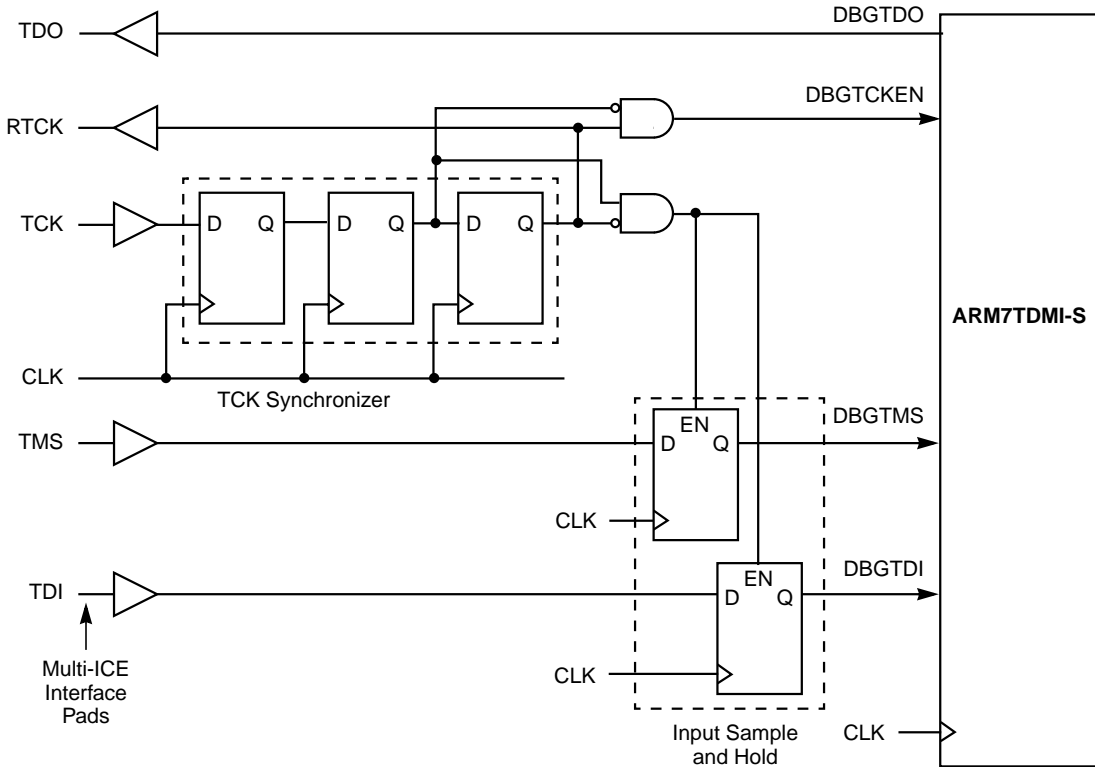
The internal state of the ARM7TDMI-S core is examined via a JTAG-style serial interface. This interface allows instructions to be serially inserted into the core pipeline without using the external data bus. So, for example, when in debug state, a store multiple (STM) could be inserted into the instruction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

6.1.2 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 (Returned TCK) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next TCK until after RTCK is received.

[Figure 6.1](#) shows this synchronization.

Figure 6.1 Clock Synchronization



6.1.3 Debug Interface Signals

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

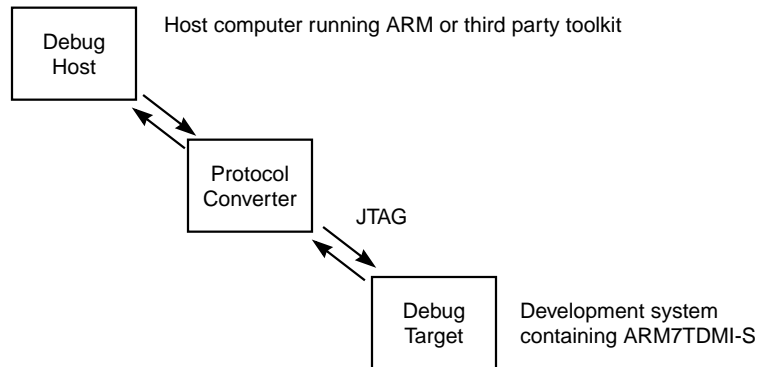
- DBGBREAK and DBGRQ are system requests for the ARM7TDMI-S core to enter debug state.
- DBGACK is used by the ARM7TDMI-S core to flag back to the system that it is in debug state.

6.2 Debug Systems

The ARM7TDMI-S core 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 core. [Figure 6.2](#) shows a typical debug system.

Figure 6.2 Typical Debug System



A debug system typically has three parts:

- The debug host
- The protocol converter
- The ARM7TDMI-S core

The debug host and the protocol converter are system-dependent. These three elements are discussed in the following subsections.

6.2.1 The Debug Host

The debug host is a computer that 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.

6.2.2 The Protocol Converter

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

6.2.3 The ARM7TDMI-S Core

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

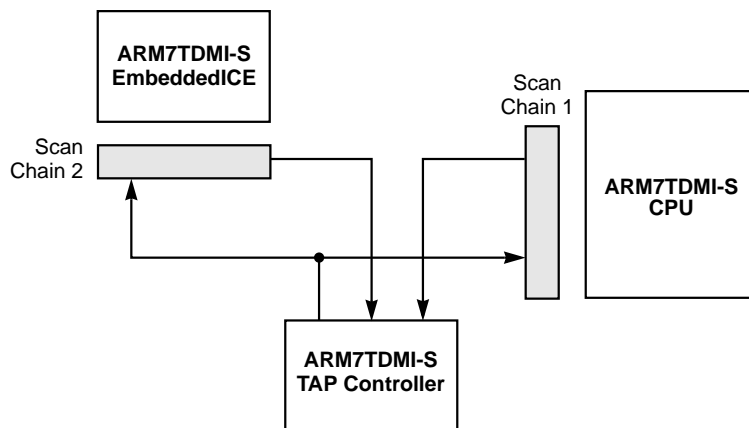
- allow the user to stall the core from program execution
- examine the core internal state
- examine the state of the memory system
- resume program execution.

The major blocks of the ARM7TDMI-S core are:

- The CPU with hardware support for debug.
- The EmbeddedICE macrocell, which is a set of registers and comparators used to generate debug exceptions (such as breakpoints). This unit is described in [Section 6.6, "Overview of EmbeddedICE."](#)
- The TAP controller, which controls the action of the scan chains via a JTAG serial interface.

These blocks are shown in [Figure 6.3](#).

Figure 6.3 ARM7TDMI-S Core Block Diagram



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

6.3 Entry into Debug State

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

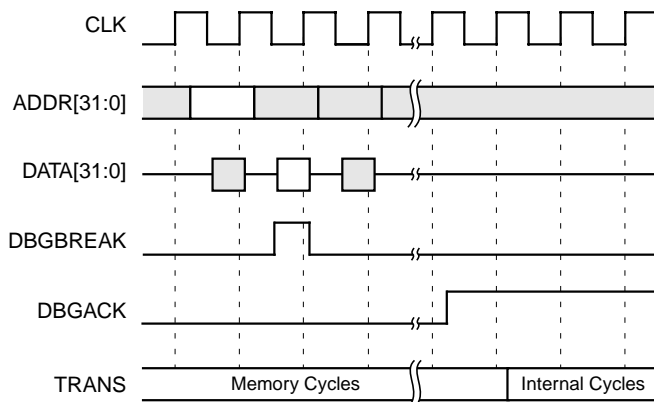
You can use EmbeddedICE to program the conditions under which a breakpoint or watchpoint might occur. Alternatively, you can use external logic to monitor the address and data bus, and flag breakpoints and watchpoints via the DBGBREAK pin.

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 by the rising edge of CLK.

Similarly, when the data is for a load or store, asserting DBGBREAK around the rising edge of CLK marks the data as watchpointed.

When a breakpoint or watchpoint is generated, there might be a delay before the ARM7TDMI-S core enters debug state. When it enters debug state, the DBGACK signal is asserted. The timing for an externally-generated breakpoint is shown in [Figure 6.4](#).

Figure 6.4 Debug State Entry



6.3.1 Entry into Debug State on Breakpoint

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

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

When a breakpointed conditional instruction reaches the execute stage of the pipeline, the breakpoint is always taken. The ARM7TDMI-S core enters debug state regardless of whether the condition was met.

