

ARM7TDMI Microprocessor Core

Technical Manual

November 1998



Order Number C14060

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 DB14-000058-02, First Edition (November 1998)

This document describes revision A of LSI Logic Corporation's ARM7 TDMI Microprocessor and will remain the official reference source for all revisions of this product until rescinded by an update.

To receive product literature, call us at 1.800.574.4286 (U.S. and Canada); +32.11.300.531 (Europe); 408.433.7700 (outside U.S., Canada, and Europe) and ask for Department JDS; or 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 © 1996–1998 by LSI Logic Corporation. All rights reserved.

TRADEMARK ACKNOWLEDGMENT

The LSI Logic logo design, CoreWare, Gigablaze, and G10 are registered trademarks and FlexStream, and Right-First-Time are trademarks of LSI Logic Corporation. ARM is a registered trademark, the ARM Powered logo, and ICEBreaker are trademarks of Advanced RISC Machines Ltd., used under license. All other brand and product names may be trademarks of their respective companies.

Contents

Preface

Chapter 1	Introduction	
1.1	Introduction	1-1
	1.1.1 General Information	1-1
	1.1.2 LSI Logic's ARM7TDMI Implementation	1-2
1.2	ARM7TDMI Architecture	1-2
	1.2.1 The THUMB Concept	1-3
	1.2.2 THUMB Advantages	1-3
1.3	CoreWare [®] Program	1-6

Chapter 2	Signal Descriptions	
2.1	Core Logic Diagram	2-1
2.2	Clock Signals	2-3
2.3	Interrupt Signals	2-3
2.4	Bus Control Interface	2-4
2.5	Debug Interface	2-7
2.6	Boundary Scan Control Interface	2-9
2.7	Boundary Scan Interface	2-11
2.8	Processor Interface	2-12
2.9	Memory Interface	2-12
2.10	Coprocessor Interface	2-15
2.11	Test Signals	2-16

Chapter 3**Programmer's Model**

3.1	Processor Operating States	3-1
3.2	Switching State	3-2
3.2.1	Entering THUMB State	3-2
3.2.2	Entering ARM State	3-2
3.3	Memory Formats	3-2
3.3.1	Big Endian Format	3-2
3.3.2	Little Endian Format	3-3
3.4	Instruction Length	3-3
3.5	Data Types	3-4
3.6	Operating Modes	3-4
3.7	Registers	3-4
3.7.1	The ARM State Register Set	3-5
3.7.2	The THUMB State Register Set	3-6
3.7.3	The Relationship Between ARM and THUMB State Registers	3-7
3.7.4	Accessing High Registers in THUMB State	3-8
3.8	Program Status Registers	3-9
3.8.1	The Condition Code Flags	3-9
3.8.2	Reserved Bits	3-10
3.8.3	The Control Bits	3-10
3.9	Exceptions	3-11
3.9.1	Action on Entering an Exception	3-11
3.9.2	Action on Leaving an Exception	3-12
3.9.3	Exception Entry/Exit Summary	3-13
3.9.4	Fast Interrupt Request (FIQ)	3-13
3.9.5	Interrupt Request (IRQ)	3-14
3.9.6	Abort	3-14
3.9.7	Software Interrupt (SWI)	3-15
3.9.8	Undefined Instruction (UDEF)	3-16
3.9.9	Exception Vectors	3-16
3.9.10	Exception Priorities	3-16
3.10	Interrupt Latencies	3-17
3.11	Reset	3-18
3.12	Pipeline Architecture	3-18

Chapter 4	ARM Instruction Set Summary	
	4.1	Instruction Set Summary 4-1
	4.2	Format Summary 4-3
	4.3	Instruction Condition Field 4-4
	4.4	Instruction Set Examples 4-5
	4.4.1	Using the Conditional Instructions 4-6
	4.4.2	Pseudo-Random Binary Sequence Generator 4-8
	4.4.3	Multiplication by Constant Using the Barrel Shifter 4-8
	4.4.4	Loading a Word from an Unknown Alignment 4-10

Chapter 5	THUMB Instruction Set Summary	
	5.1	Instruction Set Summary 5-1
	5.1.1	Instruction Cycle Time 5-3
	5.2	Format Summary 5-3
	5.3	Instruction Set Examples 5-4
	5.3.1	Multiplication by a Constant Using Shifts and Adds 5-4
	5.3.2	General Purpose Signed Divide 5-5
	5.3.3	Division by a Constant 5-8

Chapter 6	Memory Interface	
	6.1	Overview 6-1
	6.2	Cycle Types 6-2
	6.3	Address Timing 6-4
	6.4	Data Transfer Size 6-7
	6.5	Instruction Fetch 6-8
	6.6	Memory Management 6-10
	6.7	Locked Operations 6-10
	6.8	Stretching Access Times 6-11
	6.9	ARM7TDMI Data Bus 6-11
	6.10	External Data Bus 6-13
	6.10.1	The Unidirectional Data Bus 6-14
	6.10.2	Bidirectional Data Bus 6-15
	6.10.3	Example System: The ARM7TDMI Test Chip 6-18

Chapter 7	Coprocessor Interface	
7.1	Overview	7-1
7.2	Interface Signals	7-1
7.2.1	Coprocessor Present/Absent	7-2
7.2.2	Busy (Waiting)	7-2
7.2.3	Pipeline Following	7-3
7.2.4	Data Transfer Cycles	7-3
7.3	Register Transfer Cycle	7-3
7.4	Privileged Instructions	7-4
7.5	Idempotency	7-4
7.6	Undefined Instructions	7-5

Chapter 8	Debug Interface	
8.1	Overview	8-1
8.2	Debug Systems	8-2
8.3	Debug Interface Signals	8-4
8.3.1	Entry into Debug State	8-4
8.4	Scan Chains and JTAG Interface	8-7
8.4.1	Scan Limitations	8-8
8.4.2	The JTAG State Machine	8-9
8.5	Reset	8-11
8.6	Pull-up Resistors	8-11
8.7	Instruction Register	8-11
8.8	Public Instructions	8-12
8.8.1	EXTEST (0b0000)	8-12
8.8.2	SCAN_N (0b0010)	8-13
8.8.3	INTEST (0b1100)	8-13
8.8.4	IDCODE (0b1110)	8-14
8.8.5	BYPASS (0b1111)	8-14
8.8.6	CLAMP (0b0101)	8-14
8.8.7	HIGHZ (0b0111)	8-15
8.8.8	CLAMPZ (0b1001)	8-15
8.8.9	SAMPLE/PRELOAD (0b0011)	8-16
8.8.10	RESTART (0b0100)	8-16

8.9	Test Data Registers	8-16
8.9.1	Bypass Register	8-16
8.9.2	ARM7TDMI Device Identification (ID) Code Register	8-16
8.9.3	Instruction Register	8-17
8.9.4	Scan Chain Select Register	8-18
8.9.5	Scan Chains 0, 1, and 2	8-19
8.10	ARM7TDMI Core Clocks	8-24
8.10.1	Clock Switch During Debug	8-24
8.11	Determining the Core and System State	8-25
8.11.1	Determining the Core's State	8-25
8.11.2	Determining System State	8-27
8.11.3	Exit from Debug State	8-28
8.12	PC Behavior During Debug	8-30
8.12.1	Breakpoint	8-30
8.12.2	Watchpoints	8-30
8.12.3	Watchpoint with Another Exception	8-31
8.12.4	Debug Request	8-31
8.12.5	System Speed Access	8-32
8.12.6	Summary of Return Address Calculations	8-32
8.13	Priorities/Exceptions	8-33
8.13.1	Breakpoint with Prefetch Abort	8-33
8.13.2	Interrupts	8-33
8.13.3	Data Aborts	8-34
8.14	Scan Interface Timing	8-34
8.15	Debug Timing	8-38

Chapter 9

EmbeddedICE Macrocell

9.1	Overview	9-1
9.2	Watchpoint Registers	9-3
9.2.1	Programming and Reading Watchpoint Registers	9-4
9.2.2	Using the Mask Registers	9-5
9.2.3	Control Registers	9-6
9.3	Programming Breakpoints	9-8
9.3.1	Hardware Breakpoints	9-8
9.3.2	Software Breakpoints	9-9
9.4	Programming Watchpoints	9-10

9.5	Debug Control Register	9-11
9.6	Debug Status Register	9-12
9.7	Coupling Breakpoints and Watchpoints	9-14
9.7.1	CHAINOUT Signal	9-15
9.7.2	RANGEOUT Signal	9-16
9.8	Disabling EmbeddedICE Macrocell	9-17
9.9	EmbeddedICE Macrocell Timing	9-17
9.10	Programming Restriction	9-17
9.11	Debug Communication Channel	9-18
9.11.1	Debug Communications Control Registers	9-18
9.11.2	Communication Through the Communications Channel	9-19

Chapter 10

Instruction Cycle Operations

10.1	Introduction	10-2
10.2	Branch and Branch with Link	10-2
10.3	THUMB Branch with Link	10-3
10.4	Branch and Exchange (BX)	10-4
10.5	Data Operations	10-5
10.6	Multiply and Multiply Accumulate	10-7
10.7	Load Register	10-9
10.8	Store Register	10-10
10.9	Load Multiple Registers	10-10
10.10	Store Multiple Registers	10-12
10.11	Data Swap	10-13
10.12	Software Interrupt and Exception Entry	10-14
10.13	Coprocessor Data Operation	10-15
10.14	Coprocessor Data Transfer (Memory to Coprocessor)	10-16
10.15	Coprocessor Data Transfer (from Coprocessor to Memory)	10-18
10.16	Coprocessor Register Transfer (Load from Coprocessor)	10-20
10.17	Coprocessor Register Transfer (Store to Coprocessor)	10-21
10.18	Undefined Instructions and Coprocessor Absent	10-22
10.19	Unexecuted Instructions	10-23
10.20	Instruction Speed Summary	10-23

Chapter 11	Production Test	
11.1	Core Testing Strategy Overview	11-1
11.2	Scan Test Pin Definitions	11-2
11.3	Full-Scan Production Testing	11-2
11.3.1	Register File Testing	11-3

Chapter 12	Specifications
-------------------	-----------------------

Appendix A	ARM7TDMI Changes
-------------------	-------------------------

Customer Feedback

Figures

1.1	Processor Core Diagram	1-4
1.2	ARM7TDMI Core Diagram	1-5
2.1	ARM7TDMI Logic Diagram	2-2
3.1	Big Endian Addresses of Bytes Within Words	3-3
3.2	Little Endian Addresses of Bytes Within Words	3-3
3.3	Register Organization in ARM State	3-6
3.4	Register Organization in THUMB State	3-7
3.5	Mapping of THUMB State Registers onto ARM State Registers	3-8
3.6	Program Status Register Format	3-9
3.7	ARM7TDMI Pipeline	3-18
3.8	Pipeline Best Case Example	3-19
3.9	Pipeline Branch Example	3-20
3.10	Pipeline Interrupt Example	3-21
3.11	Pipeline Data Memory Access Example	3-22
4.1	ARM Instruction Set Formats	4-3
5.1	THUMB Instruction Set Formats	5-4
6.1	ARM Memory Cycle Timing	6-3
6.2	Memory Cycle Optimization	6-4
6.3	ARM7TDMI Depipelined Addresses	6-5
6.4	ARM7TDMI Pipelined Addresses	6-5
6.5	Typical System Timing	6-6

6.6	SRAM Compatible Address Timing	6-7
6.7	Decoding Byte Accesses to Memory	6-9
6.8	Memory Access	6-12
6.9	Two Cycle Memory Access	6-13
6.10	ARM7TDMI External Bus Arrangement	6-13
6.11	Bidirectional Bus Timing	6-14
6.12	Unidirectional Bus Timing	6-14
6.13	External Connection of Unidirectional Buses	6-15
6.14	Data Write Bus Cycle	6-16
6.15	ARM7TDMI Data Bus Control Circuit	6-18
6.16	The ARM7TDMI Test Chip Data Bus Circuit	6-19
6.17	Data Bus Control Signal Timing	6-20
8.1	Typical Debug System	8-3
8.2	Debug State Entry	8-5
8.3	ARM7TDMI Scan Chain Arrangement	8-9
8.4	Test Access Port (TAP) Controller State Transitions	8-10
8.5	ID Register Format	8-17
8.6	Input Scan Cell	8-20
8.7	Clock Switching on Entry to Debug State	8-24
8.8	Debug Exit Sequence	8-29
8.9	Scan General Timing	8-34
9.1	EmbeddedICE Block Diagram	9-2
9.2	EmbeddedICE Macrocell Block Diagram	9-5
9.3	Watchpoint Control Value and Mask Format	9-6
9.4	Debug Control Register Format	9-11
9.5	Debug Status Register Format	9-12
9.6	Structure of TBIT, nMREQ, DBGACK, DBGSRQ and INTDIS Bits	9-14
9.7	Debug Communications Control Register	9-18
11.1	Register File Testing Scan Path	11-3

Tables

3.1	Mode Bit States	3-11
3.2	Exception Entry/Exit	3-13
3.3	Exception Vectors	3-16
4.1	ARM Instruction Set	4-1
4.2	Condition Code Summary	4-5
5.1	THUMB Instruction Set	5-1
6.1	Memory Cycle Types	6-3
6.2	Endian Configuration Effect on Instruction Position	6-9
6.3	Output Enable Control Summary	6-17
8.1	Public Instructions	8-12
8.2	Scan Chain Number Allocation	8-19
8.3	ARM7TDMI Scan Interface Timing	8-35
8.4	Scan Chain 0 Signal Order	8-36
8.5	ARM7TDMI Debug Interface Timing	8-38
9.1	Function and Mapping of EmbeddedICE Registers	9-3
9.2	IFEN Signal Control	9-12
10.1	Branch Instruction Cycle Operations	10-3
10.2	THUMB Long Branch with Link	10-4
10.3	Branch and Exchange Instruction Cycle Operations	10-5
10.4	Data Operation Instruction Cycle Operations	10-6
10.5	Multiply Instruction Cycle Operations	10-7
10.6	Multiply Accumulate Instruction Cycle Operations	10-7
10.7	Multiply Long Instruction Cycle Operation	10-8
10.8	Multiply Accumulate Long Instruction Cycle Operation	10-8
10.9	Load Register Instruction Cycle Operations	10-9
10.10	Store Register Instruction Cycle Operations	10-10
10.11	Load Multiple Registers Instruction Cycle Operations	10-11
10.12	Store Multiple Registers Instruction Cycle Operations	10-12
10.13	Data Swap Instruction Cycle Operations	10-14
10.14	Software Interrupt Instruction Cycle Operations	10-15
10.15	Coprocessor Data Operation Instruction Cycle Operations	10-16
10.16	Coprocessor Data Transfer Instruction Cycle Operations	10-17

10.17	Coprocessor Data Transfer Instruction Cycle Operations	10-19
10.18	Coprocessor Register Transfer (Load from Coprocessor)	10-21
10.19	Coprocessor Register Transfer (Store to Coprocessor)	10-22
10.20	Undefined Instruction Cycle Operations	10-23
10.21	Unexecuted Instruction Cycle Operations	10-23
10.22	ARM Instruction Speed Summary	10-24
11.1	Scan Test Pins	11-2

Preface

This book is the primary reference and technical manual for the ARM7TDMI microprocessor core and contains a complete functional description for the core. The information in this manual applies to all process revisions of the core. Specific technology-dependent values, such as electrical timing, can be found in the appropriate *ARM7TDMI Microprocessor Core Datasheet*, which is available from LSI Logic.

Audience

This document was prepared for logic designers and applications engineers and is intended to provide an overview of LSI Logic's FlexStream™ system and to explain how to use the FlexStream software in the initial stages of chip design.

This document assumes that you have some familiarity with microprocessors and related support devices. The people who benefit the most 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](#)
- [Chapter 2, Signal Descriptions](#)
- [Chapter 3, Programmer's Model](#)
- [Chapter 4, ARM Instruction Set Summary](#)

- Chapter 5, **THUMB Instruction Set Summary**
- Chapter 6, **Memory Interface**
- Chapter 7, **Coprocessor Interface**
- Chapter 8, **Debug Interface**
- Chapter 9, **EmbeddedICE Macrocell**
- Chapter 10, **Instruction Cycle Operations**
- Chapter 11, **Production Test**
- Chapter 12, **Specifications**, see *CW001007 ARM7TDMI Microprocessor Core Datasheet*
- Appendix A, **ARM7TDMI Changes**

Related Publications

ARM7TDMI Data Sheet, available from Advanced RISC Machines Ltd. as document No. ARM DDI 0029E

CW001004 ARM7TDMI Microprocessor Core Datasheet, available from LSI Logic.

CW001007 ARM7TDMI Microprocessor Core Datasheet, available from LSI Logic.

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

ARM Architectural Reference Manual, Advanced RISC Machines Ltd and Prentice-Hall.

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.

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.

Signal names are shown in capital letters.

Active LOW signals are indicated by the prefix “n” —for example, nRESET.

The manual refers to a 32-bit quantity as a *word*, a 16-bit value as a *halfword*, and an 8-bit quantity as a *byte*.

Document Version	Release Date	Comments
Advance	December 1996	Initial release.
Preliminary	January 1998	This document was derived from the ARM document <i>ARM7TDMI Data Sheet</i> . Appendix A, ARM7TDMI Changes contains a list of the differences between LSI Logic and ARM's documents.
Final	November 1998	

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,” page 1-1](#)
 - [Section 1.2, “ARM7TDMI Architecture,” page 1-2](#)
 - [Section 1.3, “CoreWare® Program,” page 1-6](#)
-

1.1 Introduction

This section introduces the overall core capabilities and highlights the beneficial features of LSI Logic’s ARM7TDMI core implementation.

1.1.1 General Information

The ARM7TDMI 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 Implementation

The ARM7TDMI Microprocessor Core described in this manual is LSI Logic's proprietary version of the ARM7TDMI microcontroller. LSI Logic's implementation represents the world's first synthesizable version of the ARM7TDMI. LSI Logic has further 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 ARM7TDMI RTL (register-transfer level) version was developed in close conjunction with ARM, Ltd. ensuring 100% compatibility with the ARM7TDMI specification. LSI Logic's core has identical functionality and external interfaces making both the hardware and software 100% compatible with the full custom cores presently available from all other ARM7TDMI licensees. LSI Logic's RTL has been designed with single phase clocking and simplified register schemes wherever possible. This greatly eases synthesis and timing analysis of the surrounding logic and thereby facilitates the design of high-quality products with a minimum time-to-market. The RTL has been synthesized and taken through place, route, and test insertion resulting in a hardmacro ready for a system-on-a-chip design. Full scan test insertion provides high fault coverage while keeping test costs to a minimum. Finally, to implement full scan, LSI Logic has added eight additional test signals to the core (for a full description of these signals, see [Chapter 2, "Signal Descriptions."](#))

1.2 ARM7TDMI Architecture

The ARM7TDMI processor employs a unique architectural strategy known as *THUMB*, which makes it ideally suited to high volume applications with memory restrictions, or applications where code density is an issue.

1.2.1 The THUMB Concept

The key idea behind THUMB is that of a super-reduced instruction set. Essentially, the ARM7TDMI processor has two instruction sets:

- Standard 32-bit ARM set
- A 16-bit THUMB set

The THUMB set's 16-bit instruction length allows it to approach twice the density of standard ARM code while retaining most of the ARM processor's performance advantage over a traditional 16-bit processor using 16-bit registers. This is possible because THUMB code operates on the same 32-bit register set as ARM code.

THUMB code is able to provide up to 65% of the size of ARM code, and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system.

1.2.2 THUMB Advantages

THUMB instructions operate with the standard ARM register configuration, allowing excellent interoperability between ARM and THUMB states. Each 16-bit THUMB instruction has a corresponding 32-bit ARM instruction with the same effect on the processor model.

The major advantage of a 32-bit (ARM) architecture over a 16-bit architecture is its ability to manipulate 32-bit integers with single instructions, and to address a large address space efficiently. When processing 32-bit data, a 16-bit architecture will take at least two instructions to perform the same task as a single ARM instruction.

However, not all the code in a program will process 32-bit data (for example, code that performs character string handling), and some instructions, like Branches, do not process any data at all.

If a 16-bit architecture only has 16-bit instructions, and a 32-bit architecture only has 32-bit instructions, then overall the 16-bit architecture will have better code density, and better than one half the performance of the 32-bit architecture. Clearly 32-bit performance comes at the cost of code density.

THUMB breaks this constraint by implementing a 16-bit instruction length on a 32-bit architecture, making the processing of 32-bit data efficient with a compact instruction coding. This provides far better performance than a 16-bit architecture, with better code density than a 32-bit architecture.

THUMB also has a major advantage over other 32-bit architectures with 16-bit instructions. This is the ability to switch back to full ARM code and execute at full speed. Thus critical loops for applications such as

- Fast interrupts
- DSP algorithms

can be coded using the full ARM instruction set, and linked with THUMB code. The overhead of switching from THUMB code to ARM code is folded into subroutine entry time. Various portions of a system can be optimized for speed or for code density by switching between THUMB and ARM execution as appropriate.