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

- A branch or a write to the PC precedes the breakpointed instruction. In this case, when the branch is executed, the ARM7TDMI-S core flushes the instruction pipeline, thereby cancelling the breakpoint.
- An exception occurs, causing the ARM7TDMI-S core to flush the instruction pipeline and cancel the breakpoint. In normal circumstances, on exiting from an exception, the ARM7TDMI-S core branches back to the instruction that would have next been executed before the exception occurred. In this case, the pipeline is refilled, and the breakpoint is reflagged.

6.3.2 Entry into Debug State on Watchpoint

Watchpoints occur on data accesses. A watchpoint is always taken, but the core might not enter debug state immediately. In all cases, the current instruction completes. If the current instruction is a 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 core 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.

6.3.3 Entry into Debug State on Debug Request

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

- through EmbeddedICE programming (see [Section B.12, “Programming Breakpoints,”](#) and [Section B.13, “Programming Watchpoints”](#))
- by asserting the DBGRQ pin

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

6.3.4 Action of the ARM7TDMI-S Core in Debug State

When the ARM7TDMI-S core 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 core and function as normal. Because the rest of the system continues to operate, the ARM7TDMI-S core is forced to ignore aborts and interrupts.

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

The system must not change the CFGBIGEND signal during debug. If CFGBIGEND changes, the programmer’s view of the ARM7TDMI-S core changes with the debugger unaware that the core has reset. Also make sure that nRESET is held stable during debug. When the system applies reset to the ARM7TDMI-S core (that is, nRESET is driven LOW), the

ARM7TDMI-S core state changes with the debugger unaware that the core has reset.

6.4 ARM7TDMI SCore Clock Domains

The ARM7TDMI-S core 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 conditionally controls CLK to clock the core. When the ARM7TDMI-S core is in debug state, DBGTKEN conditionally controls CLK to clock the core.

6.5 Determining the Core and System State

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

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.

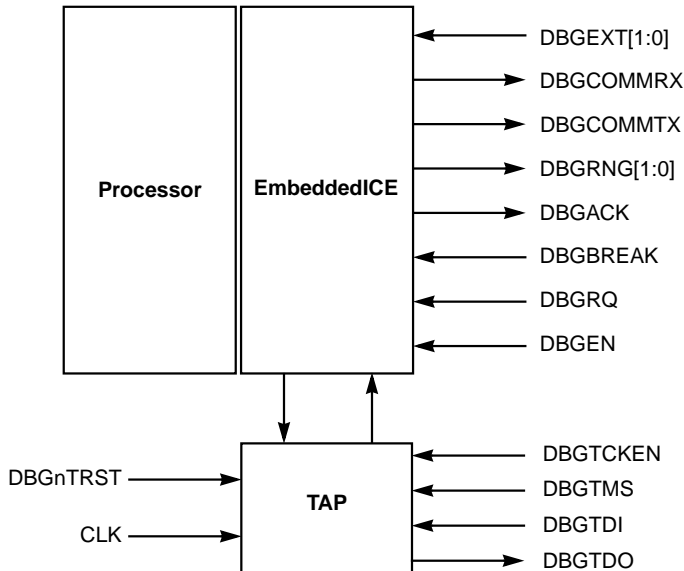
For more details about determining the core state, see [Section B.7, “Determining the Core and System State.”](#)

6.6 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 6.5](#) illustrates the relationship between the core, EmbeddedICE, and the TAP controller. It shows only the signals that are pertinent to EmbeddedICE.

Figure 6.5 The ARM7TDMI S, TAP Controller, and EmbeddedICE



The EmbeddedICE macrocell consists of:

- 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.

To disable EmbeddedICE, set the DBGEN input LOW.

Caution: Hard wiring the DBGEN input LOW permanently disables debug access.

When DBGEN is LOW, it inhibits DBGBREAK and DBGSRQ to the core, and DBGACK from the ARM7TDMI-S is always LOW.

6.7 The Debug Communications Channel

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

The communications channel consists of:

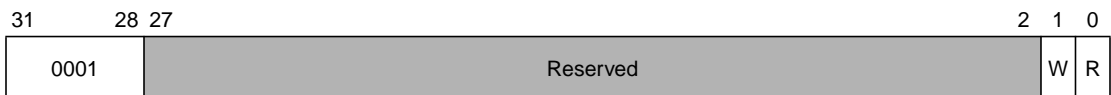
- a 32-bit Comms Data Read Register
- a 32-bit wide Comms Data Write Register
- a 6-bit Comms Control Register for synchronized handshaking between the processor and the asynchronous debugger.

These registers are located in fixed locations in the EmbeddedICE unit register map (as shown in [Figure B.5](#)) and are accessed from the processor via MCR and MRC instructions to Coprocessor 14.

6.7.1 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 6.6](#).

Figure 6.6 Debug Comms Control Register



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 can 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 then can 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 that can 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 can be placed there through the scan chain. $R = 1$ indicates 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.

Use the following instructions when accessing the Debug Comms Channel Registers:

- `MRC CP14, 0, Rd, C0, C0`
Returns the Debug Comms Control Register into Rd.
- `MCR CP14, 0, Rn, C1, C0`
Writes the value in Rn to the Comms Data Write Register.
- `MRC CP14, 0, Rd, C1, C0`
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.

6.7.2 Communications via the Comms Channel

Messages can be sent and received via the comms channel.

6.7.2.1 Sending a Message to the Debugger

When the processor wishes to send a message to the debugger, it must check to see if the Comms Data Write Register is free for use. The processor reads the Debug Comms Control Register to check the status of the W bit:

- If the W bit is cleared, the Comms Data Write Register is cleared.
- 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 cleared.

When the W bit is cleared, a message is written by a register transfer to Coprocessor 14.

Because 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 can begin again.

6.7.2.2 Receiving a Message from the Debugger

Transferring a message from the debugger to the processor is similar to sending a message to the debugger. In this case, the debugger polls the R bit of the 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 cleared, the data has been taken, and the process now can be repeated.

Chapter 7

Instruction Cycle Timing

This chapter provides the instruction cycle timing for the ARM7TDMI-S core. It contains the following sections:

- [Section 7.1, “Introduction”](#)
- [Section 7.2, “Instruction Cycle Count Summary”](#)
- [Section 7.3, “Instruction Execution Times”](#)

7.1 Introduction

The TRANS[1:0] signals predict the next cycle type. Because these signals are pipelined, the data transfer to which they apply occurs one cycle later. TRANS[1:0] are shown as such in the following tables.

In the tables in this chapter, for simplicity and ease of understanding, the following signals are treated as if they transition in the cycle to which they apply. In fact, these signals transition one cycle prior to the actual one. These signals are named as follows in the tables in this chapter:

- Address refers to ADDR[31:0]
- Lock refers to LOCK
- Size refers to SIZE[1:0]
- Write refers to WRITE
- Prot1 and Prot0 refer to PROT[1:0]
- Tbit refers to CPTBIT

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

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

Note: The letter *i* is used to indicate the instruction length.

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 core. They are therefore shown in the tables the cycle after the coprocessor has driven them.

Transaction types are shown in [Table 7.1](#).

Table 7.1 Transaction Types

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

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

7.2 Instruction Cycle Count Summary

In the pipelined architecture of the ARM7TDMI-S core, while one instruction is being fetched, the previous instruction is being decoded, and the one prior to that one is being executed.

Table 7.2 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 values in Table 7.2. These values 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 zeros or ones.
 - 2 if bits [32:16] of the multiplier operand are all zeros or ones.
 - 3 if bits [31:24] of the multiplier operand are all zeros or ones.
 - 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 7.2 Instruction Cycle Counts

Instruction	Qualifier	Cycle Count
Any unexecuted	Condition codes fail	+S
Data processing	Single cycle	+S
Data processing	Register specified shift	+I +S
Data processing	R15 destination	+N +2S
Data processing	R15, register specified shift	+I +N +2S
MUL		+(m)I +S
MLA		+I +(m)I +S

Table 7.2 Instruction Cycle Counts (Cont.)

Instruction	Qualifier	Cycle Count
MULL		+ (m)I + I + S
MLAL		+ I + (m)I + I + S
B, BL		+ N + 2S
LDR	Non R15 destination	+ N + I + S
LDR	R15 destination	+ N + I + N + 2S
STR		+ N + N
SWP		+ N + N + I + S
LDM	Non R15 destination	+ N + (n-1)S + I + S
LDM	R15 destination	+ N + (n-1)S + I + N + 2S
STM		+ N + (n-1)S + I + N
MSR, MRS		+ S
SWI, trap		+ N + 2S
CDP		+ (b)I + S
MCR		+ (b)I + C + N
MRC		+ (b)I + C + I + S
LDC, STC		+ (b)I + N + (n - 1)S + N

The cycle types N, S, I, and C are defined in [Table 7.1](#).