Figure 1.1 Processor Core Diagram

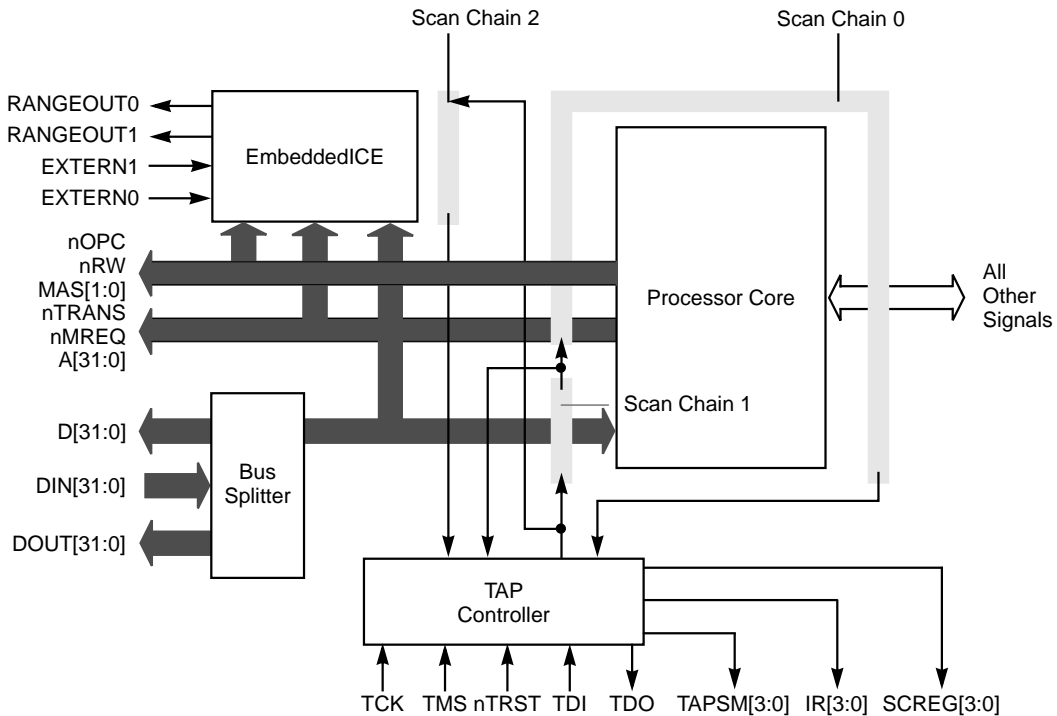
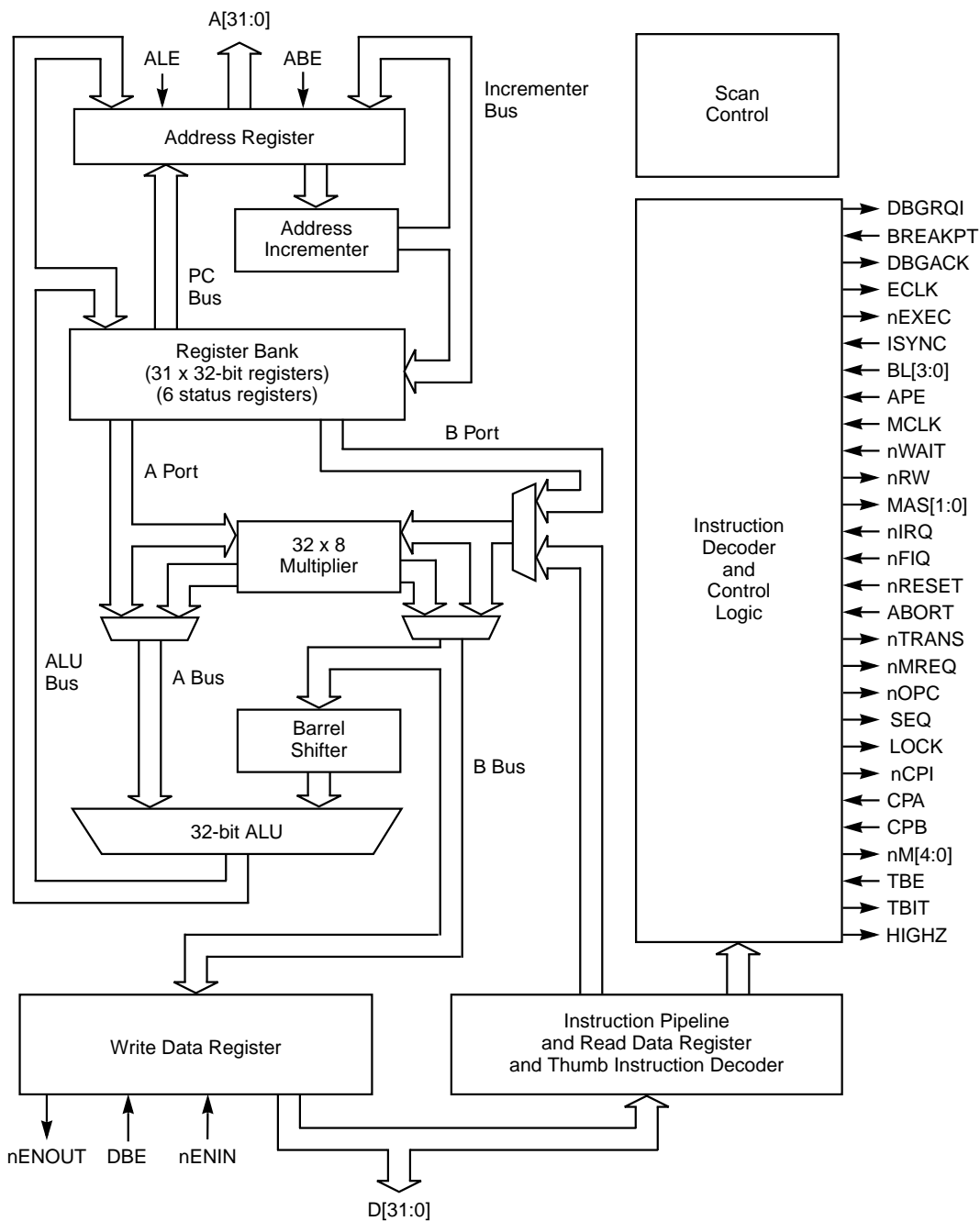


Figure 1.2 ARM7TDMI Core Diagram



1.3 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. The 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:

- Encrypted 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 FlexStream™ design support

LSI Logic's FlexStream design solution 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 Logic's FlexStream design 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

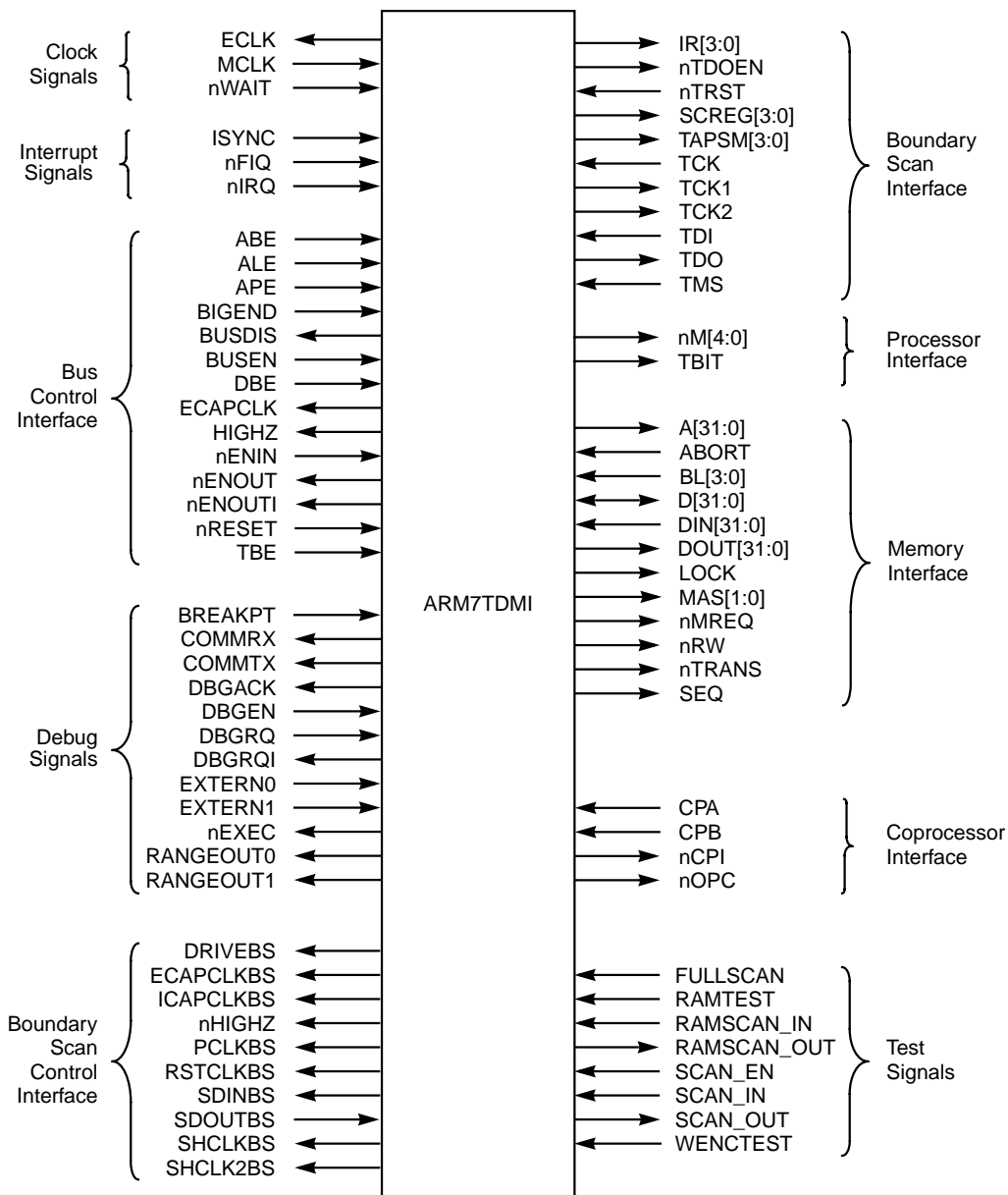
This chapter describes all ARM7TDMI core interface signals and is divided into the following sections:

- [Section 2.1, “Core Logic Diagram,” page 2-1](#)
- [Section 2.2, “Clock Signals,” page 2-3](#)
- [Section 2.3, “Interrupt Signals,” page 2-3](#)
- [Section 2.4, “Bus Control Interface,” page 2-4](#)
- [Section 2.5, “Debug Interface,” page 2-7](#)
- [Section 2.6, “Boundary Scan Control Interface,” page 2-9](#)
- [Section 2.7, “Boundary Scan Interface,” page 2-11](#)
- [Section 2.8, “Processor Interface,” page 2-12](#)
- [Section 2.9, “Memory Interface,” page 2-12](#)
- [Section 2.10, “Coprocesor Interface,” page 2-15](#)
- [Section 2.11, “Test Signals,” page 2-16](#)

2.1 Core Logic Diagram

[Figure 2.1](#) is the core logic diagram and lists the core signal interfaces.

Figure 2.1 ARM7TDMI Logic Diagram



2.2 Clock Signals

ECLK	External Clock Output	Output
	In normal operation, this is simply MCLK (optionally stretched with nWAIT) exported from the core. When the core is being debugged, this is TCLK. This allows external hardware to track when the ARM7TDMI core is clocked. When FULLSCAN is asserted, ECLK is held LOW.	
MCLK	Memory Clock Input	Input
	This clock times all ARM7TDMI memory accesses and internal operations. The clock has two distinct phases	
	<ul style="list-style-type: none">• phase 1 in which MCLK is LOW• phase 2 in which MCLK (and nWAIT) are HIGH	
	The clock may be stretched indefinitely in either phase to allow access to slow peripherals or memory. Alternatively, the nWAIT input may be used with a free running MCLK to achieve the same effect.	
nWAIT	Not Wait	Input
	When accessing slow peripherals, ARM7TDMI can be made to wait for an integer number of MCLK cycles by driving nWAIT LOW. Internally, nWAIT is ANDed with MCLK and must only change when MCLK is LOW. If nWAIT is not used it must be tied HIGH.	

2.3 Interrupt Signals

ISYNC	Synchronous Interrupts	Input
	When LOW indicates that the nIRQ and nFIQ inputs are to be synchronized by the ARM7TDMI core. When HIGH disables this synchronization for inputs that are already synchronous.	
nFIQ	Not Fast Interrupt Request	Input
	This is an interrupt request to the processor which causes it to be interrupted if taken LOW when the appropriate enable in the processor is active. The signal is level-sensitive and must be held LOW until a suitable	

response is received from the processor. nFIQ may be synchronous or asynchronous, depending on the state of ISYNC.

nIRQ	Not Interrupt Request	Input
	The same as nFIQ, but with lower priority. May be taken LOW to interrupt the processor when the appropriate enable is active. nIRQ may be synchronous or asynchronous, depending on the state of ISYNC.	

2.4 Bus Control Interface

ABE	Address Bus Enable	Input
	This is an input signal which, when LOW, puts the address bus drivers into a high impedance state. This signal has a similar effect on the following control signals: MAS[1:0], nRW, LOCK, nOPC and nTRANS. ABE must be tied HIGH when there is no system requirement to turn off the address drivers.	

ALE	Address Latch Enable	Input
	This input is used to control transparent latches on the address outputs. Normally the addresses change during phase 2 to the value required during the next cycle, but for direct interfacing to ROMs they are required to be stable to the end of phase 2. Taking ALE LOW until the end of phase 2 will ensure that this happens. This signal has a similar effect on the following control signals: MAS[1:0], nRW, LOCK, nOPC and nTRANS. If the system does not require address lines to be held in this way, ALE must be tied HIGH. The address latch is static, so ALE may be held LOW for long periods to freeze addresses.	

APE	Address Pipeline Enable	Input
	When HIGH, this signal enables the address timing pipeline. In this state, the address bus plus MAS[1:0], nRW, nTRANS, LOCK and nOPC change in phase 2 prior to the memory cycle to which they refer. When APE is LOW, these signals change in phase 1 of the actual cycle. Please refer to Chapter 6, "Memory Interface" for details of this timing.	

BIGEND	Big Endian Configuration	Input
	When this signal is HIGH the processor treats bytes in memory as being in Big Endian format. When it is LOW, memory is treated as Little Endian.	
BUSDIS	Bus Disable	Output
	This signal is HIGH when <code>INTEST</code> is selected on scan chain 0 or 4 and may be used to disable external logic driving onto the bidirectional data bus during scan testing. This signal changes on the falling edge of TCK.	
BUSEN	Data Bus Configuration	Input
	This is a static configuration signal which determines whether the bidirectional data bus, <code>D[31:0]</code> , or the unidirectional data buses, <code>DIN[31:0]</code> and <code>DOUT[31:0]</code> , are to be used for transfer of data between the processor and memory. Refer also to Chapter 6, "Memory Interface" .	
	When <code>BUSEN</code> is LOW, the bidirectional data bus, <code>D[31:0]</code> is used. In this case, <code>DOUT[31:0]</code> is driven to value <code>0x00000000</code> , and any data presented on <code>DIN[31:0]</code> is ignored.	
	When <code>BUSEN</code> is HIGH, the bidirectional data bus, <code>D[31:0]</code> is ignored and must be left unconnected. Input data and instructions are presented on the input data bus, <code>DIN[31:0]</code> , output data appears on <code>DOUT[31:0]</code> .	
DBE	Data Bus Enable	Input
	This is an input signal which, when driven LOW, puts the data bus <code>D[31:0]</code> into the high impedance state. This is included for test purposes, and should be tied HIGH at all times.	
ECAPCLK	Extest Capture Clock	Output
	This signal removes the need for the external logic in the test chip which was required to enable the internal 3-state bus during scan testing. This need not be brought out as an external pin on the test chip.	
HIGHZ		Output
	This signal denotes that the <code>HIGHZ</code> instruction has been loaded into the TAP controller. See Chapter 8, "Debug Interface" for details.	

nENIN	Not Enable Input	Input
	This signal may be used in conjunction with nENOUT to control the data bus during write cycles. See Chapter 6, "Memory Interface" .	
nENOUT	Not Enable Output	Output
	During a data write cycle, this signal is driven LOW during phase 1, and remains LOW for the entire cycle. This may be used to aid arbitration in shared bus applications. See Chapter 6, "Memory Interface" .	
nENOUTI	Not Enable Output ICE	Output
	During a coprocessor register transfer C-cycle from the EmbeddedICE macrocell communications channel coprocessor to the ARM7TDMI core, this signal goes LOW during phase 1 and stays LOW for the entire cycle. This may be used to aid arbitration in shared bus systems.	
nRESET	Not Reset	Input
	This is a level sensitive input signal which is used to start the processor from a known address. A LOW level will cause the instruction being executed to terminate abnormally. When nRESET becomes HIGH for at least one clock cycle, the processor will restart from address 0. nRESET must remain LOW (and nWAIT must remain HIGH) for at least two clock cycles. During the LOW period the processor will perform dummy instruction fetches with the address incrementing from the point where reset was activated. The address will overflow to zero if nRESET is held beyond the maximum address limit.	
TBE	Test Bus Enable	Input
	When driven LOW, TBE forces the data bus D[31:0], the Address bus A[31:0], plus LOCK, MAS[1:0], nRW, nTRANS and nOPC to high impedance. This is as if both ABE and DBE had both been driven LOW. However, TBE does not have an associated scan cell and so allows external signals to be driven high impedance during scan testing. Under normal operating conditions, TBE should be held HIGH at all times.	

2.5 Debug Interface

BREAKPT	Breakpoint	Input
	This signal allows external hardware to halt the execution of the processor for debug purposes. When HIGH causes the current memory access to be a breakpoint. If the memory access is an instruction fetch, ARM7TDMI will enter debug state if the instruction reaches the execute stage of the ARM7TDMI pipeline. If the memory access is for data, ARM7TDMI will enter debug state after the current instruction completes execution. This allows extension of the internal breakpoints provided by the EmbeddedICE macrocell module. See Chapter 6, "Memory Interface" .	
COMMRX	Communications Channel Receive	Output
	When HIGH, this signal denotes that the communications channel receive buffer is empty. This signal changes on the rising edge of MCLK. See Section 9.11.1, "Debug Communications Control Registers," for more information on the debug communications channel.	
COMMTX	Communications Channel Transmit	Output
	When HIGH, this signal denotes that the communications channel transmit buffer is empty. This signal changes on the rising edge of MCLK. See Section 9.11.1, "Debug Communications Control Registers," for more information on the debug communications channel.	
DBGACK	Debug Acknowledge	Output
	When HIGH indicates ARM7TDMI is in debug state.	
DBGEN	Debug Enable	Input
	This input signal allows the debug features of ARM7TDMI to be disabled. This signal should be driven LOW when debugging is not required.	
DBGRRQ	Debug Request	Input
	This is a level sensitive input, which when HIGH causes ARM7TDMI to enter debug state after executing the current instruction. This allows external hardware to force ARM7TDMI into the debug state, in addition to the debugging features provided by the EmbeddedICE macrocell. See Chapter 9, "EmbeddedICE Macrocell" for details.	

DBGRQI	Internal Debug Request	Output
	<p>This signal represents the debug request signal which is presented to the processor. This is the combination of external DBGRQ, as presented to the ARM7TDMI macrocell, and bit 1 of the debug control register. Thus there are two conditions where this signal can change.</p> <ul style="list-style-type: none"> • When DBGRQ changes, DBGRQI will change after a propagation delay. • When bit 1 of the debug control register has been written, this signal will change on the falling edge of TCK when the TAP controller state machine is in the Run-Test/Idle state. <p>See Chapter 9, "EmbeddedICE Macrocell" for details.</p>	
EXTERN0	External Input 0	Input
	<p>This is an input to the EmbeddedICE logic in the ARM7TDMI which allows breakpoints and/or watchpoints to be dependent on an external condition.</p>	
EXTERN1	External Input 1	Input
	<p>This is an input to the EmbeddedICE logic in the ARM7TDMI which allows breakpoints and/or watchpoints to be dependent on an external condition.</p>	
nEXEC	Not Executed	Output
	<p>When HIGH indicates that the instruction in the execution unit is not being executed, because, for example, it has failed its condition code check.</p>	
RANGEOUT0	EmbeddedICE Rangeout0	Output
	<p>This signal indicates that EmbeddedICE watchpoint register 0 has matched the conditions currently present on the address, data and control buses. This signal is independent of the state of the watchpoint's enable control bit. RANGEOUT0 changes when ECLK is LOW.</p>	
RANGEOUT1	EmbeddedICE Rangeout1	Output
	<p>This signal is the same as RANGEOUT0 but corresponds to EmbeddedICE watchpoint register 1.</p>	

2.6 Boundary Scan Control Interface

DRIVEBS	Boundary Scan Cell Enable	Output
	This signal is used to control the multiplexers in the scan cells of an external boundary scan chain. This signal changes in the Update-IR state when scan chain 3 is selected and either the <code>INTEST</code> , <code>EXTEST</code> , <code>CLAMP</code> or <code>CLAMPZ</code> instruction is loaded. When an external boundary scan chain is not connected, this output should be left unconnected.	
ECAPCLKBS	Extest Capture Clock for Boundary Scan	Output
	This is a TCK2 wide pulse generated when the TAP controller state machine is in the Capture-DR state, the current instruction is <code>EXTEST</code> and scan chain 3 is selected. This is used to capture the macrocell outputs during <code>EXTEST</code> . When an external boundary scan chain is not connected, this output should be left unconnected.	
ICAPCLKBS	Intest Capture Clock	Output
	This is a TCK2 wide pulse generated when the TAP controller state machine is in the Capture-DR state, the current instruction is <code>INTEST</code> and scan chain 3 is selected. This is used to capture the macrocell outputs during <code>INTEST</code> . When an external boundary scan chain is not connected, this output should be left unconnected.	
nHIGHZ	Not HIGHZ	Output
	This signal is generated by the TAP controller when the current instruction is <code>HIGHZ</code> . This is used to place the scan cells of that scan chain in the high impedance state. When an external boundary scan chain is not connected, this output should be left unconnected.	
PCLKBS	Boundary Scan Update Clock	Output
	This is a TCK2 wide pulse generated when the TAP controller state machine is in the Update-DR state and scan chain 3 is selected. This is used by an external boundary scan chain as the update clock. When an external boundary scan chain is not connected, this output should be left unconnected.	