7.3 Instruction Execution Times

The following subsections provide more details on the individual instructions and their execution times.

7.3.1 Branch and ARM Branch with Link Instructions

Any ARM or Thumb branch and an ARM branch with link operation execute in three cycles:

1. During the first cycle, a branch instruction calculates the branch destination while performing a prefetch from the current PC. This prefetch is done in all cases because, by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch.
2. During the second cycle, the ARM7TDMI-S core performs a fetch from the branch destination. The return address is stored in r14 if the link bit is set.
3. During the third cycle, the ARM7TDMI-S core performs a fetch from the destination + i, refilling the instruction pipeline. When the instruction is a branch with link, r14 is modified (4 is subtracted from it) to simplify the return to `MOV PC,R14`.

This modification ensures subroutines of the type `STM. . {R14}`
`LDM. . {PC}` work correctly.

Table 7.3 shows the cycle timings, where:

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

Table 7.3 Branch Instruction Cycle Operations

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

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

7.3.2 Thumb Branch with Link Instructions

A Thumb Branch with Link (BL) operation consists of two consecutive Thumb instructions, and executes in 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 core calculates the final branch destination while performing a prefetch from the current PC.
 - During the second cycle, the ARM7TDMI-S core performs a fetch from the branch destination. The return address is stored in r14.
 - During the third cycle, the ARM7TDMI-S core performs a fetch from the destination +2, refills the instruction pipeline, and modifies r14 (subtracting 2) to simplify the return to `MOV PC, R14`. This modification ensures that subroutines of the type `PUSH {..,LR}; POP {..,PC}` work correctly.

Table 7.4 shows the cycle timings of the complete operation.

Table 7.4 Thumb Long Branch with Link

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

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

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

7.3.3 Branch and Exchange Instruction

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

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

Table 7.5 shows the cycle timings.

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.

Table 7.5 Branch and Exchange Instruction Cycle Operations

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

7.3.4 Data Operations

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

The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction. The ARM7TDMI-S core writes the result (when required) into the destination register. (Compares and tests do not produce results, only the ALU status flags are affected.)

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

When a register specifies the shift length, an additional data path cycle occurs before the data operation to copy the bottom eight 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 core writes the result to the PC, the contents of the instruction pipeline are invalidated, and the ARM7TDMI-S core takes the address for the next instruction prefetch from the ALU rather than the address incrementer. The ARM7TDMI-S core 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 7.6](#).

Table 7.6 Data Operation Instruction Cycle Operations

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

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

7.3.5 Multiply and Multiply Accumulate Operations

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 7.7](#) to [Table 7.10](#), in which m is the number of cycles required by the multiplication algorithm (see [Section 7.2, "Instruction Cycle Count Summary," page 7-3](#)).

Table 7.7 Multiply Instruction Cycle Operations

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

Table 7.8 Multiply Accumulate Instruction Cycle Operations

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

Table 7.9 Multiply Long Instruction Cycle Operations

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

Note: Multiply long is available only in ARM state.

Table 7.10 Multiply Accumulate Long Instruction Cycle Operations

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

Note: Multiply-accumulate long is available only in ARM state.

7.3.6 Load Register Instruction

A load register instruction executes in a variable number of cycles:

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

The load register cycle timings are shown in [Table 7.11](#), where:

- b, h and w are byte, halfword, and word as defined in the SIZE field on [page B-29](#).
- s represents current supervisor-mode-dependent value.
- u is either 0, when the force translation bit is specified in the instruction (LDRT), or s at all other times.

Table 7.11 Load Register Instruction Cycle Operations

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

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

Note: Destination equals PC is not possible in Thumb state.

7.3.7 Store Register Instruction

A store register executes in two cycles:

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

The store register cycle timings are shown in [Table 7.12](#), where:

- s represents current mode-dependent value.
- t is either 0 (when the T bit is specified in the instruction (STRT)) or c at all other times.

Table 7.12 Store Register Instruction Cycle Operations

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

7.3.8 Load Multiple Register Operations

A load multiple (LDM) executes in four cycles:

1. During the first cycle, the ARM7TDMI-S core calculates the address of the first word to be transferred, while performing a prefetch from memory.
2. During the second cycle, the ARM7TDMI-S core fetches the first word and performs the base modification.

3. During the third cycle, the ARM7TDMI-S core moves the first word to the appropriate destination register, and fetches the second word from memory. The ARM7TDMI-S core 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 core moves the last word to its destination register. The last cycle can be merged with the next instruction prefetch to form a single memory N-cycle.

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

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

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

The LDM cycle timings are shown in [Table 7.13](#).

Table 7.13 Load Multiple Registers Instruction Cycle Operations

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

Table 7.13 Load Multiple Registers Instruction Cycle Operations (Cont.)

Cycle		Address	Size	Write	Data	TRANS[1:0]	Prot0
	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				

7.3.9 Store Multiple Register Instructions

Store multiples (STM) proceed identically as load multiples, without the final cycle. They execute in two cycles:

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

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

The STM cycle timings are shown in [Table 7.14](#).

Table 7.14 Store Multiple Registers Instruction Cycle Operations

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

7.3.10 Data Swap Operations

Data swaps are similar to 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. In the third cycle, the contents of the source register are written to the external memory. In the fourth cycle the data read during cycle 2 is written into the destination register.

The data swapped can be a byte or word quantity (b/w). The ARM7TDMI-S core might abort the swap operation in either the read or write cycle. The swap operation (read or write) does not affect the destination register.

The data swap cycle timings are shown in [Table 7.15](#), where b and w are byte and word as defined in the SIZE field on [page B-29](#).

Table 7.15 Data Swap Instruction Cycle Operations

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

Note: Data swap cannot be executed in Thumb state.

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

7.3.11 Software Interrupt and Exception Entry

Exceptions and software interrupts (SWIs) force the PC to a specific value and refill the instruction pipeline from this address:

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

The SWI cycle timings are shown in [Table 7.16](#), where:

- s represents the current supervisor-mode-dependent value.
- t represents the current Thumb-state value.
- pc is, for software interrupts, the address of the SWI instruction. For exceptions, it is the address of the instruction following the last one to be executed before entering the exception. For prefetch aborts, pc

is the address of the aborting instruction. For data aborts, it is the address of the instruction following the one that attempted the aborted data transfer.

- Xn is the appropriate trap address.

Table 7.16 Software Interrupt Instruction Cycle Operations

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

7.3.12 Coprocessor Data Processing Operation

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

If the coprocessor cannot perform the requested task, it leaves CPA and CPB HIGH.

When the coprocessor can perform the task, but cannot commit immediately, the coprocessor drives CPA LOW, but leaves CPB HIGH until able to commit. The ARM7TDMI-S core busy-waits until CPB goes LOW. However, an interrupt can cause the ARM7TDMI-S core to abandon a busy-waiting coprocessor instruction (see [Section 5.5.4, “Consequences of Busy-Waiting,” page 5-7](#)).

The coprocessor data operations cycle timings are shown in [Table 7.17](#).

Table 7.17 Coprocessor Data Operation Instruction Cycle Operations

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

Note: Coprocessor operations are available only in ARM state.

7.3.13 Load Coprocessor Register (from Memory to Coprocessor)

The load coprocessor (LDC) operation transfers one or more words of data from memory to coprocessor registers. The coprocessor commits to the transfer only when it is ready to accept the data. The WRITE line is driven LOW during the transfer cycle. When CPB goes LOW, the ARM7TDMI-S core produces addresses and expects the coprocessor to take the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred. An interrupt can cause the ARM7TDMI-S core to abandon a busy-waiting coprocessor instruction (see [Section 5.5.4, “Consequences of Busy-Waiting,” page 5-7](#)).

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

The load coprocessor register cycle timings are shown in [Table 7.18](#).

Table 7.18 Load Coprocessor Register Instruction Cycle Operations

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

Note: Coprocessor operations are available only in ARM state.

7.3.14 Store Coprocessor Register (from Coprocessor to Memory)

The store coprocessor (STC) operation transfers one or more words of data from coprocessor registers to memory. The coprocessor commits to the transfer only when it is ready to write data. The WRITE line is driven HIGH during the transfer cycle. When CPB goes LOW, the ARM7TDMI-S core produces addresses, and expects the coprocessor to write the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred. An interrupt can cause the ARM7TDMI-S core to abandon a busy-waiting coprocessor instruction (see [Section 5.5.4, "Consequences of Busy-Waiting," page 5-7](#)).

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

The store coprocessor register cycle timings are shown in [Table 7.19](#).

Table 7.19 Store Coprocessor Register Instruction Cycle Operations

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

Note: Coprocessor operations are available only in ARM state.

7.3.15 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 is written to the ARM register during the third cycle of the operation.

If the coprocessor asserts CPB to indicate a busy-wait, an interrupt can cause the ARM7TDMI-S core to abandon the coprocessor instruction (see [Section 5.5.4, “Consequences of Busy-Waiting,” page 5-7](#)).

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

The MRC cycle timings are shown in [Table 7.20](#).

Table 7.20 Coprocessor Register Transfer (MRC)

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

Note: This operation cannot occur in Thumb state.

7.3.16 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 asserts CPB to indicate a busy-wait, an interrupt can cause the ARM7TDMI-S core to abandon the coprocessor instruction (see [Section 5.5.4, “Consequences of Busy-Waiting,” page 5-7](#)).

The MCR cycle timings are shown in [Table 7.21](#).

Table 7.21 Coprocessor Register Transfer (MCR)

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

7.3.17 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 7.22](#). where:

- s represents the current mode-dependent value.
- t represents the current state-dependent value.

Note: Coprocessor operations are available only in ARM state.

Table 7.22 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											

Note: Coprocessor operations are available only in ARM state.

7.3.18 Unexecuted Instructions

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

Unexecuted instruction cycle timings are shown in [Table 7.23](#).

Table 7.23 Unexecuted Instruction Cycle Operations

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

Appendix A

Differences Between the ARM7TDMI-S and the ARM7TDMI

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

- [Section A.1, “ARM7TDMI-S Signal Naming and ARM7TDMI Equivalentents”](#)
- [Section A.2, “ARM7TDMI Signals Removed from ARM7TDMI-S Core”](#)
- [Section A.3, “ATPG Scan Interface”](#)
- [Section A.4, “Timing Parameters”](#)
- [Section A.5, “Design Considerations when Converting to ARM7TDMI-S”](#)

A.1 ARM7TDMI-S Signal Naming and ARM7TDMI Equivalentents

The ARM7TDMI-S signal names have prefixes that identify groups of functionally-related signals:

- CFGxxx indicates configuration inputs (typically hardwired for an embedded application).
- CPxxx indicates coprocessor expansion interface signals.
- DBGxxx indicates scan-based EmbeddedICE debug support inputs or outputs.

Other signals provide the system designer's interface, which is primarily memory-mapped. [Table A.1](#) provides the ARM7TDMI-S signals with their ARM7TDMI hard macrocell equivalent signals.

Table A.1 ARM7TDMI S Signals and ARM7TDMI Hard Macrocell Equivalents

ARM7TDMI S Signal	Function	ARM7TDMI Hard Macrocell Equivalent
ABORT	1 = memory abort or bus error. 0 = no error.	ABORT
ADDR[31:0] ¹	32-bit address output bus, available in the cycle preceding the memory cycle.	A[31:0]
CFGBIGEND	1 = big-endian configuration. 0 = little-endian configuration.	BIGEND
CLK ²	Master rising edge clock. All inputs are sampled on the rising edge of CLK. All timing dependencies are from the rising edge of CLK.	MCLK
CLKEN ³	System memory interface clock enable: 1 = advance the core on rising CLK. 0 = prevent the core advancing on rising CLK.	NWAIT
CPA ⁴	Coprocessor absent. Tie HIGH when no coprocessor is present.	CPA
CPB ⁴	Coprocessor busy. Tie HIGH when no coprocessor is present.	CPB
CPnI	Active LOW coprocessor instruction execute qualifier.	nCPI
CPnMREQ	Active LOW memory request signal, pipelined in the preceding access. This coprocessor interface signal uses the ARM7TDMI-S output TRANS[1:0] for bus interface design.	nMREQ
CPnOPC	Active LOW opcode fetch qualifier output, pipelined in the preceding access. This coprocessor interface signal uses the ARM7TDMI-S output PROT[1:0] for bus interface design.	nOPC
CPnTRANS	Active LOW supervisor mode access qualifier output. This coprocessor interface signal uses the ARM7TDMI-S output PROT[1:0] for bus interface design.	nTRANS
CPSEQ	Sequential address signal. This coprocessor interface signal uses 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

Table A.1 ARM7TDMI S Signals and ARM7TDMI Hard Macrocell Equivalents

ARM7TDMI S Signal	Function	ARM7TDMI Hard Macrocell Equivalent
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
DBGGEN	Debug enable. Tie this signal HIGH in order to be able to use the debug features of the ARM7TDMI.	DBGGEN
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 ⁵	Active LOW TAP controller DBGTDO output qualifier.	nTDOEN
DBGnTRST ⁵	Active LOW TAP controller reset (asynchronous assertion). Resets the ICEBreaker subsystem.	nTRST
DBG RNG[1:0]	EmbeddedICE rangeout qualifier outputs.	RANGEOUT1, RANGEOUT0
DBG RQ ⁶	External debug request (tie LOW when not required).	DBG RQ
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 ⁵	Multi-ICE TDI test data input.	TDI
DBG TDO ⁵	EmbeddedICE TAP controller serial data output.	TDO
DBG TMS ⁵	Multi-ICE TMS test mode select input.	TMS
LOCK ¹	Indicates whether the current address is part of locked access. Execution of a SWP instruction generates this signal.	LOCK
nFIQ ⁷	Active LOW fast interrupt request input.	nFIQ
nIRQ ⁷	Active LOW interrupt request input.	nIRQ
nRESET	Active LOW reset input (asynchronous assertion). Resets the processor core subsystem.	nRESET

Table A.1 ARM7TDMI S Signals and ARM7TDMI Hard Macrocell Equivalents

ARM7TDMI S Signal	Function	ARM7TDMI Hard Macrocell Equivalent
PROT[1:0] ^{1,8}	Protection output, indicates whether the current address is being accessed as instruction or data, and whether it is being accessed in a privileged mode or user mode.	nOPC, nTRANS
RDATA[31:0] ⁹	Unidirectional 32-bit input data bus.	DIN[31:0]
SIZE[1:0]	Indicates the width of the bus transaction to the current address: 00 = 8-bit. 01 = 16-bit. 10 = 32-bit. 11 = not supported.	MAS[1:0]
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 ¹	Write access indicator.	nRW

1. All 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, it is possible for the signals to change in the first phase of the clock cycle, which is unlike the ARM7TDMI hard macrocell where they 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. Thus the address class outputs (ADDR[31:0], WRITE, SIZE[1:0], PROT[1:0], and LOCK) might still change in a cycle in which CLKEN is taken LOW. You must take this possibility into account when designing a memory system.
4. CPA and CPB are sampled on the rising edge of CLK. They might no longer change in the first phase of the next cycle, as is possible with the ARM7TDMI hard macrocell.
5. 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.
6. DBGRQ must be synchronized externally to the macrocell. It is not an asynchronous input as on the ARM7TDMI hard macrocell.
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. PROT[0] is the equivalent of nOPC, and PROT[1] is the equivalent of nTRANS on the ARM7TDMI hard macrocell.
9. The ARM7TDMI-S core 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.

A.2 ARM7TDMI Signals Removed from ARM7TDMI-S Core

This section lists the ARM7TDMI signals that were not incorporated into the ARM7TDMI-S design. There were three reasons for the removals:

1. 3-state enables and bidirectional signals were removed because the ARM7TDMI-S interface does not use them.
2. Boundary scan and test clock were removed. Because the ARM7TDMI-S core is not tested via boundary scan, the importance of the TAP controller for internal and system test diminished. Because the ARM7TDMI-S uses only one clock, other signals were removed.
3. The design was changed to be synchronous, single-edge dependent and latchless without gated clocks. Redundant signals were also removed.

[Table A.2](#) lists the ARM7TDMI signals that are not in the ARM7TDMI-S interface. The number in the last column correspond to the reason for removal above.