RSTCLKBS	Boundary Scan Reset Clock	Output
	This signal denotes that either the TAP controller state machine is in the RESET state or that nTRST has been asserted. This may be used to reset external boundary scan cells.	
SDINBS	Boundary Scan Serial Input Data	Output
	This signal contains the serial data to be applied to an external scan chain and is valid on the falling edge of TCK.	
SDOUTBS	Boundary Scan Serial Output Data	Input
	This control signal is provided to ease the connection of an external boundary scan chain. This is the serial data out of the boundary scan chain. It should be setup to the rising edge of TCK. When an external boundary scan chain is not connected, this input should be tied LOW.	
SHCLKBS	Boundary Scan Shift Clock, Phase 1	Output
	This control signal is provided to ease the connection of an external boundary scan chain. SHCLKBS is used to clock the master half of the external scan cells. When in the Shift-DR state of the state machine and scan chain 3 is selected, SHCLKBS follows TCK1. When not in the Shift-DR state or when scan chain 3 is not selected, this clock is LOW. When an external boundary scan chain is not connected, this output should be left unconnected.	
SHCLK2BS	Boundary Scan Shift Clock, Phase 2	Output
	This control signal is provided to ease the connection of an external boundary scan chain. SHCLK2BS is used to clock the master half of the external scan cells. When in the Shift-DR state of the state machine and scan chain 3 is selected, SHCLK2BS follows TCK2. When not in the Shift-DR state or when scan chain 3 is not selected, this clock is LOW. When an external boundary scan chain is not connected, this output should be left unconnected.	

2.7 Boundary Scan Interface

IR[3:0]	TAP Controller Instruction Register	Output
	These 4 bits reflect the current instruction loaded into the TAP controller instruction register. The instruction encoding is as described in Section 8.8, “Public Instructions” . These bits change on the falling edge of TCK when the state machine is in the Update-IR state.	
nTDOEN	Not TDO Enable	Output
	When LOW, this signal denotes that serial data is being driven out on the TDO output. nTDOEN would normally be used as an output enable for a TDO pin in a packaged part.	
nTRST	Not Test Reset	Input
	Active LOW reset signal for the boundary scan logic. This pin must be pulsed or driven LOW to achieve normal device operation, in addition to the normal device reset (nRESET). For more information, see Chapter 8, “Debug Interface” .	
SCREG[3:0]	Scan Chain Register	Output
	These 4 bits reflect the ID number of the scan chain currently selected by the TAP controller. These bits change on the falling edge of TCK when the TAP state machine is in the Update-DR state.	
TAPSM[3:0]	TAP Controller State Machine	Output
	This bus reflects the current state of the TAP controller state machine, as shown in Section 8.4.2, “The JTAG State Machine” . These bits change off the rising edge of TCK.	
TCK	Test Clock	Input
	The clock used for test operations.	
TCK1	TCK, Phase 1	Output
	This clock is a buffered version of TCK. TCK1 is HIGH when TCK is HIGH.	

TCK2	TCK, Phase 2 This clock is a buffered, inverted version of TCK. TCK2 is HIGH when TCK is LOW. Please note that TCK2 has a slight overlap with the TCK1 signal.	Output
TDI	Test Data Input Boundary scan logic input.	Input
TDO	Test Data Output Boundary scan logic output.	Output
TMS	Test Mode Select Boundary scan test mode select signal.	Input

2.8 Processor Interface

nM[4:0]	Not Processor Mode These are output signals which are the inverses of the internal status bits indicating the processor operation mode.	Output
TBIT	THUMB Mode When HIGH, this signal denotes that the processor is executing the THUMB instruction set. When LOW, the processor is executing the ARM instruction set. This signal changes in phase 2 in the first execute cycle of a BX instruction.	Output

2.9 Memory Interface

A[31:0]	Addresses This is the processor address bus. If ALE (Address Latch Enable) is HIGH and APE (Address Pipeline Enable) is LOW, the addresses become valid during phase 2 of the cycle before the one to which they refer and remain so during phase 1 of the referenced cycle. Their stable period may be controlled by ALE or APE as described in Section 2.4, "Bus Control Interface".	Output
ABORT	Memory Abort This is an input which allows the memory system to tell the processor that a requested access is not allowed.	Input

BL[3:0]	Byte Latch Control	Input
	<p>These signals control when data and instructions are latched from the external data bus. When BL[3] is HIGH, the data on D[31:24] is latched on the falling edge of MCLK. When BL[2] is HIGH, the data on D[23:16] is latched and so on. Please refer to Chapter 6, "Memory Interface" for details on the use of these signals.</p>	
D[31:0]	Data Bus	Bidirectional
	<p>These are bidirectional signal paths which are used for data transfers between the processor and external memory. During read cycles (when nRW is LOW), the input data must be valid before the end of phase 2 of the transfer cycle. During write cycles (when nRW is HIGH), the output data will become valid during phase 1 and remain valid throughout phase 2 of the transfer cycle.</p>	
	<p><u>Note:</u> This bus is driven at all times, irrespective of whether BUSEN is HIGH or LOW. When D[31:0] is not being used to connect to the memory system it must be left unconnected. See Chapter 6, "Memory Interface".</p>	
DIN[31:0]	Data Input Bus	Input
	<p>This is the input data bus which may be used to transfer instructions and data between the processor and memory. This data input bus is only used when BUSEN is HIGH. The data on this bus is sampled by the processor at the end of phase 2 during read cycles (i.e. when nRW is LOW).</p>	
DOUT[31:0]	Data Output Bus	Output
	<p>This is the data out bus, used to transfer data from the processor to the memory system. Output data only appears on this bus when BUSEN is HIGH. At all other times, this bus is driven to value 0x00000000. When in use, data on this bus changes during phase 1 of store cycles (i.e. when nRW is HIGH) and remains valid throughout phase 2.</p>	
LOCK	Locked Operation	Output
	<p>When LOCK is HIGH, the processor is performing a <i>locked</i> memory access, and the memory controller must wait until LOCK goes LOW before allowing another device to access the memory. LOCK changes while MCLK is HIGH, and remains HIGH for the duration of the</p>	

locked memory accesses. It is active only during the data swap (*SWP*) instruction. The timing of this signal may be modified by the use of ALE and APE as described in Section 2.4, “Bus Control Interface.” This signal may also be driven to a high impedance state by driving ABE LOW.

MAS[1:0]	Memory Access Size	Output
	These are output signals used by the processor to indicate to the external memory system when a word transfer or a halfword or byte length is required. The signals take the value 0b10 for words, 0b01 for halfwords and 0b00 for bytes. 0b11 is reserved. These values are valid for both read and write cycles. The signals will normally become valid during phase 2 of the cycle before the one in which the transfer will take place. They will remain stable throughout phase 1 of the transfer cycle. The timing of the signals may be modified by the use of ALE and APE in a way similar to the A[31:0], please refer to Section 2.4, “Bus Control Interface”. The signals may also be driven to high impedance state by driving ABE LOW.	
nMREQ	Not Memory Request	Output
	This signal, when LOW, indicates that the processor requires memory access during the following cycle. The signal becomes valid during phase 1, remaining valid through phase 2 of the cycle preceding that to which it refers.	
nRW	Not Read/Write	Output
	When HIGH this signal indicates a processor write cycle; when LOW, a read cycle. It becomes valid during phase 2 of the cycle before that to which it refers, and remains valid to the end of phase 1 of the referenced cycle. The timing of this signal may be modified by the use of ALE and APE in a way similar to the A[31:0] signals, please refer to Section 2.4, “Bus Control Interface”. This signal may also be driven to a high impedance state by driving ABE LOW.	
nTRANS	Not Memory Translate	Output
	When this signal is LOW it indicates that the processor is in user mode. It may be used to tell memory management hardware when translation of the addresses should be turned on, or as an indicator of nonuser mode	

activity. The timing of this signal may be modified by the use of ALE and APE in a way similar to the A[31:0] signals, please refer to Section 2.4, “Bus Control Interface”. This signal may also be driven to a high impedance state by driving ABE LOW.

SEQ	Sequential Address	Output
	<p>This output signal will become HIGH when the address of the next memory cycle will be related to that of the last memory access. The new address will either be the same as the previous one or four greater in ARM state, or two greater in THUMB state.</p> <p>The signal becomes valid during phase 1 and remains so through phase 2 of the cycle before the cycle whose address it anticipates. It may be used, in combination with the low-order address lines, to indicate that the next cycle can use a fast memory mode (for example DRAM page mode) and/or to bypass the address translation system.</p>	

2.10 Coprocessor Interface

CPA	Coprocessor Absent	Input
	<p>A coprocessor which is capable of performing the operation that ARM7TDMI is requesting (by asserting nCPI) should take CPA LOW immediately. If CPA is HIGH at the end of phase 1 of the cycle in which nCPI went LOW, ARM7TDMI will abort the coprocessor handshake and take the undefined instruction trap. If CPA is LOW and remains LOW, ARM7TDMI will busy-wait until CPB is LOW and then complete the coprocessor instruction.</p>	
CPB	Coprocessor Busy	Input
	<p>A coprocessor which is capable of performing the operation which ARM7TDMI is requesting (by asserting nCPI), but cannot commit to starting it immediately, should indicate this by driving CPB HIGH. When the coprocessor is ready to start it should take CPB LOW. ARM7TDMI samples CPB at the end of phase 1 of each cycle in which nCPI is LOW.</p>	

nCPI	Not Coprocessor Instruction	Output
	When ARM7TDMI executes a coprocessor instruction, it will take this output LOW and wait for a response from the coprocessor. The action taken will depend on this response, which the coprocessor signals on the CPA and CPB inputs.	
nOPC	Not Opcode Fetch	Output
	When LOW this signal indicates that the processor is fetching an instruction from memory; when HIGH, data (if present) is being transferred. The signal becomes valid during phase 2 of the previous cycle, remaining valid through phase 1 of the referenced cycle. The timing of this signal may be modified by the use of ALE and APE in a way similar to the A[31:0] signals, please refer to Section 2.4, "Bus Control Interface". This signal may also be driven to a high impedance state by driving ABE LOW.	

2.11 Test Signals

FULLSCAN	Master Scan Mode Select	Input
	Asserting FULLSCAN enables either Production Test Mode or Ramtest Mode, depending on the value of RAMTEST. FULLSCAN should remain asserted for the duration of scan testing and must be deasserted during normal operation.	
RAMTEST	Ramtest Scan Mode Select	Input
	When FULLSCAN is asserted, asserting RAMTEST places the core in Ramtest mode. RAMTEST should remain asserted for the duration of the ramtest scan. If FULLSCAN is deasserted, asserting RAMTEST will have no effect on the core state.	
RAMSCAN_IN	Ramtest Scan Chain Input	Input
	During Ramtest Mode, RAMSCAN_IN is the scan input for the core memory scan chain.	
RAMSCAN_OUT	Ramtest Scan Chain Output	Output
	During Ramtest Mode, RAMSCAN_OUT is the scan output for the core memory scan chain.	

SCAN_EN	Global Scan Enable	Input
	In either Production Test mode or Ramtest mode, asserting SCAN_EN enables serial loading of the scan registers through the scan chain.	
SCAN_IN	Full Scan Chain Input	Input
	In Production Test mode, SCAN_IN is the scan input for the core scan chain.	
SCAN_OUT	Full Scan Chain Output	Output
	In Production Test mode, SCAN_OUT is the scan output for the core scan chain.	
WENCTEST	Ramtest Write Enable	Input
	This test signal is used only when FULLSCAN is asserted. WENCTEST controls core memory writes in the Ramtest mode. When FULLSCAN is deasserted, WENCTEST should also be deasserted.	

Chapter 3

Programmer's Model

This chapter describes the two operating states of the ARM7TDMI. It contains the following sections:

- [Section 3.1, "Processor Operating States," page 3-1](#)
- [Section 3.2, "Switching State," page 3-2](#)
- [Section 3.3, "Memory Formats," page 3-2](#)
- [Section 3.4, "Instruction Length," page 3-3](#)
- [Section 3.5, "Data Types," page 3-4](#)
- [Section 3.6, "Operating Modes," page 3-4](#)
- [Section 3.7, "Registers," page 3-4](#)
- [Section 3.8, "Program Status Registers," page 3-9](#)
- [Section 3.9, "Exceptions," page 3-11](#)
- [Section 3.10, "Interrupt Latencies," page 3-17](#)
- [Section 3.11, "Reset," page 3-18](#)
- [Section 3.12, "Pipeline Architecture," page 3-18](#)

3.1 Processor Operating States

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

ARM state which executes 32-bit, word-aligned ARM instructions.

THUMB state which 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.

3.2 Switching State

This section describes the method for entering either the THUMB or ARM state.

3.2.1 Entering THUMB State

Entry into THUMB state can be achieved by executing a BX instruction with the state bit (bit 0) set in the operand register.

Transition to THUMB state will also occur automatically on return from an exception (IRQ, FIQ, UNDEF, ABORT, SWI etc.), if the exception was entered with the processor in THUMB state.

3.2.2 Entering ARM State

Entry into ARM state happens:

1. On execution of the BX instruction with the state bit clear in the operand register.
2. On the processor taking an exception (IRQ, FIQ, RESET, UNDEF, ABORT, SWI etc.). In this case, the PC is placed in the exception mode's link register, and execution commences at the exception's vector address.

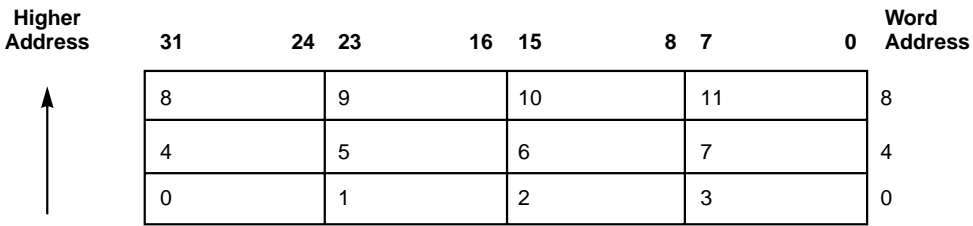
3.3 Memory Formats

ARM7TDMI 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. ARM7TDMI can treat words in memory as being stored either in *Big Endian* or *Little Endian* format.

3.3.1 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 at the highest numbered byte. Byte 0 of the memory system is therefore connected to data lines 31 through 24.

Figure 3.1 Big Endian Addresses of Bytes Within Words



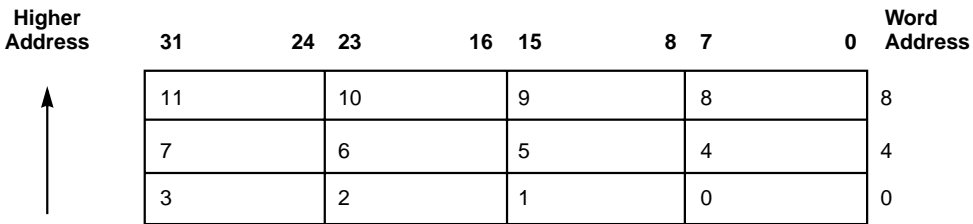
Note:

- ◆ Most significant byte is at lowest address.
- ◆ Word is addressed by byte address of most significant byte.

3.3.2 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 is therefore connected to data lines 7 through 0.

Figure 3.2 Little Endian Addresses of Bytes Within Words



Note:

- ◆ Most significant byte is at lowest address.
- ◆ Word is addressed by byte address of least significant byte.

3.4 Instruction Length

Instructions are either 32 bits long (in ARM state) or 16 bits long (in THUMB state).

3.5 Data Types

ARM7TDMI supports byte (8 bit), halfword (16 bit) and word (32 bit) data types. Words must be aligned to 4-byte boundaries and halfwords to 2-byte boundaries.

3.6 Operating Modes

ARM7TDMI supports seven modes of operation:

- User (usr): The normal ARM program execution state
- FIQ (fiq): Designed to support 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 may be made under software control, or may be brought about by external interrupts or exception processing. Most application programs will 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.

3.7 Registers

ARM7TDMI has a total of 37 registers (31 general purpose 32-bit registers and six status registers), but these cannot all be seen at once. The processor state and operating mode dictate which registers are available to the programmer.

3.7.1 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. All of these except R15 are general-purpose registers, and may be used to hold either data or address values. In addition to these, there is a seventeenth register used to store status information

3.7.1.1 Register 14

Used as the subroutine link register. This receives a copy of R15 when a Branch and Link (BL) instruction is executed. At all other times it may be treated as a general-purpose register. The corresponding banked registers R14_svc, R14_irq, R14_fiq, R14_abt and R14_und are similarly used to hold the return values of R15 when interrupts and exceptions arise, or when Branch and Link instructions are executed within interrupt or exception routines.

3.7.1.2 Register 15

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.
















3.7.1.3 Register 16

This is the CPSR (Current Program Status Register). This contains condition code flags and the current mode bits.






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.

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.

3.7.2 The THUMB State Register Set

The THUMB state register set is a subset of the ARM state set. The programmer has direct access to eight general registers, R0-R7, as well as 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. This is 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.

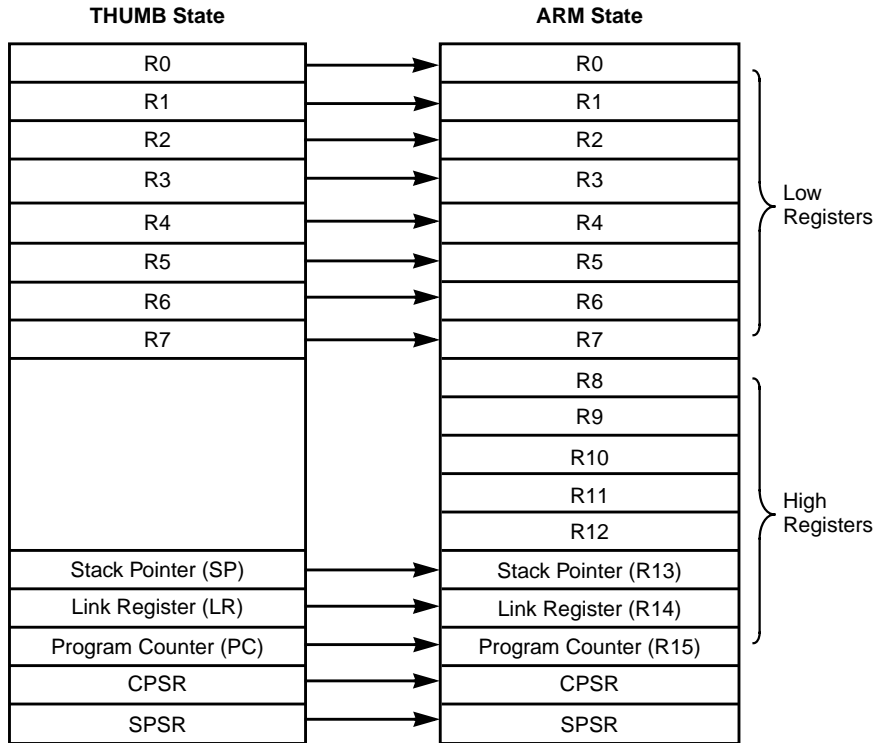
3.7.3 The Relationship Between ARM and THUMB State Registers

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 onto ARM state R13
- THUMB state LR maps onto ARM state R14
- The THUMB state Program Counter maps onto the ARM state Program Counter (R15)

This relationship is shown in [Figure 3.5](#).

Figure 3.5 Mapping of THUMB State Registers onto ARM State Registers



3.7.4 Accessing High Registers in THUMB State

In THUMB state, registers R8–R15 (the *High registers*) 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.

A value may be transferred from a register in the range R0–R7 (a Low register) to a High register, and from a High register to a Low register, using special variants of the `MOV` instruction. High register values can also be compared against or added to Low register values with the `CMP` and `ADD` instructions.

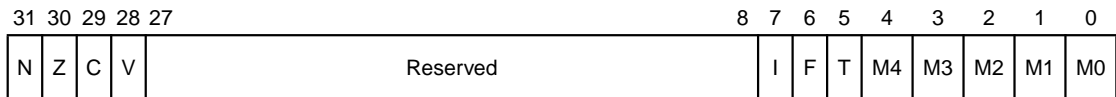
3.8 Program Status Registers

The ARM7TDMI contains a Current Program Status register (CPSR), plus five Saved Program Status registers (SPSRs) for use by exception handlers. 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 is shown in [Figure 3.6](#).

Figure 3.6 Program Status Register Format



3.8.1 The Condition Code Flags

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 may be changed as a result of arithmetic and logical operations, and may be tested to determine whether an instruction should be executed.

In ARM state, all instructions may be executed conditionally: see [Section 4.3, "Instruction Condition Field,"](#) for details.

In THUMB state, only the Branch instruction is capable of conditional execution.

3.8.2 Reserved Bits

Reserved **[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. Also, your program should not rely on them containing specific values, since in future processors they may read as one or zero.

3.8.3 The Control Bits

The bottom 8 bits of a PSR (incorporating I, F, T and M[4:0]) are known collectively as the control bits. These will change when an exception arises. If the processor is operating in a privileged mode, they can also be manipulated by software.

The T Bit **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 is reflected on the TBIT external signal.

Note that the software must never change the state of the TBIT in the CPSR. If this happens, 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 disable the IRQ and FIQ interrupts respectively.

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

The M4, M3, M2, M1 and M0 bits (M[4:0]) are the mode bits. These determine the processor's operating mode, as shown in the following table. Not all combinations of the mode bits define a valid processor mode. Only those explicitly described should be used. If any illegal value is programmed into the mode bits, M[4:0], then the processor will enter an unrecoverable state. If this occurs, reset should be applied. 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

3.9 Exceptions

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

It is possible for several exceptions to arise at the same time. If this happens, they are dealt with in a fixed order see [Section 3.9.10, “Exception Priorities”](#).

3.9.1 Action on Entering an Exception

When handling an exception, the ARM7TDMI:

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 (that is, current PC + 4 or PC + 8 depending on the exception. See

Table 3.2 Exception Entry/Exit for details). If the exception has been entered from THUMB state, then the value written into the Link register is the current PC offset by a value such that the program resumes from the correct place on return from the exception. This means that the exception handler need not determine from which state the exception was entered. For example, in the case of a Software Interrupt (SWI), `MOVS PC, R14_svc` will always return 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

It may also set the interrupt disable flags to prevent otherwise unmanageable nestings of exceptions.

If the processor is in THUMB state when an exception occurs, it will automatically switch into ARM state when the PC is loaded with the exception vector address.

3.9.2 Action on Leaving an Exception

On completion, the exception handler:

1. Moves the Link Register, minus an offset where appropriate, to the PC. (The offset will vary 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: An explicit switch back to THUMB state is never needed, since restoring the CPSR from the SPSR automatically sets the T bit to the value it held immediately prior to the exception.

3.9.3 Exception Entry/Exit Summary

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.

Table 3.2 Exception Entry/Exit

Exception	Return Instruction	Previous State		Notes
		ARMTHUMB R14_xR14_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 which had the prefetch abort.
2. Where PC is the address of the instruction which did not get executed since the FIQ or IRQ took priority.
3. Where PC is the address of the Load or Store instruction which generated the data abort.
4. The value saved in R14_svc upon reset is unpredictable.

3.9.4 Fast Interrupt Request (FIQ)

The FIQ (Fast Interrupt Request) exception is designed to support a data transfer or channel process, and in ARM state has sufficient private registers to remove the need for register saving (thus minimizing the overhead of context switching).

FIQ is externally generated by taking the nFIQ input LOW. This input can except either synchronous or asynchronous transitions, depending on the state of the ISYNC input signal. When ISYNC is LOW, nFIQ and nIRQ are considered asynchronous, and a cycle delay for synchronization is incurred before the interrupt can affect the processor flow.

Irrespective of whether the exception was entered from ARM or THUMB state, a FIQ handler should leave the interrupt by executing

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

FIQ may be disabled by setting the CPSR's F flag (but note that this is not possible from User mode). If the F flag is clear, ARM7TDMI checks for a LOW level on the output of the FIQ synchronizer at the end of each instruction.

3.9.5 Interrupt Request (IRQ)

The IRQ (Interrupt Request) exception is a normal interrupt caused by a LOW level on the nIRQ input. IRQ has a lower priority than FIQ and is masked out when an FIQ sequence is entered. It may be disabled at any time by setting the I bit in the CPSR, though this can only be done from a privileged (nonuser) mode.

Irrespective of whether the exception was entered from ARM or THUMB state, an IRQ handler should return from the interrupt by executing

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

3.9.6 Abort

An abort indicates that the current memory access cannot be completed. It can be signalled by the external ABORT input. ARM7TDMI checks for the abort exception during memory access cycles.

There are two types of abort:

Prefetch abort (PABT) occurs during an instruction prefetch.

Data abort (DABT) occurs during a data access.

If a prefetch abort occurs, the prefetched instruction is marked as invalid, but the exception will not be taken until the instruction reaches the head 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.

If 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.
3. Block data transfer instructions (`LDM`, `STM`) complete. If write back is set, the base is updated. If the instruction would have overwritten the base with data (i.e., it has the base in the transfer list), the overwriting is prevented. All register overwriting is prevented after an abort is indicated, which means in particular that R15 (always the last register to be transferred) is preserved 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 should execute the following irrespective of the state (ARM or THUMB):

```
SUBS PC,R14_abt,#4      for a prefetch abort
SUBS PC,R14_abt,#8      for a data abort
```

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

3.9.7 Software Interrupt (SWI)

The software interrupt instruction (`SWI`) is used for entering Supervisor mode, usually to request a particular supervisor function. A `SWI` handler should return by executing the following irrespective of the state (ARM or THUMB):

```
MOV PC, R14_svc
```

This restores the PC and CPSR, and returns to the instruction following the `SWI`.

3.9.8 Undefined Instruction (UDEF)

When ARM7TDMI comes across an instruction which it cannot handle, it takes the undefined instruction trap. This mechanism may be used to extend either the THUMB or ARM instruction set by software emulation.

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

```
MOVS PC,R14_und
```

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

3.9.9 Exception Vectors

[Table 3.3](#) lists the exception vector addresses.

Table 3.3 Exception Vectors

Address	Exception	Mode on Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

3.9.10 Exception Priorities

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

Highest priority:

1. Reset
2. Data abort

3. FIQ
4. IRQ
5. Prefetch abort

Lowest priority:

6. Undefined Instruction, Software interrupt.

3.9.10.1 Not All Exceptions Can Occur at Once

Undefined Instruction and Software Interrupt are mutually exclusive, since they each correspond to particular (nonoverlapping) decodings of the current instruction.

If a data abort occurs at the same time as a FIQ, and FIQs are enabled (i.e., the CPSR's F flag is clear), ARM7TDMI enters the data abort handler and then immediately proceeds to the FIQ vector. A normal return from FIQ will cause the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection. The time for this exception entry should be added to worst-case FIQ latency calculations.

3.10 Interrupt Latencies

The worst case latency for FIQ, assuming that it is enabled, consists of the longest time the request can take to pass through the synchronizer ($T_{syncmax}$ if asynchronous), plus the time for the longest instruction to complete (T_{ldm} , the longest instruction is an LDM which loads all the registers including the PC), plus the time for the data abort entry (T_{exc}), plus the time for FIQ entry (T_{fiq}). At the end of this time ARM7TDMI will be executing the instruction at 0x1C.

$T_{syncmax}$ is three processor cycles, T_{ldm} is 20 cycles, T_{exc} is three cycles, and T_{fiq} is two cycles. The total time is therefore 28 processor cycles. This is just over 1.4 microseconds in a system which uses a continuous 20 MHz processor clock. The maximum IRQ latency calculation is similar, but must allow for the fact that FIQ has higher priority and could delay entry into the IRQ handling routine for an arbitrary length of time. The minimum latency for FIQ or IRQ consists of the shortest time the request can take through the synchronizer ($T_{syncmin}$) plus T_{fiq} . This is four processor cycles.

3.11 Reset

When the nRESET signal goes LOW, ARM7TDMI abandons the executing instruction and then continues to fetch instructions from incrementing word addresses.

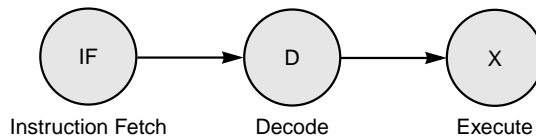
When nRESET goes HIGH again, ARM7TDMI:

1. Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The value of the saved PC and SPSR is not defined.
2. Forces M[4:0] to 0b10011 (Supervisor mode), sets the I and F bits in the CPSR, and clears the CPSR's T bit.
3. Forces the PC to fetch the next instruction from address 0x00.
4. Execution resumes in ARM state.

3.12 Pipeline Architecture

The ARM7TDMI core implements a three-stage pipeline (Instruction Fetch, Decode, and Execute) that always executes instructions in the order received and is fully interlocked in hardware. [Figure 3.7](#) shows the ARM7TDMI three-stage pipeline.

Figure 3.7 ARM7TDMI Pipeline



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

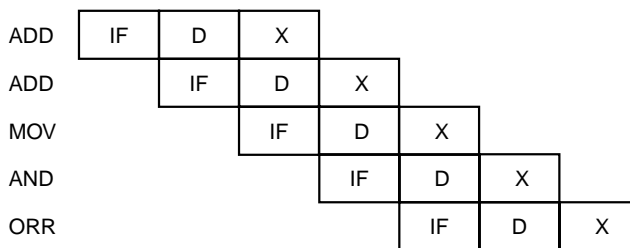
1. Instruction Fetch (IF) – The core fetches the instruction from memory.
2. Decode (D) – The core decodes the instruction and determines which registers are needed for this operation. If necessary, the core decompresses a 16-bit THUMB instruction into a 32-bit ARM instruction.

- Execute (X) – The core reads from the necessary register bank, executes all shift and ALU operations, and writes results to the appropriate register bank.

Pipeline operation is identical for both ARM and THUMB modes of operation. When in THUMB mode, the core decompresses each THUMB instruction (in the D stage) to provide the equivalent information that a decoded ARM instruction would provide. The only pipeline difference between ARM and THUMB mode is how the core handles the PC register (R15). The core increments PC by four addresses after each ARM instruction fetch, or by two addresses for each THUMB instruction fetch.

Figure 3.8 shows the ARM7TDMI pipeline executing code where all instructions operate on data already available in the CPU registers.

Figure 3.8 Pipeline Best Case Example



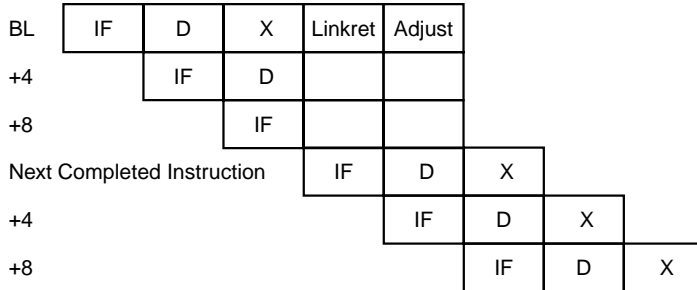
The only bus traffic in Figure 3.8 is from instruction fetches; one memory access for each instruction executed. In this example, the pipeline is working as efficiently as possible; there are no wasted slots in the pipeline and the pipeline is never stalled.

Figure 3.8 is a good example of a smooth and continuous pipeline flow. Of course, the ARM7TDMI pipeline doesn't always run so smoothly and other events can interrupt the pipeline operation. Specifically, there are four effects that can disrupt the continuous operation of the pipeline:

- Changes to the PC that cause changes in the program flow
- Hardware interrupts that cause changes to the program flow
- Multicycle instructions
- Instructions that require data accesses to memory

Changes to PC – The PC (R15) can change due to a direct modification, a branch operation, or an exception. When such a change in program flow occurs, the core flushes the pipeline and directs the first pipeline stage to fetch the instruction pointed to by the new PC value. [Figure 3.9](#) shows the ARM7TDMI pipeline executing code in ARM mode when a branch instruction occurs.

Figure 3.9 Pipeline Branch Example

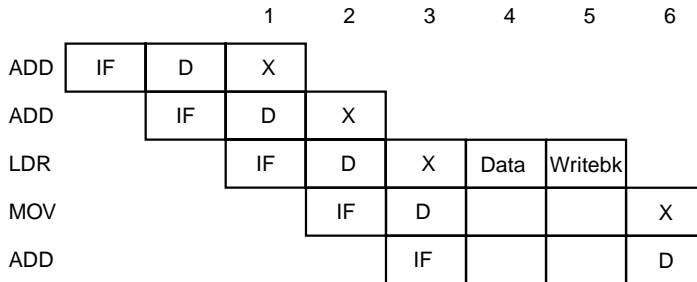


Following the branch (BL) instruction, two instructions are fetched (prefetched) in the branch shadow. Since the ARM architecture does not allow for branch delay slots, these two instructions are discarded before either reaches the Execute stage. After these two discarded instructions, the pipeline flow returns to normal operation. The third instruction and all subsequent instructions will fetch, decode, and execute as normal.

Hardware interrupts – When an interrupt activates, it changes the program flow and alters the pipeline execution. The core completes current instruction execution, flushes the pipeline, and directs the first stage in the pipeline to fetch a new instruction from the interrupt exception vector address. Worst-case interrupt latencies are discussed in [Section 3.10, “Interrupt Latencies.”](#)

[Figure 3.10](#) shows the ARM7TDMI pipeline operation when the flow is interrupted by an interrupt IRQ.

Figure 3.11 Pipeline Data Memory Access Example



When the `LDR` instruction executes, it causes the pipeline to stall for two cycles. The core has already fetched the two instructions subsequent to the `LDR` and they remain in the pipeline. These two instructions are stalled but not discarded, as in a branch or interrupt operation. In the first stall cycle (cycle 4), the core reads data for the `LDR` from memory. In the second stall cycle (cycle 5), the core writes the data to the internal register file. If the example contained an `STR` (store register) instruction rather than `LDR`, the pipeline would stall for a single cycle for a write to memory. `STR` does not require a register write back.

So far, the pipeline operation examples have assumed that `nWAIT` is never asserted. If the core asserts `nWAIT` in any clock cycle, this stops the core clock, so that all stages of the pipeline (and everything else in the core) stop while `nWAIT` remains asserted. `nWAIT` clearly has an important effect on execution time and interrupt latency in any real system.

Chapter 4

ARM Instruction Set Summary

This chapter provides a summary of the ARM instruction set. It contains the following sections:

- [Section 4.1, “Instruction Set Summary,” page 4-1](#)
- [Section 4.2, “Format Summary,” page 4-3](#)
- [Section 4.3, “Instruction Condition Field,” page 4-4](#)
- [Section 4.4, “Instruction Set Examples,” page 4-5](#)

For detailed information on the ARM instruction set, see the *ARM Architectural Reference Manual*.

4.1 Instruction Set Summary

The ARM instruction set is summarized below.

Table 4.1 ARM Instruction Set

Mnemonic	Instruction	Action
ADC	Add with carry	$Rd := Rn + Op2 + \text{Carry}$
ADD	Add	$Rd := Rn + Op2$
AND	And	$Rd := Rn \text{ AND } Op2$
B	Branch	$R15 := \text{address}$
BIC	Bit clear	$Rd := Rn \text{ AND NOT } Op2$
BL	Branch with link	$R14 := R15, R15 := \text{address}$

(Sheet 1 of 3)

Table 4.1 ARM Instruction Set (Cont.)

Mnemonic	Instruction	Action
BX	Branch and exchange	R15 : = Rn, T bit : = Rn[0]
CDP	Coprocessor data processing	(Coprocessor-specific)
CMN	Compare negative	CPSR flags : = Rn + Op2
CMP	Compare	CPSR flags : = Rn - Op2
EOR	Exclusive OR	Rd : = (Rn AND NOT Op2) OR (op2 AND NOT Rn)
LDC	Load coprocessor from memory	Coprocessor load
LDM	Load multiple registers	Stack manipulation (Pop)
LDR	Load register from memory	Rd : = (address)
MCR	Move CPU register to coprocessor register	cRn : = rRn {<op>cRm}
MLA	Multiply accumulate	Rd : = (Rm * Rs) + Rn
MOV	Move register or constant	Rd : = Op2
MRC	Move from coprocessor register to CPU register	Rn : = cRn {<op>cRm}
MRS	Move PSR status/flags to register	Rn : = PSR
MSR	Move register to PSR status/flags	PSR : = Rm
MUL	Multiply	Rd : = Rm * Rs
MVN	Move negative register	Rd : = 0xFFFFFFFF EOR Op2
ORR	Or	Rd : = Rn OR Op2
RSB	Reverse subtract	Rd : = Op2 - Rn
RSC	Reverse subtract with carry	Rd : = Op2 - Rn - 1 + Carry
SBC	Subtract with carry	Rd : = Rn - Op2 - 1 + Carry
STC	Store coprocessor register to memory	address : = CRn
STM	Store multiple	Stack manipulation (Push)
(Sheet 2 of 3)		

In the absence of a suffix, the condition field of most instructions is set to 'Always' (suffix AL). This means the instruction will always be executed regardless of the CPSR condition codes.

Table 4.2 Condition Code Summary

Code	Suffix	Flags	Meaning
0b0000	EQ	Z Set	equal
0b0001	NE	Z Clear	not equal
0b0010	CS	C Set	unsigned higher or same
0b0011	CC	C Clear	unsigned lower
0b0100	MI	N Set	negative
0b0101	PL	N Clear	positive or zero
0b0110	VS	V Set	overflow
0b0111	VC	V Clear	no overflow
0b1000	HI	C Set and Z Clear	unsigned higher
0b1001	LS	C Clear or Z Set	unsigned lower or same
0b1010	GE	N Equals V	greater or equal
0b1011	LT	N not Equal to V	less than
0b1100	GT	Z Clear AND (N Equals V)	greater than
0b1101	LE	Z Set OR (N not Equal to V)	less than or equal
0b1110	AL	(ignored)	always

4.4 Instruction Set Examples

The following examples show ways in which the basic core instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some), mostly they just save code.

4.4.1 Using the Conditional Instructions

4.4.1.1 Using Conditionals for Logical OR

```
CMP    Rn,#p        ; If Rn = p OR Rm = q THEN GOTO
                        ; Label.
BEQ    Label
CMP    Rm,#q
BEQ    Label
```

This can be replaced by

```
CMP    Rn,#p
CMPNE  Rm,#q        ; If condition not satisfied try
                        ; other test.
BEQ    Label
```

4.4.1.2 Absolute Value

```
TEQ    Rn,#0        ; Test sign
RSBMI  Rn,Rn,#0     ; and 2's complement if necessary.
```

4.4.1.3 Multiplication by 4, 5 or 6 (Run Time)

```
MOV    Rc,Ra,LSL#2; Multiply by 4,
CMP    Rb,#5        ; test value,
ADDCS  Rc,Rc,Ra     ; complete multiply by 5,
ADDHI  Rc,Rc,Ra     ; complete multiply by 6.
```

4.4.1.4 Combining Discrete and Range Tests

```
TEQ    Rc,#127     ; Discrete test,
CMPNE  Rc,#" "-1   ; range test
MOVLS  Rc,#"."     ; IF Rc<=" " OR Rc=ASCII(127)
                        ; THEN Rc:="."
```

4.4.1.5 Division and Remainder

A short general purpose divide routine follows.

```
                        ; Enter with numbers
                        ; in Ra and Rb.
                        ;
MOV    Rcnt,#1        ; Bit to control the
                        ; division.
Div1  CMP    Rb,#0x80000000 ; Move Rb until
                        ; greater than Ra.
```

```

        CMPCC Rb,Ra
        MOVCC Rb,Rb,ASL#1
        MOVCC Rcnt,Rcnt,ASL#1
        BCC Div1
        MOV Rc,#0
Div2  CMP Ra,Rb ; Test for possible
        ; subtraction.
        SUBCS Ra,Ra,Rb ; Subtract if ok,
        ADDCS Rc,Rc,Rcnt ; put relevant bit
        ; into result
        MOVS Rcnt,Rcnt,LSR#1 ; shift control bit
        MOVNE Rb,Rb,LSR#1 ; halve unless
        ; finished.
        BNE Div2
        ;
        ; Divide result in Rc,
        ; remainder in Ra.