Table A.2 ARM7TDMI Signals not on ARM7TDMI-S

ARM7TDMI Signal Name	Description	Reason for Removal
ABE	Address bus enable	1
ALE	Address latch enable	1
APE	Address pipeline enable	3
BL[3:0]	Byte latch control	3
BUSDIS	Bus disable	1, 2
BUSEN	Data bus configuration	1
D[31:0]	Data bus	1
DBE	Data bus enable	1
DBGREQ	Internal debug request	2
DRIVEBS	Boundary scan cell enable	2

Table A.2 ARM7TDMI Signals not on ARM7TDMI-S

ARM7TDMI Signal Name	Description	Reason for Removal
ECAPCLK	Exttest capture clock	2
ECAPCLKBS	Exttest capture clock for boundary scan	2
ECLK	External clock output	2
HIGHZ	HIGHZ instruction	2
ICAPCLKBS	TCK pulse	2
IR[3:0]	TAP controller instruction register	2
ISYNC	Synchronous interrupts	3
nENIN	NOT enable input	1
nENOUT	NOT enable output	1
nENOUTI	NOT enable output	1
nHIGHZ	NOT HIGHZ	2
nM[4:0]	NOT processor mode	3
PCLKBS	Boundary scan update clock	2
RSTCLKBS	Boundary scan reset clock	2
SCREG[3:0]	Scan chain register	2
SDINBS	Boundary scan serial input data	2
SDOUTBS	Boundary scan serial output data	2
SHCLKBS	Boundary scan shift clock, phase 1	2
SHCLK2BS	Boundary scan shift clock, phase 2	2
TAPSM[3:0]	TAP controller state machine	2
TBE	Test bus enable	1, 2
TCK	Test clock	2
TCK1	TCK, phase 1	2
TCK2	TCK, phase 2	2

A.3 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 and HIGH for scan test
- SCANENABLE2 is LOW for normal usage and HIGH for scan test
- SCANIN is the serial scan path input
- SCANOUT is the serial scan path output

A.4 Timing Parameters

All inputs are sampled on the rising edge of CLK.

The clock enables are sampled on every rising clock edge.

All other inputs are sampled on rising edge of CLK when their corresponding clock enable is active HIGH.

Outputs are all sampled on the rising edge of CLK with the appropriate clock enable (CLKEN) active.

A.5 Design Considerations when Converting to ARM7TDMI-S

When an ARM7TDMI hard macrocell design is being converted to the ARM7TDMI-S core, a number of areas require special consideration. These are the:

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

A.5.1 Master Clock

CLK is the master clock to the ARM7TDMI-S core. It 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.

A.5.2 JTAG Interface Timing

All JTAG signals on the ARM7TDMI-S core are synchronous to the master clock input, CLK. When an external TCLK is used, use an external synchronizer to the ARM7TDMI-S core.

A.5.3 Interrupt Timing

As with all ARM7TDMI-S signals, the interrupt signals, nIRQ and nFIQ, are sampled on the rising edge of CLK.

When converting an ARM7TDMI hard macrocell design where ISYNC is asserted LOW, add a synchronizer to the design to synchronize the interrupt signals before they are applied to the ARM7TDMI-S core.

A.5.4 Address Class Signal Timing

The address class outputs (ADDR[31:0], WRITE, SIZE[1:0], PROT[1:0], and LOCK) on the ARM7TDMI-S core all change in response to the rising edge of CLK. Thus 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 core to make sure that they can change only in the second phase of the clock.

Because CLKEN 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 behavior is similar to that 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.

Make sure that the correct address is used for the memory cycle, even though ADDR[31:0] might have moved on to the address for the next memory cycle.

For further details, refer to [Chapter 4, “Memory Interface.”](#)

Appendix B

Detailed Debug Operation

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

- [Section B.1, “Scan Chains and JTAG Interface”](#)
- [Section B.2, “Resetting the TAP Controller”](#)
- [Section B.3, “Instruction Register”](#)
- [Section B.4, “Public Instructions”](#)
- [Section B.5, “Test Data Registers”](#)
- [Section B.6, “ARM7TDMI-S Core Clock Domains”](#)
- [Section B.7, “Determining the Core and System State”](#)
- [Section B.8, “Behavior of the Program Counter During Debug”](#)
- [Section B.9, “Priorities and Exceptions”](#)
- [Section B.10, “Scan Interface Timing”](#)
- [Section B.11, “The Watchpoint Registers”](#)
- [Section B.12, “Programming Breakpoints”](#)
- [Section B.13, “Programming Watchpoints”](#)
- [Section B.14, “The Debug Control Register”](#)
- [Section B.15, “The Debug Status Register”](#)
- [Section B.16, “Coupling Breakpoints and Watchpoints”](#)
- [Section B.17, “EmbeddedICE Issues”](#)

B.1 Scan Chains and JTAG Interface

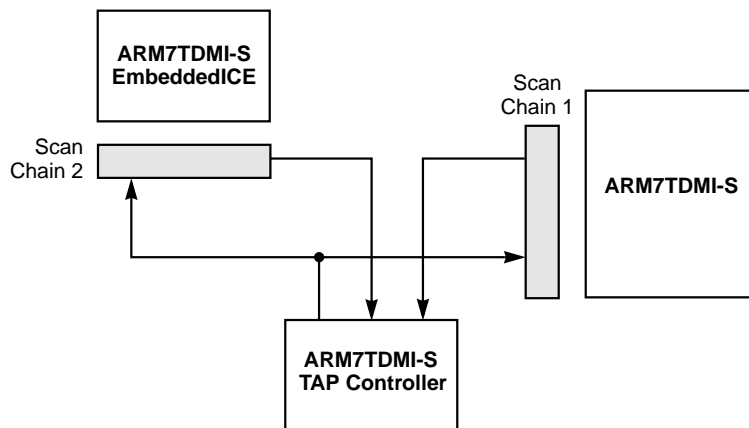
Two JTAG-style scan chains within the ARM7TDMI-S core allow debugging and EmbeddedICE programming. A JTAG-style Test Access Port (TAP) controller controls the scan chains.

For further details of the JTAG specification, refer to IEEE Standard 1149.1 - 1990 Standard Test Access Port and Boundary-Scan Architecture.

B.1.1 Scan Limitations

The two scan paths are referred to as Scan Chain 1 and Scan Chain 2. They are shown in [Figure B.1](#).

Figure B.1 ARM7TDMI SScan Chain Arrangements



B.1.1.1 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 is (from serial data in to out):

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

The ARM7TDMI-S core provides data for Scan Chain 1 cells as shown in [Table B.1](#).

Table B.1 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
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

Table B.1 Scan Chain 1 Cells (Cont.)

Number	Signal	Type
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

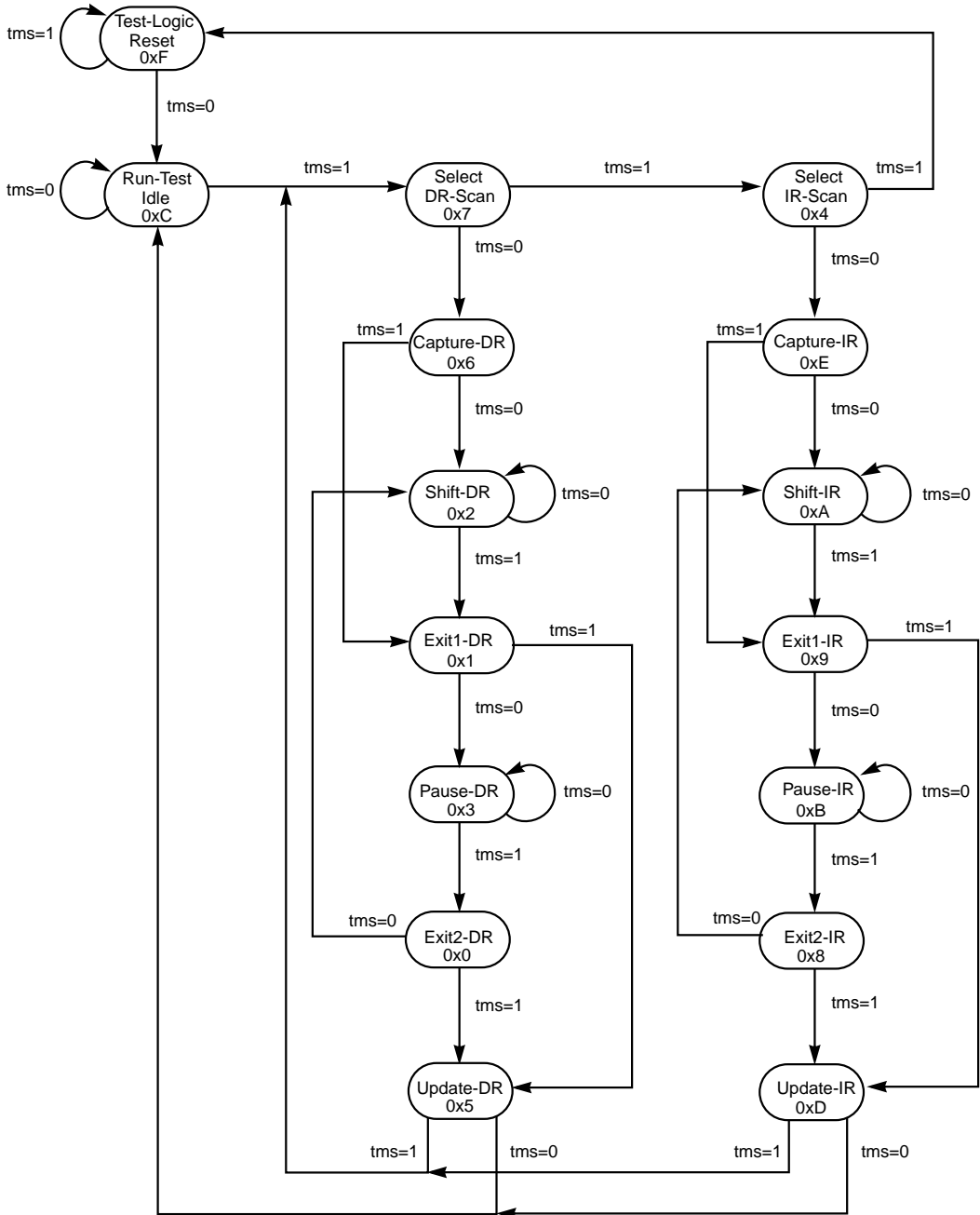
B.1.1.2 Scan Chain 2

Scan Chain 2 allows access to the EmbeddedICE registers. Refer to [Section B.5, “Test Data Registers,”](#) for details.