```

4.4.1.6 Overflow Detection in the ARM7TDMI

Overflow in unsigned multiply with a 32-bit result

```

UMULL Rd,Rt,Rm,Rn ;3 to 6 cycles
TEQ Rt,#0 ;+ 1 cycle and a register
BNE overflow

```

Overflow in signed multiply with a 32-bit result

```

SMULL Rd,Rt,Rm,Rn ;3 to 6 cycles
TEQ Rt,Rd,ASR#31 ;+ 1 cycle and a register
BNE overflow

```

Overflow in unsigned multiply accumulate with a 32-bit result

```

UMLAL Rd,Rt,Rm,Rn ;4 to 7 cycles
TEQ Rt,#0 ;+ 1 cycle and a register
BNE overflow

```

Overflow in signed multiply accumulate with a 32-bit result

```

SMLAL Rd,Rt,Rm,Rn ;4 to 7 cycles
TEQ Rt,Rd,ASR#31 ;+ 1 cycle and a register
BNE overflow

```

Overflow in unsigned multiply accumulate with a 64-bit result

```

UMULL Rl,Rh,Rm,Rn ;3 to 6 cycles
ADDS Rl,Rl,Ra1 ;lower accumulate
ADC Rh,Rh,Ra2 ;upper accumulate
BCS overflow ;1 cycle and 2 registers

```

Overflow in signed multiply accumulate with a 64-bit result

```
SMULL Rl,Rh,Rm,Rn      ;3 to 6 cycles
ADDS  Rl,Rl,Ra1        ;lower accumulate
ADC   Rh,Rh,Ra2        ;upper accumulate
BVS   overflow         ;1 cycle and 2 registers
```

Note: Overflow checking is not applicable to unsigned and signed multiplies with a 64-bit result, since overflow does not occur in such calculations.

4.4.2 Pseudo-Random Binary Sequence Generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive-OR feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (i.e. $2^{32} - 1$ cycles before repetition), so this example uses a 33-bit register with taps at bits 33 and 20. The basic algorithm is newbit: = bit 33 EOR bit 20, shift left the 33-bit number and put in newbit at the bottom; this operation is performed for all the new bits needed (i.e. 32 bits). The entire operation can be done in 5 S cycles:

```
                                ; Enter with seed in Ra (32 bits),
                                ; Rb (1 bit in Rb lsb), uses Rc.
                                ;
TST   Rb,Rb,LSR#1              ; Top bit into carry
MOVS  Rc,Ra,RRX                ; 33-bit rotate right
ADC   Rb,Rb,Rb                 ; carry into lsb of Rb
EOR   Rc,Rc,Ra,LSL#12          ; (involved!)
EOR   Ra,Rc,Rc,LSR#20          ; (similarly involved!)
                                ; new seed in Ra, Rb as before
```

4.4.3 Multiplication by Constant Using the Barrel Shifter

- Multiplication by 2^n (1, 2, 4, 8, 16, 32..)

```
MOV   Ra, Rb, LSL #n
```

- Multiplication by $2^n + 1$ (3, 5, 9, 17..)

```
ADD   Ra,Ra,Ra,LSL #n
```

- Multiplication by $2^n - 1$ (3, 7, 15..)

```
RSB   Ra,Ra,Ra,LSL #n
```


- Multiplication by 6

```
ADD    Ra,Ra,Ra,LSL #1; multiply by 3
MOV    Ra,Ra,LSL#1  ; and then by 2
```

- Multiplication by 10 and Add in Extra Number

```
ADD    Ra,Ra,Ra,LSL#2; multiply by 5
ADD    Ra,Rc,Ra,LSL#1; multiply by 2 and add in
                        next digit
```

- General Recursive Method for $Rb := Ra * C$, C a Constant:

- If C even, say $C = 2^n * D$, D odd:

```
D=1:  MOV    Rb,Ra,LSL #n
D<>1: {Rb := Ra*D}
      MOV    Rb,Rb,LSL #n
```

- If $C \text{ MOD } 4 = 1$, say $C = 2^n * D + 1$, D odd, $n > 1$:

```
D=1:  ADD    Rb,Ra,Ra,LSL #n
D<>1: {Rb := Ra*D}
      ADD    Rb,Ra,Rb,LSL #n
```

- If $C \text{ MOD } 4 = 3$, say $C = 2^n * D - 1$, D odd, $n > 1$:

```
D=1:  RSB    Rb,Ra,Ra,LSL #n
D<>1: {Rb := Ra*D}
      RSB    Rb,Ra,Rb,LSL #n
```

This is not quite optimal, but close. For example, a multiply by 45 is done by:

```
RSB    Rb,Ra,Ra,LSL#2  ; multiply by 3
RSB    Rb,Ra,Rb,LSL#2  ; multiply by 4 * 3-1 = 11
ADD    Rb,Ra,Rb,LSL# 2 ; multiply by 4 * 11 + 1 = 45
```

rather than by:

```
ADD    Rb,Ra,Ra,LSL#3  ; multiply by 9
ADD    Rb,Rb,Rb,LSL#2  ; multiply by 5 * 9 = 45
```

4.4.4 Loading a Word from an Unknown Alignment

```
                                ; enter with address in Ra (32 bits)
                                ; uses Rb, Rc; result in Rd.
                                ; Note d must be less than c e.g. 0,1
                                ;
BIC    Rb,Ra,#3                ; get word aligned address
LDMIA  Rb,{Rd,Rc}              ; get 64 bits containing answer
AND    Rb,Ra,#3                ; correction factor in bytes
MOVS   Rb,Rb,LSL#3             ; ...now in bits and test if aligned
MOVNE  Rd,Rd,LSR Rb           ; produce bottom of result word
                                ; (if not aligned)
RSBNE  Rb,Rb,#32              ; get other shift amount
ORRNE  Rd,Rd,Rc,LSL Rb        ; combine two halves to get result
```

Chapter 5

THUMB Instruction Set Summary

This chapter describes the THUMB instruction set. It contains the following sections:

- [Section 5.1, “Instruction Set Summary,” page 5-1](#)
- [Section 5.2, “Format Summary,” page 5-3](#)
- [Section 5.3, “Instruction Set Examples,” page 5-4](#)

For detailed information on the THUMB instruction set, see the *ARM Architectural Reference Manual*.

5.1 Instruction Set Summary

The following table summarizes the THUMB instruction set.

Table 5.1 THUMB Instruction Set

Mnemonic	Instruction	Equivalent ARM Instructions	Low Register Operand	High Register Operand	Condition Codes Set
ADC	Add with carry	ADC	✓		✓
ADD	Add	ADD	✓	✓	✓ ¹
AND	Logical and	AND	✓		✓
ASR	Arithmetic shift right	MOV	✓		✓
B	Unconditional branch	B	✓		
Bxx	Conditional branch	B	✓		

(Sheet 1 of 3)

Table 5.1 THUMB Instruction Set (Cont.)

Mnemonic	Instruction	Equivalent ARM Instructions	Low Register Operand	High Register Operand	Condition Codes Set
BIC	Bit clear	BIC	✓		✓
BL	Branch and link	BL			
BX	Branch and exchange	BX	✓	✓	
CMN	Compare negative	CMN	✓		✓
CMP	Compare	CMP	✓	✓	✓
EOR	Exclusive or	EOR	✓		✓
LDMIA	Load multiple	LDM	✓		
LDR	Load word	LDR	✓		
LDRB	Load byte	LDR	✓		
LDRH	Load halfword	LDR	✓		
LDRSB	Load register signed byte	LDR	✓		
LDRSH	Load register halfword	LDR	✓		
LSL	Logical shift left	MOV	✓		✓
LSR	Logical shift right	MOV	✓		✓
MOV	Move register	MOV	✓	✓	✓ ²
MUL	Multiply	MUL	✓		✓
MVN	Move negative register	MVN	✓		✓
NEG	Negate	RSB	✓		✓
ORR	Logical or	ORR	✓		✓
POP	Pop registers	LDM	✓		
PUSH	Push registers	LDM	✓		
ROR	Rotate right	MOV	✓		✓
SBC	Subtract with carry	SBC	✓		✓
(Sheet 2 of 3)					

Table 5.1 THUMB Instruction Set (Cont.)

Mnemonic	Instruction	Equivalent ARM Instructions	Low Register Operand	High Register Operand	Condition Codes Set
STMIA	Store multiple	STM	✓		
STR	Store word	STR	✓		
STRB	Store byte	STR	✓		
STRH	Store halfword	STR	✓		
SUB	Subtract	SUB	✓		✓
SWI	Software interrupt	SWI			
TST	Test bits	TST	✓		✓
(Sheet 3 of 3)					

1. The condition codes are unaffected by the format 5, 12 and 13 versions of this instruction.
2. The condition codes are unaffected by the format 5 version of this instruction.

5.1.1 Instruction Cycle Time

All THUMB instructions have an equivalent ARM instruction as shown in the table above. The instruction cycle times for the THUMB instructions are identical to that of the equivalent ARM instruction. For more information on instruction cycle time, refer to [Chapter 10, "Instruction Cycle Operations."](#)

5.2 Format Summary

The THUMB instruction set formats are shown in the following figure.

Figure 5.1 THUMB Instruction Set Formats

	15	14	13	12	11	10	9	8	7	6	5	3	2	0			
1	0	0	0	Op	Offset5			Rs	Rd		Move Shifted Register						
2	0	0	0	1	1	1	Op	Rn/offset3		RS	Rd		Add/Subtract				
3	0	0	1	Op	Rd		Offset8					Move/Compare/Add/Subtract/Immediate					
4	0	1	0	0	0	0	Op		Rs	Rd		ALU Operations					
5	0	1	0	0	0	1	Op	H1	H2	Rs/Hs	Rd/Hd		Hi Register Operations/Branch Exchange				
6	0	1	0	0	1	Rd		Word8					PC Relative Load				
7	0	1	0	1	L	S	0	Ro		Rb	Rd		Load/Store with Register Offset				
8	0	1	0	1	H	S	1	Ro		Rb	Rd		Load/Store Sign-Extended Byte/Halfword				
9	0	1	1	B	L	Offset5			Rb	Rd		Load/Store with Immediate Offset					
10	1	0	0	0	L	Offset5			Rb	Rd		Load/Store Halfword					
11	1	0	0	1	L	Rd		Work8					SP-Relative Load/Store				
12	1	0	1	0	SP	Rd		Work8					Load Address				
13	1	0	1	1	0	0	0	0	S	SWord7					Add Offset to Stack Pointer		
14	1	0	1	1	L	1	0	R	Rlist					Push/Pop Registers			
15	1	1	0	0	L	Rb		Rlist					Multiple Load/Store				
16	1	1	0	1	Cond			Soffset8					Conditional Branch				
17	1	1	0	1	1	1	1	1	Value8					Software Interrupt			
18	1	1	1	0	0	Offset11					Unconditional Branch						
19	1	1	1	1	H	Offset					Long Branch with Link						

5.3 Instruction Set Examples

The following examples show ways in which the THUMB instructions may be used to generate small and efficient code. Each example also shows the ARM equivalent so these may be compared.

5.3.1 Multiplication by a Constant Using Shifts and Adds

The following shows code to multiply by various constants using one, two or three Thumb instructions along side the ARM equivalents. For other constants it is generally better to use the built-in `MUL` instruction rather than using a sequence of four or more instructions.

THUMB

ARM

Multiplication by 2^n (1, 2, 4, 8, ...)

LSL Ra, Rb, LSL #n MOV Ra, Rb, LSL #n

Multiplication by $2^n + 1$ (3, 5, 9, 17, ...)

LSL Rt, Rb, #n ADD Ra, Rb, Rb, LSL #n
ADD Ra, Rt, Rb

Multiplication by $2^n - 1$ (3, 7, 15, ...)

LSL Rt, Rb, #n RSB Ra, Rb, Rb, LSL #n
SUB Ra, Rt, Rb

Multiplication by -2^n (-2, -4, -8, ...)

LSL Ra, Rb, #n MOV Ra, Rb, LSL #n
MVN Ra, Ra RSB Ra, Ra, #0

Multiplication by $-2^n - 1$ (-3, -7, -15, ...)

LSL Rt, Rb, #n SUB Ra, Rb, Rb, LSL #n
SUB Ra, Rb, Rt

Multiplication by any $C = \{2^n + 1, 2^n - 1, -2^n \text{ or } -2^n - 1\} * 2^n$

Effectively this is any of the multiplications in 2 to 5 followed by a final shift.

This allows the following additional constants to be multiplied.

6, 10, 12, 14, 18, 20, 24, 28, 30, 34, 36, 40, 48, 56,
60, 62

(2..5) (2..5)
LSL Ra, Ra, #n MOV Ra, Ra, LSL #n

5.3.2 General Purpose Signed Divide

This example shows a general purpose signed divide and remainder routine in both THUMB and ARM code.

5.3.2.1 Thumb Code

```
signed_divide
; Signed divide of R1 by R0: returns quotient in R0,
; remainder in R1

; Get abs value of R0 into R3
    ASR     R2, R0, #31 ; Get 0 or -1 in R2 depending
                        ; on sign of R0
    EOR     R0, R2      ; EOR with -1 (0xFFFFFFFF) if
                        ; negative
    SUB     R3, R0, R2  ; and ADD 1 (SUB -1) to get
                        ; abs value
; SUB always sets flag so go & report division by 0 if
; necessary
;     BEQ     divide_by_zero

; Get abs value of R1 by xoring with 0xFFFFFFFF and adding 1
; if negative
    ASR     R0, R1, #31 ; Get 0 or -1 in R3 depending
                        ; on sign of R1
    EOR     R1, R0      ; EOR with -1 (0xFFFFFFFF) if
                        ; negative
    SUB     R1, R0      ; and ADD 1 (SUB -1) to get
                        ; abs value

; Save signs (0 or -1 in R0 & R2) for later use in
; determining sign of quotient & remainder.
    PUSH     {R0, R2}

; Justification, shift 1 bit at a time until divisor (R0
; value) is just <= than dividend (R1 value). To do this
; shift dividend right by 1 and stop as soon as shifted
; value becomes >.
    LSR     R0, R1, #1
    MOV     R2, R3
    B       %FT0

just_1 LSL     R2, #1
0     CMP     R2, R0
     BLS     just_1

     MOV     R0, #0      ; Set accumulator to 0
     B       %FT0      ; Branch into division loop

div_1 LSR     R2, #1
0     CMP     R1, R2 ; Test subtract
     BCC     %FT0
     SUB     R1, R2      ; If successful do a real
```



```

                                ; subtract
0      ADC      R0, R0          ; Shift result and add 1 if
                                ; subtract succeeded

                                CMP      R2, R3          ; Terminate when R2 == R3 (we
                                ; have just
                                BNE      div_1           ; tested subtracting the
                                ; 'ones' value)

                                ; Now fixup the signs of the quotient (R0) and
                                ; remainder (R1)
POP      {R2, R3}           ; Get dividend/divisor signs
                                ; back
EOR      R3, R2             ; Result sign
EOR      R0, R3             ; Negate if result sign = -1
SUB      R0, R3

EOR      R1, R2             ; Negate remainder if dividend
                                ; sign = -1
SUB      R1, R2

MOV      pc, lr

```

5.3.2.2 ARM Code

```

signed_divide
; effectively zero a4 as top bit will be shifted out later
    ANDS    a4, a1, #&80000000
    RSBMI   a1, a1, #0
    EORS    ip, a4, a2, ASR #32
; ip bit 31 = sign of result
; ip bit 30 = sign of a2
    RSBCS   a2, a2, #0

; central part is identical code to udiv
; (without MOV a4, #0 which comes for free as part of signed
; entry sequence)
    MOVS    a3, a1
    BEQ     divide_by_zero

just_1
; justification stage shifts 1 bit at a time
    CMP     a3, a2, LSR #1
    MOVLS   a3, a3, LSL #1
            ; NB: LSL #1 is always OK if LS succeeds
    BLO     s_loop

```

```

div_1
    CMP     a2, a3
    ADC     a4, a4, a4
    SUBCS   a2, a2, a3

    TEQ     a3, a1
    MOVNE   a3, a3, LSR #1
    BNE     s_loop2
    MOV     a1, a4

    MOVS    ip, ip, ASL #1
    RSBCS   a1, a1, #0
    RSBMI   a2, a2, #0

    MOV     pc, lr

```

5.3.3 Division by a Constant

Division by a constant can often be performed by a short fixed sequence of shifts, adds and subtracts. For an explanation of the algorithm see *The ARM Cookbook* (ARM DUYI-0005B), section entitled “*Division by a constant.*”

Here is an example of a divide by 10 routine based on the algorithm in the ARM Cookbook in both THUMB and ARM code.

5.3.3.1 THUMB Code

```

udiv10
; takes argument in a1
; returns quotient in a1, remainder in a2
    MOV     a2, a1
    LSR     a3, a1, #2
    SUB     a1, a3
    LSR     a3, a1, #4
    ADD     a1, a3
    LSR     a3, a1, #8
    ADD     a1, a3
    LSR     a3, a1, #16
    ADD     a1, a3
    LSR     a1, #3
    ASL     a3, a1, #2
    ADD     a3, a1
    ASL     a3, #1
    SUB     a2, a3
    CMP     a2, #10
    BLT     %FT0

```

```

        ADD    a1, #1
        SUB    a2, #10
0       MOV    pc, lr

```

5.3.3.2 ARM Code

```

udiv10
; takes argument in a1
; returns quotient in a1, remainder in a2
        SUB    a2, a1, #10
        SUB    a1, a1, a1, lsr #2
        ADD    a1, a1, a1, lsr #4
        ADD    a1, a1, a1, lsr #8
        ADD    a1, a1, a1, lsr #16
        MOV    a1, a1, lsr #3
        ADD    a3, a1, a1, asl #2
        SUBS   a2, a2, a3, asl #1
        ADDPL  a1, a1, #1
        ADDMI  a2, a2, #10
        MOV    pc, lr

```


Chapter 6

Memory Interface

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

- [Section 6.1, “Overview,” page 6-1](#)
 - [Section 6.2, “Cycle Types,” page 6-2](#)
 - [Section 6.3, “Address Timing,” page 6-4](#)
 - [Section 6.4, “Data Transfer Size,” page 6-7](#)
 - [Section 6.5, “Instruction Fetch,” page 6-8](#)
 - [Section 6.6, “Memory Management,” page 6-10](#)
 - [Section 6.7, “Locked Operations,” page 6-10](#)
 - [Section 6.8, “Stretching Access Times,” page 6-11](#)
 - [Section 6.9, “ARM7TDMI Data Bus,” page 6-11](#)
 - [Section 6.10, “External Data Bus,” page 6-13](#)
-

6.1 Overview

ARM7TDMI’s memory interface consists of the following basic elements:

- 32-bit address bus
This specifies to memory the location to be used for the transfer.
- 32-bit data bus
Instructions and data are transferred across this bus. Data may be word, halfword, or byte wide in size.
- A bidirectional data bus, D[31:0], and separate unidirectional data buses, DIN[31:0] and DOUT[31:0].

Most of the text in this chapter describes the bus behavior assuming that the bidirectional bus is in use. However, the behavior applies equally to the unidirectional buses.

- Control signals

These specify, for example, the size of the data to be transferred, and the direction of the transfer together with providing privileged information.

This collection of signals allow the core to be simply interfaced to DRAM, SRAM and ROM. To fully exploit page mode access to DRAM, information is provided on whether or not the memory accesses are sequential. In general, interfacing to static memories is much simpler than interfacing to dynamic memory.

6.2 Cycle Types

All memory transfer cycles can be placed in one of four categories:

1. Nonsequential cycle. The core requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.
2. Sequential cycle. The core requests a transfer to or from an address which is either the same as the address in the preceding cycle, or is one word or halfword after the preceding address.
3. Internal cycle. The core does not require a transfer, as it is performing an internal function and no useful prefetching can be performed at the same time.
4. Coprocessor register transfer. The core wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

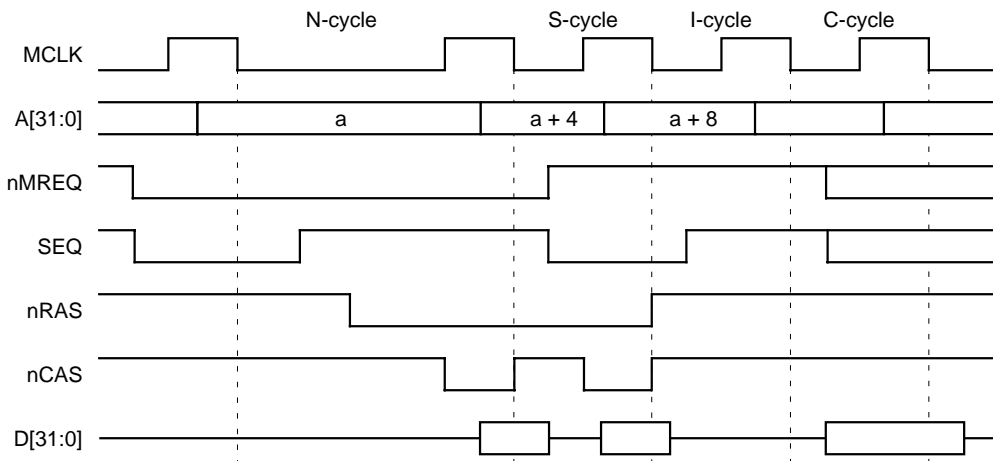
These four classes are distinguishable to the memory system by inspection of the nMREQ and SEQ control lines, see [Table 6.1](#). These control lines are generated during phase 1 of the cycle before the cycle whose characteristics they forecast, and this pipelining of the control information gives the memory system sufficient time to decide whether or not it can use a page mode access.

Table 6.1 Memory Cycle Types

nMREQ	SEQ	Cycle type
0	0	Nonsequential (N-cycle)
0	1	Sequential (S-cycle)
1	0	Internal (I-cycle)
1	1	Coprocessor register transfer (C-cycle)

Figure 6.1 shows the pipelining of the control signals, and suggests how the DRAM address strobes (nRAS and nCAS) might be timed to use page mode for S-cycles. Note that the N-cycle is longer than the other cycles. This is to allow for the DRAM precharge and row access time, and is not a core requirement.

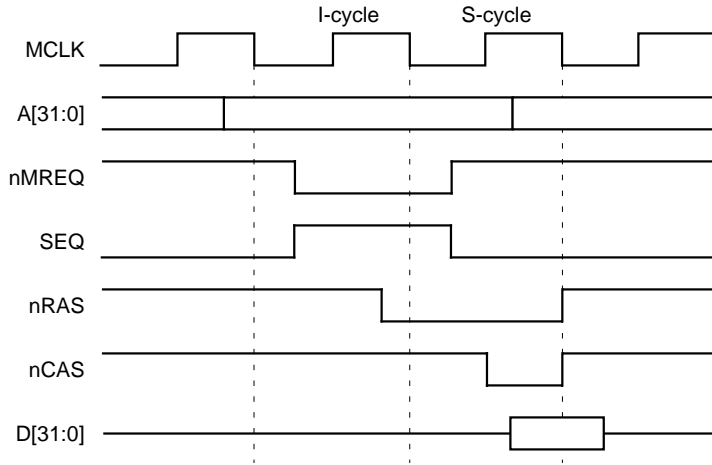
Figure 6.1 ARM Memory Cycle Timing



When an S-cycle follows an N-cycle, the address will always be one word or halfword greater than the address used in the N-cycle. This address (marked "a" in the above diagram) should be checked to ensure that it is not the last in the DRAM page before the memory system commits to the S-cycle. If it is at the page end, the S-cycle cannot be performed in page mode and the memory system will have to perform a full access.

The processor clock must be stretched to match the full access. When an S-cycle follows an I-cycle, the address will be the same as that used in the I-cycle. This fact may be used to start the DRAM access during the preceding cycle, which enables the S-cycle to run at page mode speed while performing a full DRAM access. This is shown in [Figure 6.2](#).

Figure 6.2 Memory Cycle Optimization



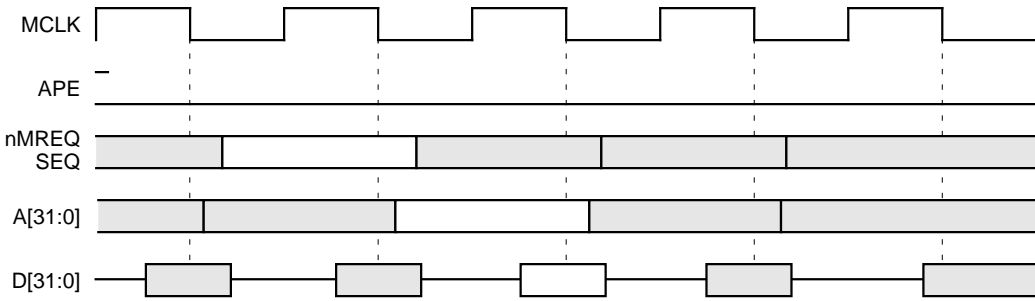
6.3 Address Timing

ARM7TDMI's address bus can operate in one of two configurations — pipelined or depipelined—this is controlled by the APE input signal. These configurations make it easy to design both SRAM and DRAM based systems with the ARM7TDMI core.

It is a requirement of SRAMs and ROMs that the address be held stable throughout the memory cycle. In a system containing SRAM and ROM only, APE may be tied permanently LOW, producing the desired address timing. This is shown in [Figure 6.3](#).

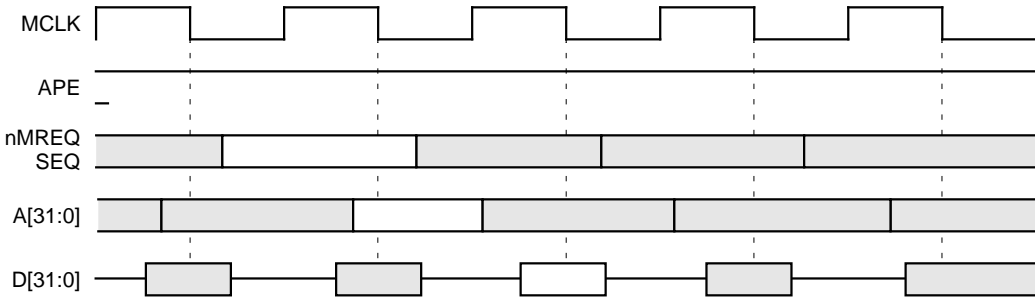
Note: APE effects the timing of the address bus A[31:0], plus nRW, MAS[1:0], LOCK, nOPC and nTRANS.

Figure 6.3 ARM7TDMI Depipelined Addresses



In a DRAM based system, it is desirable to obtain the address from the core as early as possible. When APE is HIGH, the core's address becomes valid in the MCLK HIGH phase before the memory cycle to which it refers. This timing allows longer for address decoding and the generation of DRAM control signals. [Figure 6.4](#) shows the effect on the timing when APE is HIGH.

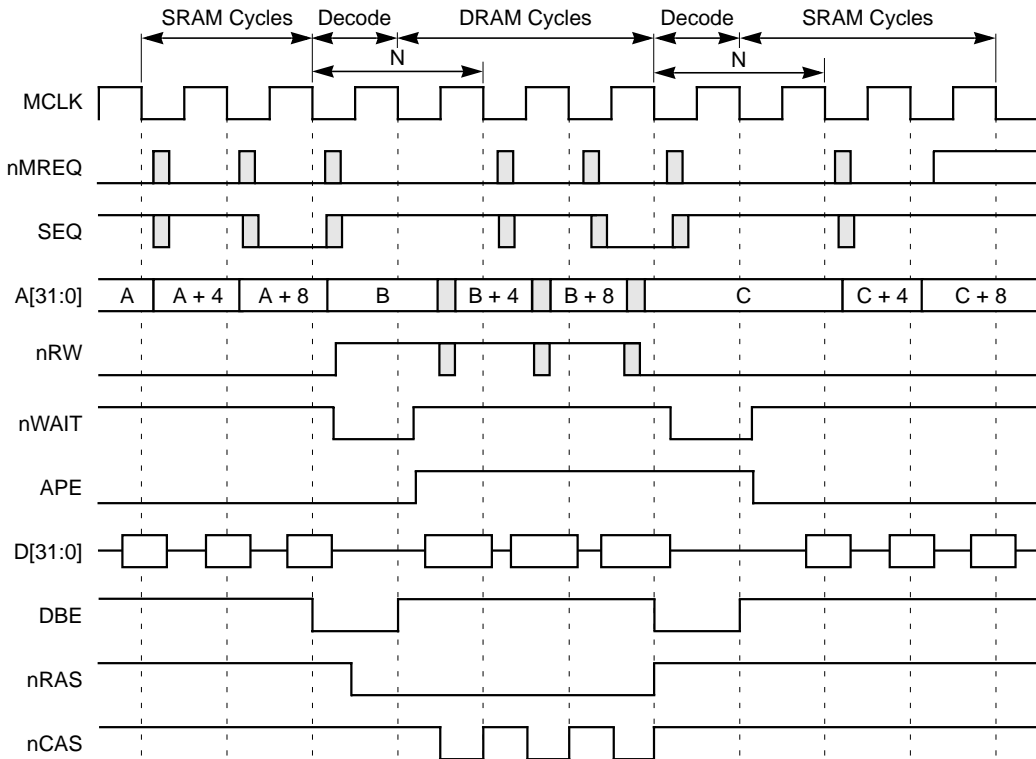
Figure 6.4 ARM7TDMI Pipelined Addresses



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

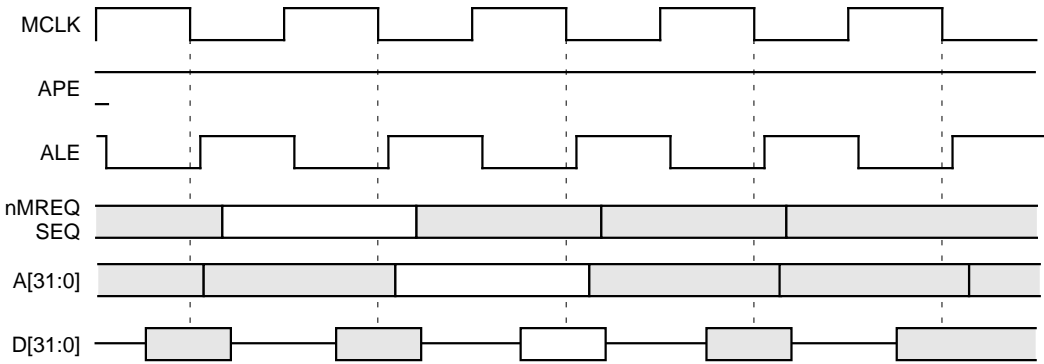
By way of an example, [Figure 6.5](#), shows a combination of accesses to a mixed DRAM/SRAM system. Here, the SRAM has zero wait states, and the DRAM has a 2:1 N-cycle/S-cycle ratio. A single wait state is inserted for address decode when a nonsequential access occurs. Typical, externally generated DRAM control signals are also shown.

Figure 6.5 Typical System Timing



Previous ARM processors included the ALE signal, and this is retained for backwards compatibility. This signal also allows the address timing to be modified to achieve the same results as APE, but in an asynchronous manner. To obtain clean MCLK LOW timing of the address bus by this mechanism, ALE must be driven HIGH with the falling edge of MCLK, and LOW with the rising edge of MCLK. ALE can simply be the inverse of MCLK but the delay from MCLK to ALE must be carefully controlled so that the T_{ald} timing constraint is achieved. [Figure 6.6](#) shows how ALE can be used to achieve SRAM compatible address timing. Refer to *CW001007 ARM7TDMI Microprocessor Core Datasheet* for details of the exact timing constraints.

Figure 6.6 SRAM Compatible Address Timing



Note: If ALE is to be used to change address timing, then APE must be tied HIGH. Similarly, if APE is to be used, ALE must be tied HIGH.

6.4 Data Transfer Size

In an ARM7TDMI core system, words, halfwords or bytes may be transferred between the processor and the memory. The size of the transaction taking place is determined by the MAS[1:0] pins. These are encoded as follows:

MAS[1:0]	00	Byte
	01	Halfword
	10	Word
	11	Reserved

The processor always produces a byte address, but instructions are either words (4 bytes) or halfwords (2 bytes), and data can be any size. Note that when word instructions are fetched from memory, A[1:0] are undefined and when halfword instructions are fetched, A[0] is undefined. The MAS[1:0] outputs share the same timing as the address bus and thus can be modified by the use of ALE and APE as described in [Section 6.3, “Address Timing.”](#)

When a data read of byte or halfword size is performed (e.g., `LDRB`), the memory system may safely ignore the fact that the request is for a subword sized quantity and present the whole word. The core will always

correctly extract the addressed byte or halfword from the data. The memory system may also choose just to supply the addressed byte or halfword. This may be desirable in order to save power or to simplify the decode logic.

When a byte or halfword write occurs (e.g., *STRH*), the core will broadcast the byte or halfword across the whole of the bus. The memory system must then decode $A[1:0]$ to enable writing only to the addressed byte or halfword.

One way of implementing the byte decode in a DRAM system is to separate the 32-bit wide block of DRAM into four byte wide banks, and generate the column address strobes independently as shown in [Figure 6.7](#).

When the processor is configured for Little Endian operation, byte 0 of the memory system should be connected to data lines 7 through 0 ($D[7:0]$) and strobed by $nCAS0$. $nCAS1$ drives the bank connected to data lines 15 through 8, and so on. This has the added advantage of reducing the load on each column strobe driver, which improves the precision of this time-critical signal.

In the Big Endian case, byte 0 of the memory system should be connected to data lines 31 through 24.

6.5 Instruction Fetch

ARM7TDMI will perform 32- or 16-bit instruction fetches depending on whether the processor is in ARM or THUMB state. The processor state may be determined externally by the value of the TBIT signal. When this is LOW, the processor is in ARM state and 32-bit instructions are fetched. When TBIT is HIGH, the processor is in THUMB state and 16-bit instructions are fetched. The size of the data being fetched is also indicated on the $MAS[1:0]$ bits, as described in Section 6.4, “Data Transfer Size”.

When the processor is in ARM state, 32-bit instructions are fetched on $D[31:0]$. When the processor is in THUMB state, 16-bit instructions are fetched from either the upper, $D[31:16]$, or the lower $D[15:0]$ half of the bus. This is determined by the endian configuration of the memory

system, as configured by the BIGEND input, and the state of A[1]. [Table 6.2](#) shows which half of the data bus is sampled in the different configurations.

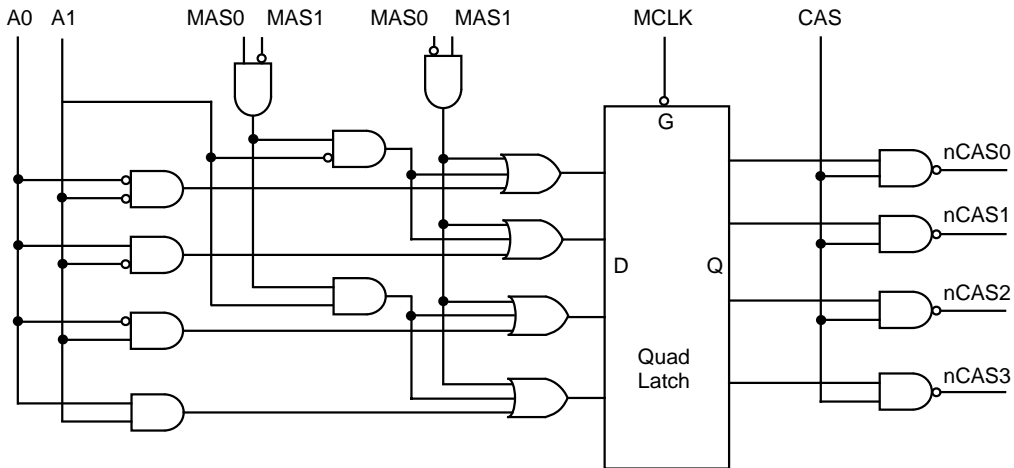
Table 6.2 Endian Configuration Effect on Instruction Position

	Little Endian BIGEND = 0	Big Endian BIGEND = 1
A[1] = 0	D[15:0]	D[31:16]
A[1] = 1	D[31:16]	D[15:0]

When a 16-bit instruction is fetched, the core ignores the unused half of the data bus.

[Table 6.2](#) describes instructions fetched from the bidirectional data bus (i.e. BUSEN is LOW). When the unidirectional data buses are in use (i.e. BUSEN is HIGH), data will be fetched from the corresponding half of the DIN[31:0] bus.

Figure 6.7 Decoding Byte Accesses to Memory



6.6 Memory Management

The core address bus may be processed by an address translation unit before being presented to the memory, and the core is capable of running a virtual memory system. The ABORT input to the processor may be used by the memory manager to inform the core of page faults. Various other signals enable different page protection levels to be supported:

- nRW can be used by the memory manager to protect pages from being written to.
- nTRANS indicates whether the processor is in user or a privileged mode, and may be used to protect system pages from the user, or to support completely separate mappings for the system and the user.

Address translation will normally only be necessary on an N-cycle, and this fact may be exploited to reduce power consumption in the memory manager and avoid the translation delay at other times. The times when translation is necessary can be deduced by keeping track of the cycle types that the processor uses.

6.7 Locked Operations

The ARM instruction set includes a data swap (*SWP*) instruction that allows the contents of a memory location to be swapped with the contents of a processor register. This instruction is implemented as an uninterruptable pair of accesses; the first access reads the contents of the memory, and the second writes the register data to the memory. These accesses must be treated as a contiguous operation by the memory controller to prevent another device from changing the affected memory location before the swap is completed. The core drives the LOCK signal HIGH for the duration of the swap operation to warn the memory controller not to give the memory to another device.

6.8 Stretching Access Times

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

Either MCLK can be stretched before it is applied to the core, or the nWAIT input can be used together with a free-running MCLK. Taking nWAIT LOW has the same effect as stretching the LOW period of MCLK, and nWAIT must only change when MCLK is LOW.

The core does not contain any dynamic logic which relies upon regular clocking to maintain its internal state. Therefore there is no limit upon the maximum period for which MCLK may be stretched, or nWAIT held LOW.

6.9 ARM7TDMI Data Bus

To ease the connection of the core to subword sized memory systems, input data and instructions may be latched on a byte by byte basis. This is achieved by use of the BL[3:0] input signals where BL[3] controls the latching of the data present on D[31:24] of the data bus and so on.

In a memory system containing word wide memory only, BL[3:0] may be tied HIGH. For subword wide memory systems, BL[3:0] are used to latch the data as it is read out of memory. For example, a word access to halfword wide memory must take place in two memory cycles. In the first cycle, the data for D[15:0] is obtained from the memory and latched into the processor on the falling edge of MCLK when BL[1:0] are both HIGH. In the second cycle, the data for D[31:16] is latched into the processor on the falling edge of MCLK when BL[3:2] are both HIGH.

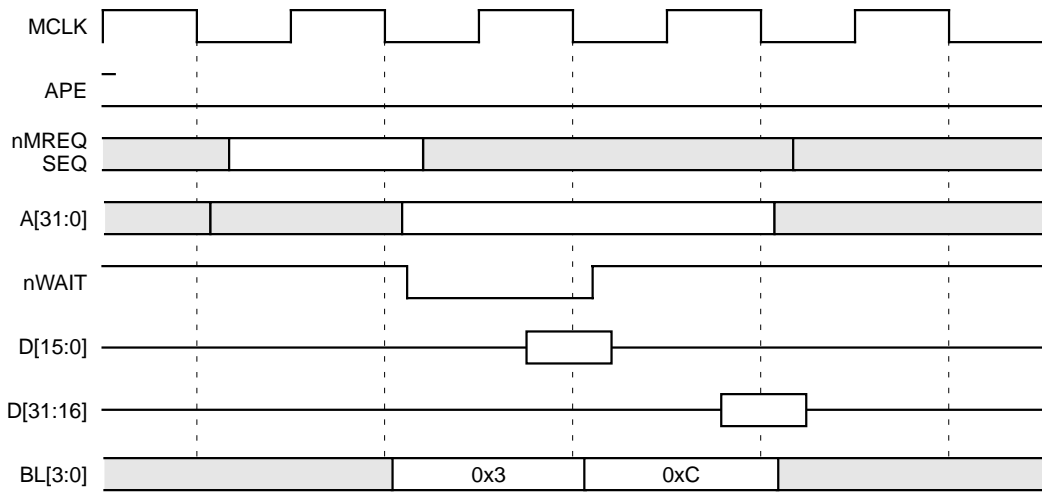
A memory access like this is shown in [Figure 6.8](#). Here, a word access is performed from halfword wide memory in two cycles. In the first, the data read is applied to the lower half of the bus, in the second cycle the read data is applied to the upper half of the bus. Since two memory cycles were required, nWAIT is used to stretch the internal processor clock. However, nWAIT does not effect the operation of the data latches. In this way, data may be extracted from memory word, halfword or byte

at a time, and the memory may have as many wait states as required. In any multicycle memory access, nWAIT is held LOW until the final quantum of data is latched.

In this example, BL[3:0] were driven to value 0x3 in the first cycle so that only the latches on D[15:0] were opened. In fact, BL[3:0] could have been driven to value 0xF and all the latches opened. Since in the second cycle, the latches on D[31:16] were written with the correct data, this would not have effected the processor's operation.

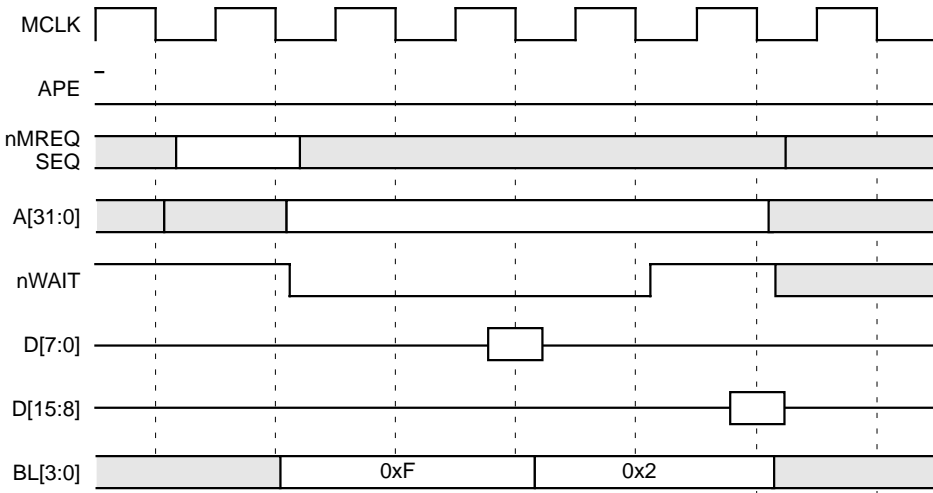
Note: BL[3:0] should all be HIGH during store cycles.