B.1.2 TAP State Machine

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

Figure B.2 Test Access Port Controller State Transitions



B.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 pulse the DBGnTRST signal as follows:

- 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 can be tied LOW.

Note: A clock on CLK with DBGTKEN HIGH is not necessary to reset the device.

The action of reset is as follows:

1. System mode is selected. 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 DBGTKEN, the contents of the ID register are clocked out of DBGTDO.

B.3 Instruction Register

The Instruction Register is four bits in length. There is no parity bit.

The fixed value 0001 is loaded into the Instruction Register during the CAPTURE-IR controller state.

B.4 Public Instructions

Table B.2 lists the public instructions. In the following descriptions, the ARM7TDMI-S core samples DBGTDI and DBGTMS on the rising edge of CLK with DBGTKEN HIGH.

Table B.2 Public Instructions

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

B.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 four bits long in this implementation, although no finite length is specified.

B.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 are 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.

B.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 [Section B.5.2, “ARM7TDMI-S Device Identification \(ID\) Code Register,”](#) 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 ID register captures the device identification code.
- 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.

B.4.4 BYPASS (1111)

The BYPASS instruction connects a one-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.

B.4.5 RESTART (0100)

The RESTART instruction restarts the processor on exit from debug state. The RESTART instruction connects the Bypass Register between DBGTDI and DBGTDO, 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.

B.5 Test Data Registers

Five test data registers can 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 DBGTKEN enable is HIGH.

B.5.1 Bypass Register

This one-bit register provides a path between DBGTDI and DBGTDO to bypass the device during scan testing.

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 DBGTKEN.

There is no parallel output from the Bypass Register.

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

B.5.2 ARM7TDMI-S Device Identification (ID) Code Register

This 32-bit register reads the 32-bit device identification code. No programmable supplementary identification code is provided. The default device identification code is 0x0F1F.0F0F.

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.

31	28 27	12 11	1 0
Version	Part Number	Manufacturer Identity	1

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

B.5.3 Instruction Register

This four-bit register changes the current TAP instruction.

In the SHIFT-IR state, the Instruction Register is selected as the serial path between DBGTDI and DBGTDO.

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

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

On reset, IDCODE becomes the current instruction.

B.5.4 Scan Path Select Register

This four-bit register changes the current active scan chain.

SCAN_N as the current instruction in the SHIFT-DR state selects the Scan Path Select Register as the serial path between DBGTDI and DBGTDO.

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

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

[Table B.3](#) shows the scan chain number allocation.

Table B.3 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.

B.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 each consists of a serial register and a multiplexer.

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 multiplexer 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.

The TAP controller internally generates all the control signals for the scan cells.

The current instruction and the state of the TAP state machine determine the action of the TAP controller.

B.5.5.1 Scan Chain 1

Scan Chain 1 is used to communicate 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_NTAP instruction can be used to select Scan Chain 1.

Scan Chain 1 is 33 bits long: 32 bits for the data value and one bit for the scan cell on the DBGBREAK core input.

The scan chain order from DBGTDI to DBGTDO is the ARM7TDMI-S data bits (bits 0 to 31) followed by the 33rd bit (the DBGBREAK scan cell). 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 core synchronizes back to system speed before executing the instruction. See [Section B.8.5, “System Speed Access,”](#) for further details.
- After the ARM7TDMI-S core has entered debug state, the value of the 33rd bit on the first occasion that it is captured and scanned out tells the debugger whether the core entered debug state from a breakpoint (bit 33 LOW) or from a watchpoint (bit 33 HIGH).

B.5.5.2 Scan Chain 2

Scan Chain 2 allows access to the EmbeddedICE registers. Scan Chain 2 must be selected using the SCAN_N TAP controller instruction, and then the TAP controller must be put in INTEST mode.

Scan Chain 2 is 38 bits long. The scan chain order from DBGTDI to DBGTDO is the read/write bit, the register address bits (bits 4 to 0), and 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 B.5](#) for further details.

B.6 ARM7TDMI-S Core Clock Domains

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

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

During normal operation, CLKEN conditionally enables CLK to clock the core. When the ARM7TDMI-S core is in debug state, DBGCKEN conditionally enables CLK to clock the core.

B.7 Determining the Core and System State

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

Before you can examine the core and system state, the debugger must determine whether the processor entered debug from Thumb state or ARM state. It examines bit 4 of the EmbeddedICE Debug Status Register. When bit 4 is HIGH, the core has entered debug from Thumb state; when bit 4 is LOW the core has entered debug entered from ARM state.

B.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. You can use any register.

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

The debugger must restore the original mode before exiting debug state. For example, if the debugger 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, 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 core is fully static. However, you cannot use this method for determining the state of the rest of the system.

While in debug state, only the following instructions can be scanned into the instruction pipeline for execution:

- all data processing operations, except TEQP
- all load, store, load multiple, and store multiple instructions
- MSR and MRS

B.7.2 Determining System State

In order to meet the dynamic timing requirements of the memory system, any attempt to access the system state must occur with the clock qualified by CLKEN. To perform a memory access, CLKEN must be used to force the ARM7TDMI-S core to run in normal operating mode. Bit 33 of Scan Chain 1 controls this.

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 core to:

1. Switch automatically to CLKEN control.
2. Execute the instruction at system speed.
3. Re-enter debug state.

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

The debugger usually uses EmbeddedICE to control debugging, and so the states of TRANS[1:0] and DBGACK can be determined by reading the EmbeddedICE status register. Refer to [Section B.15, “The Debug Status Register,”](#) 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 multiples
- store multiples

See also [Section B.7.3, “Exit from Debug State.”](#)

When the ARM7TDMI-S core 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.

B.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 [Section B.8, “Behavior of the Program Counter During Debug,”](#) for details on calculating the branch.

Bit 33 of Scan Chain 1 forces the ARM7TDMI-S core 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 core 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 core 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 core 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 therefore must 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 might 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 B.3 shows the behavior of the ARM7TDMI-S core on exit from the debug state.

Figure B.3 Debug Exit Sequence

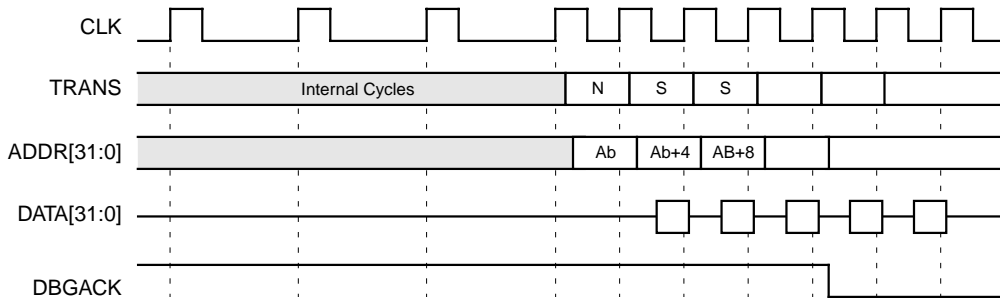


Figure B.2 shows that the final memory access occurs in the cycle after DBGACK goes HIGH. This point is the one at which the cycle counter is disabled. Figure B.3 shows that the first memory access that the cycle counter has not previously seen occurs in the cycle after DBGACK goes LOW. The counter is to be re-enabled at this point.