Figure 6.8 Memory Access



As a further example, a halfword load from 2-wait state byte-wide memory is shown in [Figure 6.9](#). Here, each memory access takes two cycles. In the first, access, BL[3:0] are driven to value 0xF. The correct data is latched from D[7:0] while unknown data is latched from D[31:8]. In the second access, the byte for D[15:8] is latched and so the halfword on D[15:0] has been correctly read from the memory. The fact that internally D[31:16] are unknown does not matter because internally the processor will extract only the halfword it is interested in.

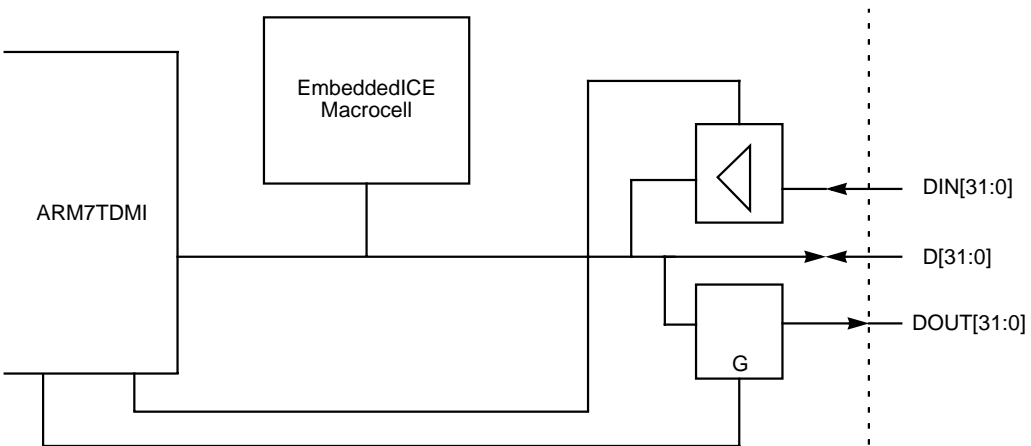
Figure 6.9 Two Cycle Memory Access



6.10 External Data Bus

The core has a bidirectional data bus, D[31:0]. However, since some ASIC design methodologies prohibit the use of bidirectional buses, unidirectional data in, DIN[31:0], and data out, DOUT[31:0], buses are also provided. The logical arrangement of these buses is shown in [Figure 6.10](#).

Figure 6.10 ARM7TDMI External Bus Arrangement



When the bidirectional data bus is being used, the unidirectional buses must be disabled by driving BUSEN LOW. The timing of the bus for three cycles, load-store-load, is shown in [Figure 6.11](#).

Figure 6.11 Bidirectional Bus Timing

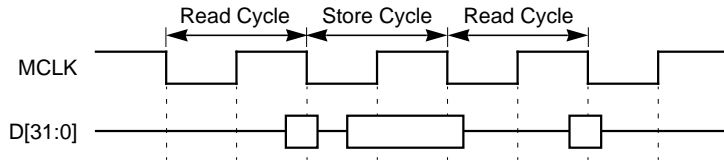
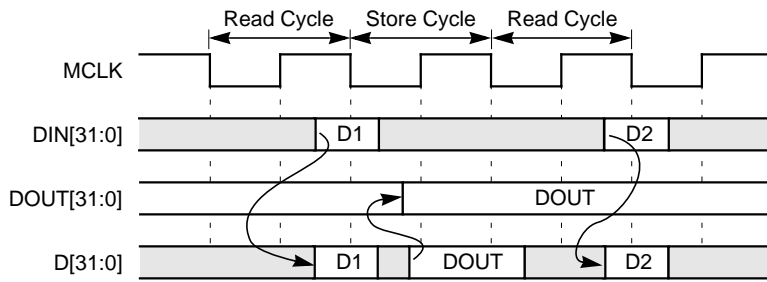


Figure 6.12 Unidirectional Bus Timing



6.10.1 The Unidirectional Data Bus

When the unidirectional data buses are being used, (i.e. when BUSEN is HIGH), the bidirectional bus, D[31:0], must be left unconnected.

When BUSEN is HIGH, all instructions and input data are presented on the input data bus, DIN[31:0]. The timing of this data is similar to that of the bidirectional bus when in input mode. The setup and hold of the data must occur on the falling edge of MCLK. For the exact timing requirements refer to *CW001007 ARM7TDMI Microprocessor Core Datasheet*.

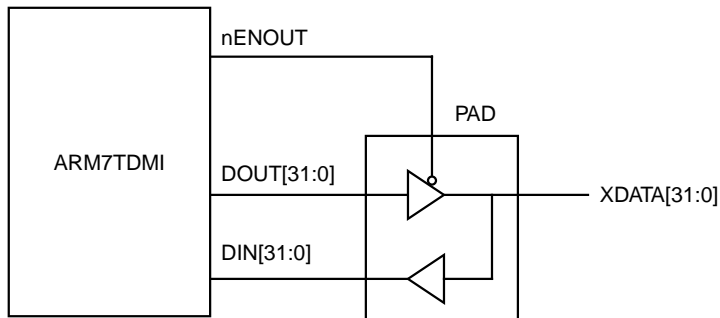
In this configuration, all output data is presented on DOUT[31:0]. The value on this bus only changes when the processor performs a store cycle. Again, the timing of the data is similar to that of the bidirectional data bus. The value on DOUT[31:0] changes on the falling edge of MCLK.

The bus timing of a read-write-read cycle combination is shown in [Figure 6.12](#).

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

Typically, the unidirectional buses would be used internally in ASIC embedded applications. Externally, most systems still require a bidirectional data bus to interface to external memory. [Figure 6.13](#), shows how the unidirectional buses may be joined up at the pads of an ASIC to connect to an external bidirectional bus.

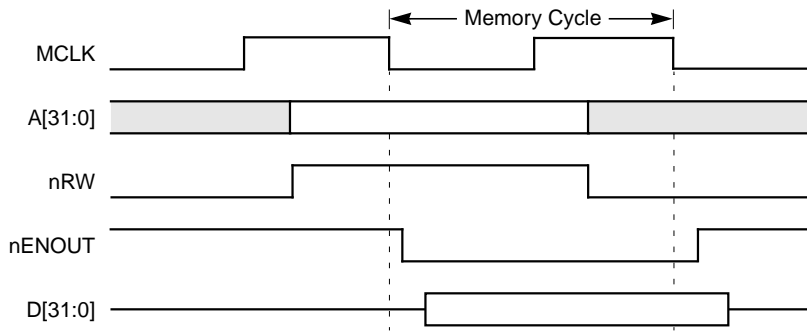
Figure 6.13 External Connection of Unidirectional Buses



6.10.2 Bidirectional Data Bus

The core has a bidirectional data bus, `D[31:0]`. Most of the time, the core reads from memory and so this bus is configured to input. During write cycles however, the core must output data. During phase 2 of the previous cycle, the signal `nRW` is driven HIGH to indicate a write cycle. During the actual cycle, `nENOUT` is driven LOW to indicate that the core is driving `D[31:0]` as an output. [Figure 6.14](#) shows this bus timing (`DBE` has been tied HIGH in this example). [Figure 6.15](#) shows the circuit which exists in the core for controlling exactly when the external bus is driven out.

Figure 6.14 Data Write Bus Cycle



The core macrocell has an additional bus control signal, nENIN, which allows the external system to manually 3-state the bus. In the simplest systems, nENIN can be tied LOW and nENOUT can be ignored. However, in many applications when the external data bus is a shared resource, greater control may be required. In this situation, nENIN can be used to delay when the external bus is driven. Note that for backwards compatibility, DBE is also included. At the macrocell level, DBE and nENIN have almost identical functionality and in most applications one can be tied off.

[Section 6.10.3, “Example System: The ARM7TDMI Test Chip,”](#) describes how the core may be interfaced to an external data bus, using the ARM7TDMI test chip as an example.

The core has another output control signal called TBE. This signal is normally only used during test and must be tied HIGH when not in use. When driven LOW, TBE forces all 3-state outputs to HIGH impedance. It is as if both DBE and ABE have been driven LOW, causing the data bus, the address bus, and all other signals normally controlled by ABE to become high impedance. Note, however, that there is no scan cell on TBE. Thus, TBE is completely independent of scan data and may be used to put the outputs into a high impedance state while scan testing takes place.

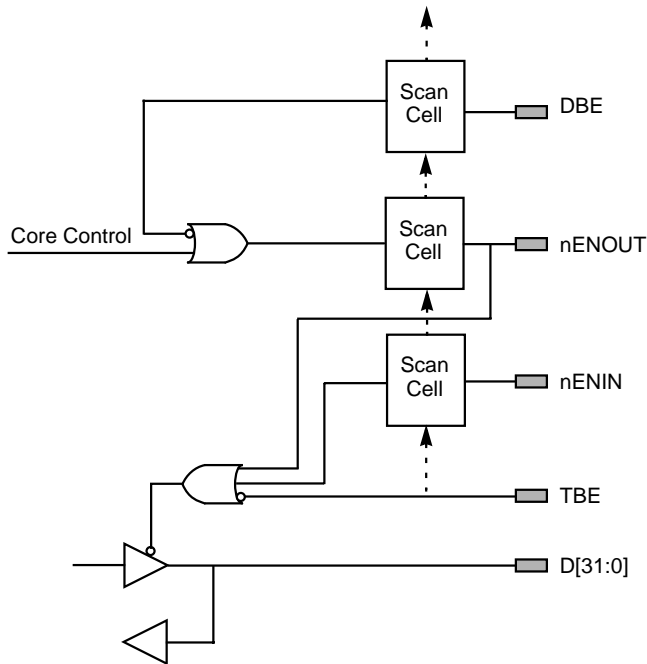
[Table 6.3](#) lists the 3-state control of the core outputs.

Only signals with a ✓ in the ABE, DBE or TBE column can be driven to the high impedance state.

Table 6.3 Output Enable Control Summary

ARM7TDMI Output	ABE	DBE	TBE
A[31:0]	✓		✓
D[31:0]		✓	
nRW	✓		✓
LOCK	✓		✓
MAS[1:0]	✓		✓
nOPC	✓		✓
nTRANS	✓		✓
DBGACK			
ECLK			
nCPI			
nENOUT			
nEXEC			
nM[4:0]			
TBIT			
nMREQ			
SDOUTBS			
SDOUTDATA			
SEQ			
DOUT[31:0]			

Figure 6.15 ARM7TDMI Data Bus Control Circuit



6.10.3 Example System: The ARM7TDMI Test Chip

Connecting the core data bus, D[31:0], to an external shared bus requires some simple additional logic. This will vary from application to application. As an example, the following describes how the core macrocell was connected to the bidirectional data bus pads of the ARM7TDMI test chip.

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

All this can be achieved using a simple nonoverlapping clock generator. The actual circuit implemented in the ARM7TDMI test chip is shown in [Figure 6.16](#). Note that at the core level, TBE and DBE are tied HIGH (inactive). This is because in a packaged part, there is no need to ever

manually force the internal buses into a high impedance state. Note also that at the pad level, the signal EDBE is factored into the bus control logic. This allows the external memory controller to arbitrate the bus and asynchronously disable ARM7TDMI test chip if required.

Figure 6.16 The ARM7TDMI Test Chip Data Bus Circuit

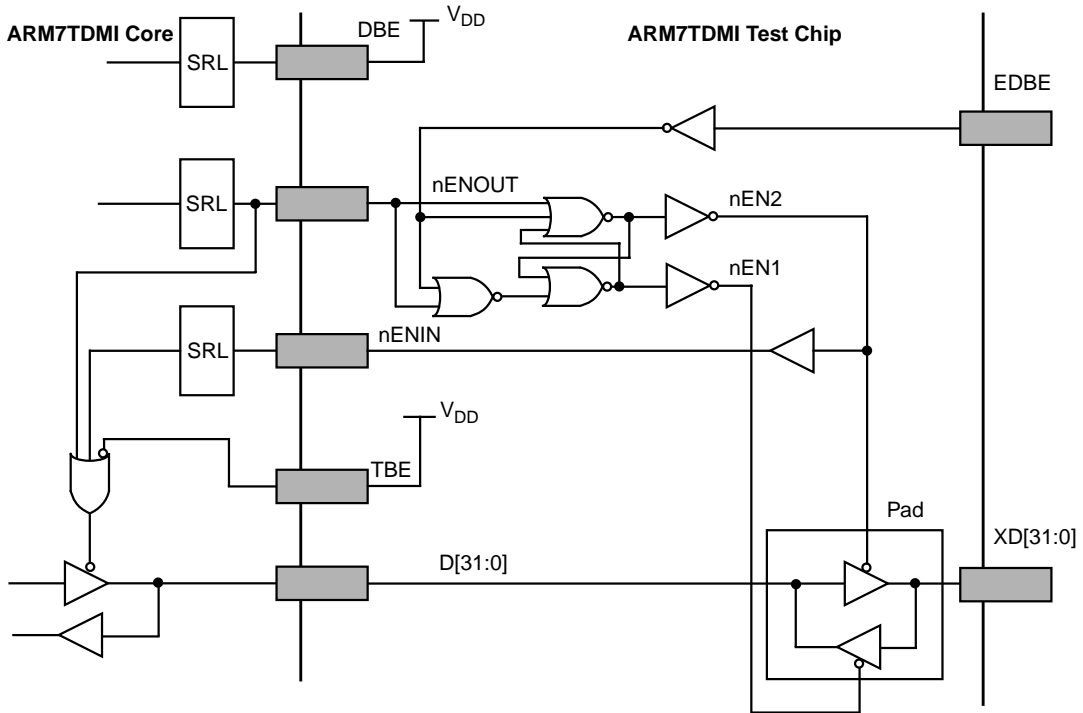
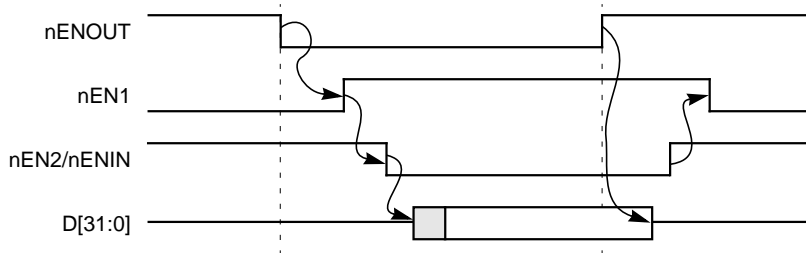


Figure 6.17 shows how the various control signals interact. Under normal conditions, when the data bus is configured as input, nENOUT is HIGH, nEN1 is LOW, and nEN2/nENIN is HIGH. Thus the pads drive XD[31:0] onto D[31:0].

When a write cycle occurs, nRW is driven HIGH to indicate a write during phase 2 of the previous cycle, (ie, with the address). During phase 1 of the actual cycle, nENOUT is driven LOW to indicate that the core is about to drive the bus. The falling edge of this signal makes nEN1 go HIGH, which disables the input half pad from driving D[31:0]. This in turn makes nEN2 go LOW, which enables the output half of the pad so that the core is now driving the external data bus, XD[31:0]. nEN2 is then buffered and driven back into the core on nENIN, so that finally the core

macrocell drives D[31:0]. The delay between all the signals ensures that there is no clash on the data bus as it changes direction from input to output.

Figure 6.17 Data Bus Control Signal Timing



When the bus changes direction at the end of the cycle, the various control signals switch the other way. Again, the nonoverlap ensures that there is never a bus clash. This time, nENOUT is driven HIGH to denote that the core no longer needs to drive the bus and the core's output is immediately switched off. This causes nEN2 to disable the output half of the pad which in turn causes nEN1 to switch on the input half. Thus, the bus is back to its original input configuration.

Note that the data out time of the core is not directly determined by nENOUT and nENIN, and so delaying exactly when the bus is driven will not affect the propagation delay. Please refer to *CW001007 ARM7TDMI Microprocessor Core Datasheet* for timing details.

Chapter 7

Coprocessor Interface

This chapter describes the ARM7TDMI coprocessor interface and contains the following sections:

- [Section 7.1, “Overview,” page 7-1](#)
 - [Section 7.2, “Interface Signals,” page 7-1](#)
 - [Section 7.3, “Register Transfer Cycle,” page 7-3](#)
 - [Section 7.4, “Privileged Instructions,” page 7-4](#)
 - [Section 7.5, “Idempotency,” page 7-4](#)
 - [Section 7.6, “Undefined Instructions,” page 7-5](#)
-

7.1 Overview

The functionality of the core instruction set may be extended by the addition of up to 16 external coprocessors. When the coprocessor is not present, instructions intended for it will trap, and suitable software may be installed to emulate its functions. Adding the coprocessor will then increase the system performance in a software compatible way. Note that some coprocessor numbers have already been assigned. Contact ARM Ltd. for up-to-date information.

7.2 Interface Signals

Three dedicated signals control the coprocessor interface, nCPI, CPA and CPB. The CPA and CPB inputs should be driven HIGH except when they are being used for handshaking.

7.2.1 Coprocessor Present/Absent

The core takes nCPI LOW whenever it starts to execute a coprocessor (or undefined) instruction. (This will not happen if the instruction fails to be executed because of the condition codes.) Each coprocessor will have a copy of the instruction, and can inspect the CP# field to see which coprocessor it is for. Every coprocessor in a system must have a unique number and if that number matches the contents of the CP# field the coprocessor should drive the CPA (coprocessor absent) line LOW. If no coprocessor has a number which matches the CP# field, CPA and CPB will remain HIGH, and the core will take the undefined instruction trap. Otherwise the core observes the CPA line going LOW, and waits until the coprocessor is not busy.

7.2.2 Busy (Waiting)

If CPA goes LOW, the core will watch the CPB (coprocessor busy) line. Only the coprocessor which is driving CPA LOW is allowed to drive CPB LOW, and it should do so when it is ready to complete the instruction. The core will busy-wait while CPB is HIGH, unless an enabled interrupt occurs, in which case it will break off from the coprocessor handshake to process the interrupt. When the core returns from processing the interrupt to retry the coprocessor instruction.

When CPB goes LOW, the instruction continues to completion. This will involve data transfers taking place between the coprocessor and either the core or memory, except in the case of coprocessor data operations which complete immediately when the coprocessor ceases to be busy.

All three interface signals are sampled by both the core and the coprocessor(s) on the rising edge of MCLK. If all three are LOW, the instruction is committed to execution, and if transfers are involved they will start on the next cycle. If nCPI has gone HIGH after being LOW, and before the instruction is committed, the core has broken off from the busy-wait state to service an interrupt. The instruction may be restarted later, but other coprocessor instructions may come sooner, and the instruction should be discarded.

7.2.3 Pipeline Following

In order to respond correctly when a coprocessor instruction arises, each coprocessor must have a copy of the instruction. All core instructions are fetched from memory using the main data bus, and coprocessors are connected to this bus, so they can keep copies of all instructions as they go into the core pipeline. The nOPC signal indicates when an instruction fetch is taking place, and MCLK gives the timing of the transfer, so these may be used together to load an instruction pipeline within the coprocessor.

7.2.4 Data Transfer Cycles

Once the coprocessor has gone not busy in a data transfer instruction, it must supply or accept data at the core bus rate (defined by MCLK). It can deduce the direction of transfer by inspection of the L bit in the instruction, but must only drive the bus when permitted to by DBE being HIGH. The coprocessor is responsible for determining the number of words to be transferred; the core will continue to increment the address by one word per transfer until the coprocessor tells it to stop. The termination condition is indicated by the coprocessor driving CPA and CPB HIGH.

There is no limit, in principle, to the number of words which one coprocessor data transfer can move, but by convention no coprocessor should allow more than 16 words in one instruction. More than this would worsen the worst case core interrupt latency, as the instruction is not interruptible once the transfers have commenced. At 16 words, this instruction is comparable with a block transfer of 16 registers, and therefore does not affect the worst case latency.

7.3 Register Transfer Cycle

The coprocessor register transfer cycle is the one case when the core requires the data bus without requiring the memory to be active. The memory system is informed that the bus is required by the core taking both nMREQ and SEQ HIGH. When the bus is free, DBE should be taken HIGH to allow the core or the coprocessor to drive the bus, and an MCLK cycle times the transfer.

7.4 Privileged Instructions

The coprocessor may restrict certain instructions for use in privileged modes only. To do this, the coprocessor will have to track the nTRANS output.

As an example of the use of this facility, consider the case of a floating-point coprocessor (FPU) in a multitasking system. The operating system could save all the floating-point registers on every task switch, but this is inefficient in a typical system where only one or two tasks will use floating-point operations. Instead, there could be a privileged instruction which turns the FPU on or off. When a task switch happens, the operating system can turn the FPU off without saving its registers. If the new task attempts an FPU operation, the FPU will appear to be absent, causing an undefined instruction trap. The operating system will then realize that the new task requires the FPU, so it will re-enable it and save FPU registers. The task can then use the FPU as normal. If, however, the new task never attempts an FPU operation (as will be the case for most tasks), the state saving overhead will have been avoided.