Note: When a system speed access from debug state occurs, the ARM7TDMI-S core temporarily drops out of debug state, and so DBGACK can go LOW. If there are peripherals that are sensitive to the number of memory accesses, to make them believe that the ARM7TDMI-S core is still in debug state, program the EmbeddedICE Control Register to force the value on DBGACK to be HIGH. See [Section B.15, “The Debug Status Register,”](#) for more details.

B.8 Behavior of the Program Counter During Debug

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

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

B.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 four 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 core 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, 2 for the instructions, and 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 core 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).

B.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 [Section B.8.1, "Breakpoints,"](#) above.

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

B.8.3 Watchpoint with Another Exception

If a watchpointed access simultaneously causes a data abort, the ARM7TDMI-S core enters debug state in abort mode. Entry into debug

is held off until the core changes into abort mode, and has fetched the instruction from the abort vector.

A similar sequence follows when an interrupt or any other exception occurs during a watchpointed memory access. The ARM7TDMI-S core 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; this case must be considered in return branch calculation when exiting the debug state. For example, suppose that an abort has occurred on a watchpointed access and 10 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 is refetched and executed. This case triggers the watchpoint again, and the ARM7TDMI-S core re-enters debug state.

B.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.

B.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 core 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.

B.8.6 Summary of Return Address Calculations

The calculation of the branch return address is as follows:

- for normal breakpoints and watchpoints, the branch is:
– $(4 + N + 3S)$
- for entry through debug request (DBGRQ), or watchpoint with exception, the branch is:

$$- (3 + N + 3S)$$

where N is the number of debug speed instructions executed (including the final branch), and S is the number of system speed instructions executed.

B.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 [Section B.8, “Behavior of the Program Counter During Debug.”](#) This section covers the priorities.

B.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 provided the breakpoint is activated (it might be data-dependent), the ARM7TDMI-S core enters the debug state.

Therefore the prefetch abort takes higher priority than the breakpoint.

B.9.2 Interrupts

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

Therefore debug takes higher priority than the interrupt, but the ARM7TDMI-S core does remember that an interrupt has occurred.

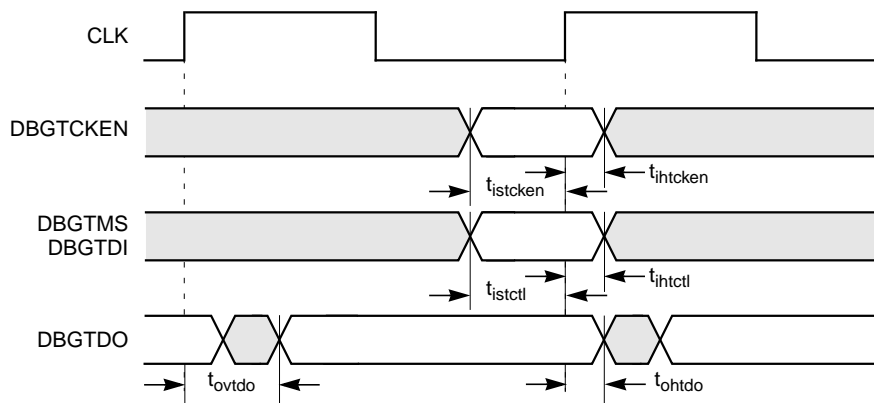
B.9.3 Data Aborts

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

B.10 Scan Interface Timing

Figure B.4 provides general scan timing information.

Figure B.4 General Scan Timing



B.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 B.4](#).

Table B.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

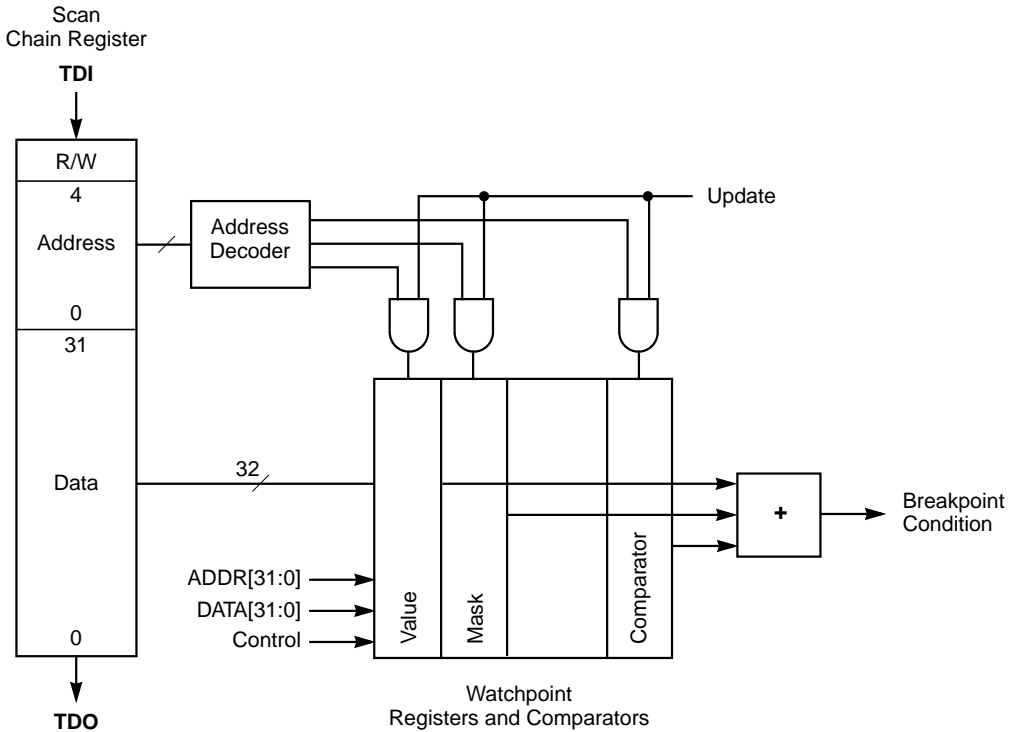
B.11.1 Programming and Reading Watchpoint Registers

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 consisting of:

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

This setup is shown in [Figure B.5](#).

Figure B.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 five-bit address field. A one is shifted into the read/write bit.

To read a register, its address is shifted into the address field and a 0 is shifted into the read/write bit. The 32-bit data field is ignored.

The register addresses are shown in [Table B.4](#).

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

B.11.2 Using the Watchpoint 0/1 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 Watchpoint 0/1 Data Mask Register can be programmed to 0xFFFF.FFFF (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 Watchpoint 0/1 Value register.

B.11.3 Watchpoint 0/1 Control and Mask Registers

Figure B.6 shows the Watchpoint 0/1 Control Value register. The lower eight bits of this register and the Watchpoint 0/1 Control Mask Register are mapped identically.

Figure B.6 Watchpoint 0/1 Control Value Register

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

Figure B.7 Watchpoint 0/1 Control Mask Register

8	7	6	5	4	3	2	1	0
res	RANGE_MASK	CHAIN_MASK	DBGEXT_MASK	PROT[1]_MASK	PROT[0]_MASK	SIZE_MASK[1:0]	WRITE_MASK	

ENABLE **Enable bit** **8**
 This bit is the ENABLE bit and cannot be masked.
 When a watchpoint match occurs, the internal DBG-BREAK signal is asserted only when the ENABLE bit is set. This bit exists only in the value register. It cannot be masked.

RANGE	<p>Range bit 7</p> <p>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.</p>
CHAIN	<p>Chain bit 6</p> <p>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.</p> <p>In the ARM7TDMI-S EmbeddedICE, the CHAINOUT output of Watchpoint 1 is connected to the CHAIN input of Watchpoint 0.</p> <p>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.</p> <p>The CHAINOUT register is cleared when the Control Value Register is written or when nTRST is LOW.</p>
DBGEXT	<p>External Input to the EmbeddedICE 5</p> <p>DBGEXT[1:0] are external input signals to the EmbeddedICE that each allow the watchpoint to be dependent upon some external condition.</p> <p>For the Watchpoint 0 Control Value and Control Mask registers, this field reflects the state of the DGEXT[0] signal. For the Watchpoint 1 Control Value and Control Mask registers, this field reflects the state of the DGEXT[1] signal.</p>
PROT[1]	<p>User/Non-User Mode 4</p> <p>PROT[1] compares 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.</p>
PROT[0]	<p>Instruction Fetch/Data Access 3</p> <p>PROT[0] detects whether the current cycle is an instruction fetch (PROT[0] = 0) or a data access (PROT[0] = 1).</p>

SIZE[1:0] **Size of Bus Activity** **[2:1]**
SIZE[1:0] compares against the SIZE[1:0] signal from the core in order to detect the size of bus activity.

SIZE[1:0]	Data Size
00	byte
01	halfword
10	word
11	(reserved)

WRITE **Write bit** **0**
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.

For each of the bits [7:0] in the Control Value Register, there is a corresponding bit in the Control Mask Register. These bits remove the dependency on particular signals.

B.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 monitor a particular bit pattern being fetched from any address. One EmbeddedICE watchpoint can therefore be used to support any number of software breakpoints.

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

B.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 0b11. For a breakpoint in Thumb state, program bits [1:0] of the Address Mask Register to 0b01.
3. Program only the Data Value Register 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 0xFFFF.FFFF (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.

B.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 0xFFFF.FFFF (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 0xDFFF.DFFF. 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 0x0000.0000.
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. To distinguish 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.

B.12.2.1 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.

B.12.2.2 Clearing the Breakpoint

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

B.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 0x0000.0000.
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 0xFFFF.FFFF (all

bits set to 1). Otherwise program the Data Mask Register to 0x0000.0000.

4. Program the Control Value Register with PROT[0] = 1, WRITE = 0 for a read or WRITE = 1 for a write, and SIZE[1:0] with the value corresponding to the appropriate data size.
5. Program the Control Mask Register with PROT[0] = 0, WRITE = 0, SIZE[1:0] = 0, and all other bits to 1. You can set WRITE or SIZE[1:0] to 1 when both reads and writes or data size accesses are to be watchpointed, respectively.
6. To distinguish 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, setting one or more of the address mask bits can provide simple range breakpoints.

B.14 The Debug Control Register

The Debug Control Register is three bits wide. Control bits are written during a register write access (with the read/write bit HIGH). Control bits are read during a register read access (with the read/write bit LOW).

[Figure B.8](#) shows the function of each bit in this register.

Figure B.8 Debug Control Register Format

2	1	0
INTDIS	DBGRQ	DBGACK

INTDIS

Interrupt Disable

When bit 2 (INTDIS) is set, the interrupt signals to the processor are inhibited. Both IRQ and FIQ are disabled when the processor is in debug state (DBGACK = 1) or when INTDIS is forced.

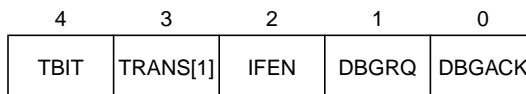
DBGACK	INTDIS	Interrupts
0	0	permitted
1	x	inhibited
x	1	inhibited

- DBGRQ** **Debug Request** **1**
This bit allows the value on DBGRQ to be forced. As shown in [Figure B.10](#), 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.
- DBGACK** **Debug Acknowledge** **0**
This bit allows the value on DBGACK to be forced. 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 core, which 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).

B.15 The Debug Status Register

The Debug Status Register is five 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 B.9](#).

Figure B.9 **Debug Status Register Format**

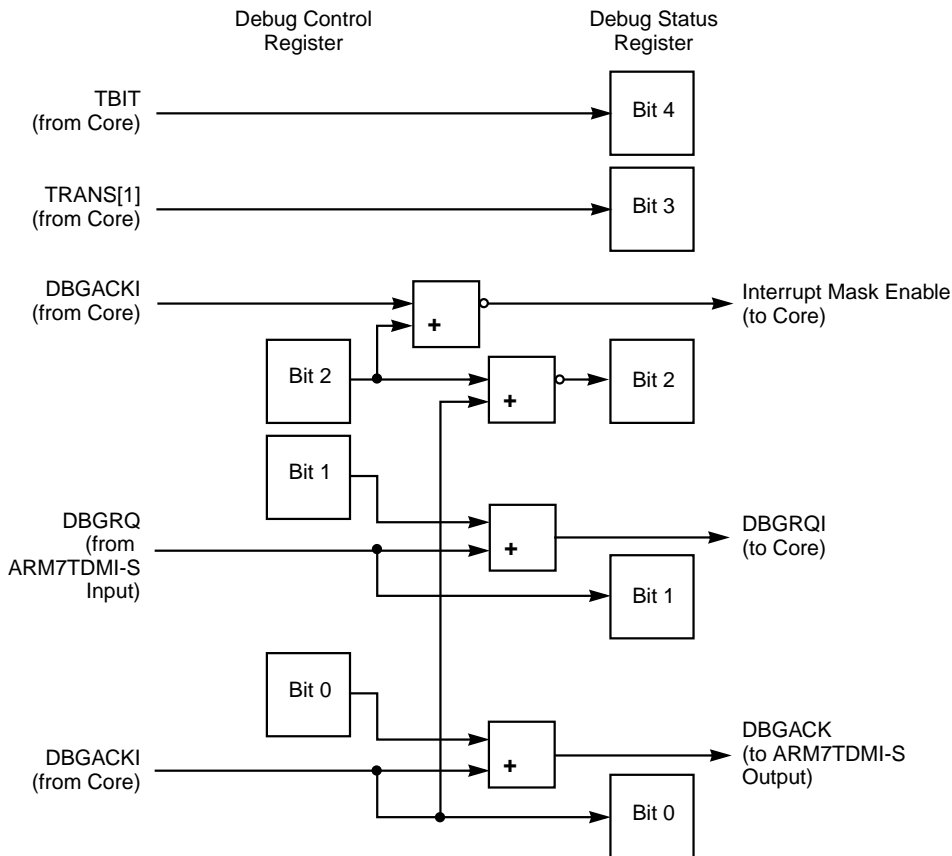


- TBIT** **TBIT Status** **4**
Bit 4 allows TBIT to be read, which allows the debugger to determine the processor state and which instructions to execute.

TRANS[1]	TRANS[1] Signal Status	3
	Bit 3 allows the state of the TRANS[1] signal from the core to be read. This state allows the debugger to determine whether or not a memory access from the debug state has completed.	
IFEN	Interrupt Enable Signal Status	2
	Bit 2 allows the state of the core interrupt enable signal (IFEN) to be read.	
DBGRQ	Debug Request Status	1
	This bit allows the value on the synchronized version of DBGRQ to be read.	
DBGACK	Debug Acknowledge Status	0
	This bit allows the value on the synchronized version of DBGACK to be read.	

Figure B.10 shows the structure of the debug control and status registers.

Figure B.10 Debug Control and Status Register Structure



B.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; it combines the outputs of both watchpoints.

B.16.1 Breakpoint and Watchpoint Coupling Example

This subsection provides an example of coupling the breakpoints and watchpoints together. In this example:

- Av[31:0] is the value in the Address Value Register
- Am[31:0] is the value in the Address Mask Register
- A[31:0] is the address bus from the ARM7TDMI S
- Dv[31:0] is the value in the Data Value Register
- Dm[31:0] is the value in the Data Mask Register
- D[31:0] is the data bus from the ARM7TDMI S
- Cv[8:0] is the value in the Control Value Register
- Cm[7:0] is the value in the Control Mask Register
- C[9:0] is the combined control bus from the ARM7TDMI-S core, other watchpoint registers, and the DBGEXT signal.

The CHAINOUT signal is derived as follows:

```
WHEN (({Av[31:0],Cv[4:0]} XNOR {A[31:0],C[4:0]}) OR {Am[31:0],Cm[4:0]}) == 0xFFFFFFFF)
CHAINOUT = ((({Dv[31:0],Cv[6:4]} XNOR {D[31:0],C[7:5]}) OR
{Dm[31:0],Cm[7:5]}) == 0x7FFFFFFFFF)
```

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 drives 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. When the CHAIN input is asserted, the breakpoint address matches, and the breakpoint triggers correctly.

B.16.2 DBGRNG Signal

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. The DBGRNG signal is derived as follows:

$$\begin{aligned} \text{DBGRNG} = & (((\{A \vee [31:0], C \vee [4:0]\} \text{ XNOR} \\ & \{A[31:0], C[4:0]\}) \text{ OR} \\ & \{A \text{ m } [31:0], C \text{ m } [4:0]\}) == 0\text{xFFFFFFFF}) \text{ AND} \\ & (((\{D \vee [31:0], C \vee [7:5]\} \text{ XNOR } \{D[31:0], C[7:5]\}) \text{ OR} \\ & D \text{ m } [31:0], C \text{ m } [7:5]\}) == 0\text{x7FFFFFFFF}) \end{aligned}$$

Selectable ranges are restricted to 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 0x0000.0000 and an address mask of 0x0000.001F.
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 0x0000.0000 and an address mask of 0x0000.00FF.
2. Set the ENABLE bit.
3. Program the RANGE bit to match a 0.
4. Program all other Watchpoint 0 registers as normal for a breakpoint.

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

B.17 EmbeddedICE Issues

To disable EmbeddedICE, tie the DBGGEN input LOW. When DBGGEN is LOW:

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

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