7.5 Idempotency

A consequence of the implementation of the coprocessor interface, with the interruptible busy-wait state, is that all instructions may be interrupted at any point up to the time when the coprocessor goes not busy. If so interrupted, the instruction will normally be restarted from the beginning after the interrupt has been processed. It is therefore essential that any action taken by the coprocessor before it goes not busy must be idempotent, i.e., must be repeatable with identical results.

For example, consider a FIX operation in a floating point coprocessor which returns the integer result to a core register. The coprocessor must stay busy while it performs the floating-point to fixed-point conversion, as the core will expect to receive the integer value on the cycle immediately following that where it goes not busy. The coprocessor must therefore preserve the original floating point value and not corrupt it during the conversion, because it will be required again if an interrupt arises during the busy period.

The coprocessor data operation class of instruction is not generally subject to idempotency considerations, as the processing activity can take place after the coprocessor goes not busy. There is no need for the core to be held up until the result is generated, because the result is confined to stay within the coprocessor.

7.6 Undefined Instructions

Undefined instructions are treated by the core as coprocessor instructions. All coprocessors must be absent (ie CPA and CPB must be HIGH) when an undefined instruction is presented. ARM7TDMI will then take the undefined instruction trap. Note that the coprocessor need only look at bit 27 of the instruction to differentiate undefined instructions (which all have a 0 in bit 27) from coprocessor instructions (which all have a 1 in bit 27).

Note that when in THUMB state, coprocessor instructions are not supported but undefined instructions are. Thus, all coprocessors must monitor the state of the TBIT output from the core. When the core is in THUMB state, coprocessors must appear absent (i.e., they must drive CPA and CPB HIGH) and the instructions seen on the data bus must be ignored. In this way, coprocessors will not erroneously execute THUMB instructions, and all undefined instructions will be handled correctly.

Chapter 8

Debug Interface

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

- [Section 8.1, “Overview,” page 8-1](#)
- [Section 8.2, “Debug Systems,” page 8-2](#)
- [Section 8.3, “Debug Interface Signals,” page 8-4](#)
- [Section 8.4, “Scan Chains and JTAG Interface,” page 8-7](#)
- [Section 8.5, “Reset,” page 8-11](#)
- [Section 8.6, “Pull-up Resistors,” page 8-11](#)
- [Section 8.7, “Instruction Register,” page 8-11](#)
- [Section 8.8, “Public Instructions,” page 8-12](#)
- [Section 8.9, “Test Data Registers,” page 8-16](#)
- [Section 8.10, “ARM7TDMI Core Clocks,” page 8-24](#)
- [Section 8.11, “Determining the Core and System State,” page 8-25](#)
- [Section 8.12, “PC Behavior During Debug,” page 8-30](#)
- [Section 8.13, “Priorities/Exceptions,” page 8-33](#)
- [Section 8.14, “Scan Interface Timing,” page 8-34](#)
- [Section 8.15, “Debug Timing,” page 8-38](#)

8.1 Overview

The core debug interface is based on the 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 core contains hardware extensions for advanced debugging features. These are intended to ease the user's development of application software, operating systems, and the hardware itself.

The debug extensions allow the core to be stopped either on a given instruction fetch (breakpoint) or data access (watchpoint), or asynchronously by a debug request. When this happens, the core is said to be in *debug state*. At this point, the core's internal state and the system's external state may be examined. Once examination is complete, the core and system state may be restored and program execution resumed.

The core is forced into debug state either by a request on one of the external debug interface signals, or by an internal functional unit known as the EmbeddedICE macrocell. Once in debug state, the core isolates itself from the memory system. The core can then be examined while all other system activity continues as normal.

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

8.2 Debug Systems

The ARM7TDMI 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 core. Such a system typically has three parts:

1. The Debug Host

This is a computer, for example a PC, running a software debugger such as ARMSD. The debug host allows the user to issue high level commands such as "set breakpoint at location XX", or "examine the contents of memory from 0x0 to 0x100".

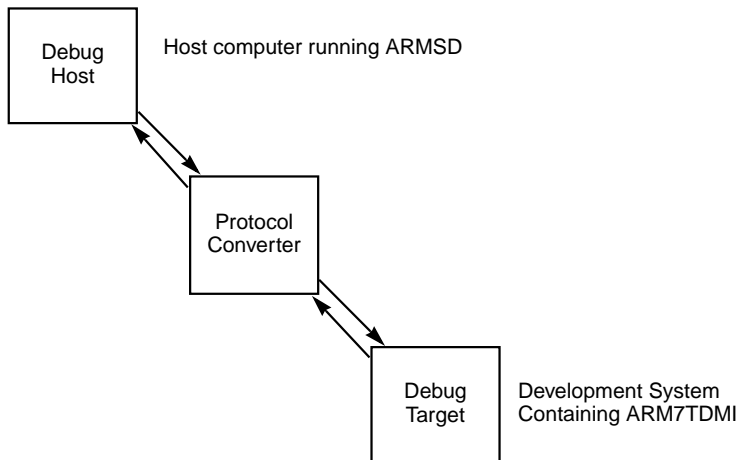
2. The Protocol Converter

The Debug Host will be connected to the core development system through an interface (an RS232, for example). The messages broadcast over this connection must be converted to the interface signals of the core, and this function is performed by the protocol converter.

3. ARM7TDMI core

The core, with hardware extensions to ease debugging, is the lowest level of the system. The debug extensions allow the user to stall the core from program execution, examine its internal state and the state of the memory system, and then resume program execution.

Figure 8.1 Typical Debug System



The anatomy of the core is shown in [Figure 8.3](#). The major blocks are:

ARM7TDMI – This is the CPU core, with hardware support for debug.

EmbeddedICE macrocell – This is a set of registers and comparators used to generate debug exceptions (e.g., breakpoints). This unit is described in [Chapter 9, "EmbeddedICE Macrocell"](#).

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

The Debug Host and the Protocol Converter are system dependent. The rest of this chapter describes the core's hardware debug extensions.

8.3 Debug Interface Signals

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

- BREAKPT and DBGRQ
with which the system requests that the core enter debug state.
- DBGACK
which the core uses to flag back to the system that it is in debug state.

8.3.1 Entry into Debug State

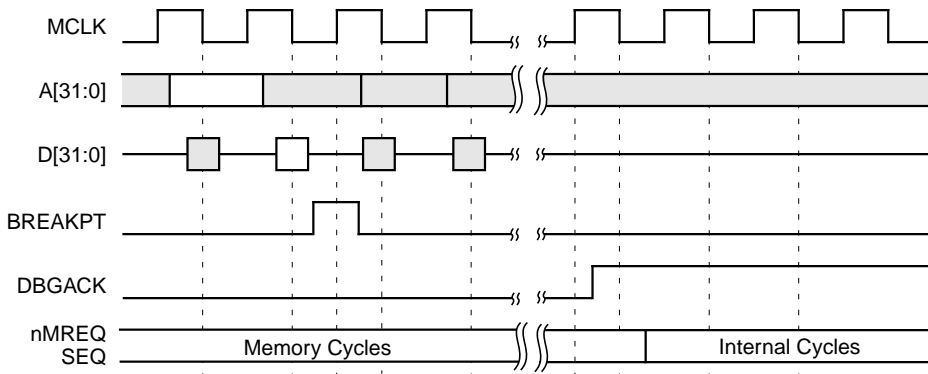
The core is forced into debug state after a breakpoint, watchpoint or debug request has occurred.

Conditions under which a breakpoint or watchpoint occur can be programmed using the EmbeddedICE macrocell. Alternatively, external logic can monitor the address and data bus, and flag breakpoints and watchpoints using the BREAKPT pin.

The timing is the same for externally generated breakpoints and watchpoints. Data must always be valid around the falling edge of MCLK. If this data is an instruction which generates a breakpoint, the BREAKPT signal must be HIGH on the next rising edge of MCLK. Similarly, if the data is for a load or store, this can be marked as a watchpoint by asserting BREAKPT on the next rising edge of MCLK.

When a breakpoint or watchpoint is generated, there may be a delay before the core enters debug state. When it does, the DBGACK signal is asserted in the HIGH phase of MCLK. The timing for an externally generated breakpoint is shown in [Figure 8.2](#).

Figure 8.2 Debug State Entry



8.3.1.1 Entry into Debug State on Breakpoint

After an instruction has generated a breakpoint, the core does not enter debug state immediately. Instructions are marked as being a breakpoint as they enter the core's instruction pipeline.

Thus the core only enters debug state when (and if) the instruction reaches the pipeline's execute stage.

A breakpoint instruction may not cause the core to enter debug state for one of two reasons:

- A branch precedes the breakpoint instruction.
When the branch is executed, the instruction pipeline is flushed and the breakpoint is cancelled.
- An exception has occurred.
Again, the instruction pipeline is flushed and the breakpoint is cancelled. However, the normal way to exit from an exception is to branch back to the instruction that would have executed next. This involves refilling the pipeline, and so the breakpoint can be reflagged.

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

Breakpoint instructions do not get executed: instead, the core enters debug state. Thus, when the internal state is examined, the state before the breakpoint instruction is seen. Once examination is complete, the breakpoint should be removed and program execution restarted from the previous breakpoint instruction.

8.3.1.2 Entry into Debug State on Watchpoint

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

Watchpoints can be thought of as being similar to data aborts. The difference is if a data abort occurs, although the instruction completes, all subsequent changes to the core's state are prevented. This allows the cause of the abort to be cured by the abort handler, and the instruction re-executed. This is not so in the case of a watchpoint. Here, the instruction completes and all changes to the core's state occur (i.e., load data is written into the destination registers, and base write back occurs). Thus the instruction does not need to be restarted.

Watchpoints are *always* taken. If an exception is pending when a watchpoint occurs, the core enters debug state in the mode of that exception.

8.3.1.3 Entry into Debug State on Debug Request

ARM7TDMI may also be forced into debug state on debug request. This can be done either through EmbeddedICE macrocell programming (see [Chapter 9, "EmbeddedICE Macrocell"](#)), or by the assertion of the DBGRQ signal. This signal is an asynchronous input and is thus synchronized by logic inside the core before it takes effect. Following synchronization, the core will normally enter debug state at the end of the current instruction. However, if the current instruction is a busy-waiting access to a coprocessor, the instruction terminates and the core enters debug state immediately (this is similar to the action of nIRQ and nFIQ).

8.3.1.4 Action of ARM7TDMI in Debug State

Once the core is in debug state, nMREQ and SEQ are forced to indicate internal cycles. This allows the rest of the memory system to ignore the core and function as normal. Since the rest of the system continues operation, the core must be forced to ignore aborts and interrupts.

The BIGEND signal should not be changed by the system during debug. If it changes, not only will there be a synchronization problem, but the programmer's view of the core will change without the debugger's knowledge. nRESET must also be held stable during debug. If the system applies reset to the core (ie. nRESET is driven LOW) then the core's state will change without the debugger's knowledge.

The BL[3:0] signals must remain HIGH while the core is clocked by DCLK in debug state to ensure all of the data in the scan cells is correctly latched by the internal logic.

When instructions are executed in debug state, the core outputs (except nMREQ and SEQ) will change asynchronously to the memory system. For example, every time a new instruction is scanned into the pipeline, the address bus will change. Although this is asynchronous it should not affect the system, since nMREQ and SEQ are forced to indicate internal cycles regardless of what the rest of ARM7TDMI is doing. The memory controller must be designed to ensure that this asynchronous behavior does not affect the rest of the system.

8.4 Scan Chains and JTAG Interface

There are three JTAG-style scan chains inside the core. These allow testing, debugging and EmbeddedICE macrocell programming. The scan chains are controlled from a JTAG-style TAP (Test Access Port) controller. For further details of the JTAG specification, please refer to IEEE Standard 1149.1 - 1990 "*Standard Test Access Port and Boundary-Scan Architecture*". In addition, support is provided for an optional fourth scan chain. This is intended to be used for an external boundary scan chain around the pads of a packaged device. The control signals provided for this scan chain are described later.

Note: The scan cells are not fully JTAG compliant. The following sections describe the limitations on their use.

8.4.1 Scan Limitations

The three scan paths are referred to as scan chain 0, 1 and 2: these are shown in [Figure 8.3 ARM7TDMI Scan Chain Arrangement](#).

8.4.1.1 Scan Chain 0

Scan chain 0 allows access to the entire periphery of the core, including the data bus. The scan chain functions allow interdevice testing (EXTEST) and serial testing of the core (INTEST).

The order of the scan chain (from SDINBS to SDOUTBS) is: data bus bits [0:31], the control signals, followed by the address bus bits [31:0].

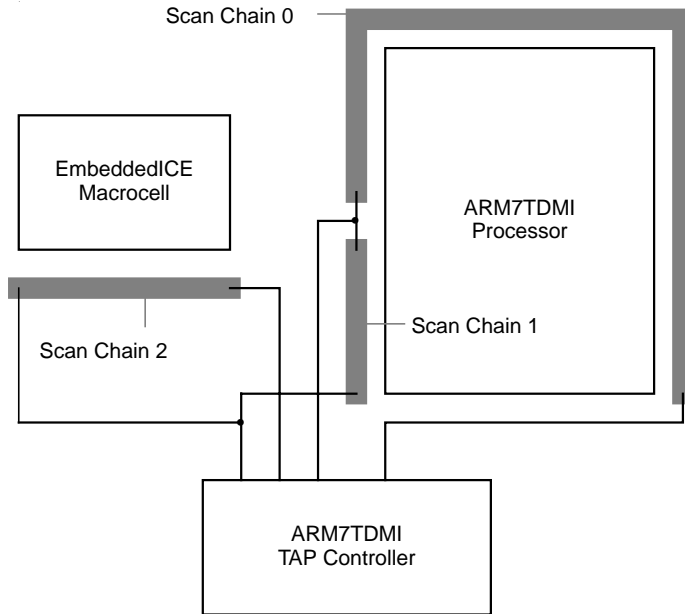
8.4.1.2 Scan Chain 1

Scan chain 1 is a subset of the signals that are accessible through scan chain 0. Access to the core's data bus D[31:0], and the BREAKPT signal is available serially. There are 33 bits in this scan chain, the order being (from serial data in to out): data bus bits 0 through 31, followed by BREAKPT.

8.4.1.3 Scan Chain 2

This scan chain simply allows access to the EmbeddedICE macrocell registers. Refer to [Chapter 9, "EmbeddedICE Macrocell"](#) for more detail.

Figure 8.3 ARM7TDMI Scan Chain Arrangement

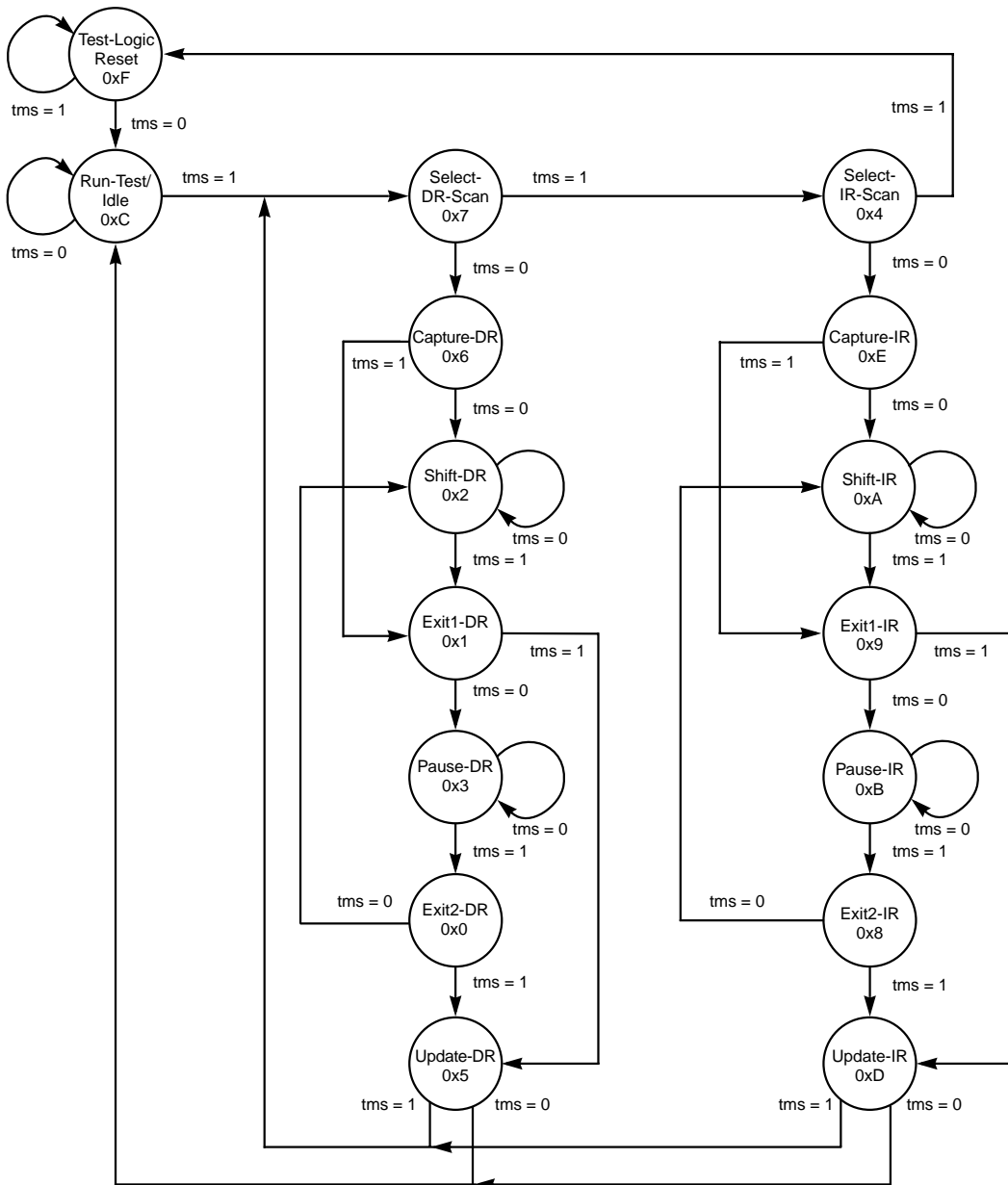


8.4.2 The JTAG State Machine

The process of serial test and debug is best explained in conjunction with the JTAG state machine. [Figure 8.4](#) shows the state transitions that occur in the TAP controller.

The state numbers are also shown on the diagram. These are output from the core on the TAPSM[3:0] bits.

Figure 8.4 Test Access Port (TAP) Controller State Transitions



8.5 Reset

The boundary scan interface includes a state machine controller (the TAP controller). In order to force the TAP controller into the correct state after power up of the device, a reset pulse must be applied to the nTRST signal. If the boundary scan interface is to be used, nTRST must be driven LOW, and then HIGH again. If the boundary scan interface is not to be used, the nTRST input should be driven by the same signal as nRESET. Note that a clock on TCK is not necessary to reset the device.

The action of reset is as follows:

1. System mode is selected (i.e., the boundary scan chain cells do *not* intercept any of the signals passing between the external system and the core).
2. The IDCODE instruction is selected. If the TAP controller is put into the Shift-DR state and TCK is pulsed, the contents of the ID register will be clocked out of TDO.

8.6 Pull-up Resistors

The IEEE 1149.1 standard effectively requires that TDI and TMS should have internal pull-up resistors. In order to minimize static current draw, these resistors are not fitted to the core. Accordingly, the 4 inputs to the test interface (the above TDO, TDI, TMS, and TCK) must all be driven to good logic levels to achieve normal circuit operation.

8.7 Instruction Register

The instruction register is 4 bits in length.

There is no parity bit. The fixed value loaded into the instruction register during the Capture-IR controller state is 0b0001.

8.8 Public Instructions

Table 8.1 lists the public instructions supported by the core.

Table 8.1 Public Instructions

Instruction	Code
EXTEST	0b0000
SCAN_N	0b0010
INTEST	0b1100
IDCODE	0b1110
BYPASS	0b1111
CLAMP	0b0101
HIGHZ	0b0111
CLAMPZ	0b1001
SAMPLE/PRELOAD	0b0011
RESTART	0b0100

In the descriptions that follow, TDI and TMS are sampled on the rising edge of TCK and all output transitions on TDO occur as a result of the falling edge of TCK.

8.8.1 EXTEST (0b0000)

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

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

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

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

8.8.2 SCAN_N (0b0010)

This instruction connects the Scan Path Select Register between TDI and TDO. During the Capture-DR state, the fixed value 0b1000 is loaded into the register. During the Shift-DR state, the ID number of the desired scan path is shifted into the scan path select register. In the Update-DR state, the scan register of the selected scan chain is connected between TDI and TDO, and remains connected until a subsequent `SCAN_N` instruction is issued. On reset, scan chain 3 is selected by default. The scan path select register is 4 bits long in this implementation, although no finite length is specified.

8.8.3 INTEST (0b1100)

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

The `INTEST` instruction connects the selected scan chain between TDI and TDO.

When the instruction register is loaded with the `INTEST` instruction, all the scan cells are placed in their test mode of operation.

In the Capture-DR state, the value of the data applied from the core logic to the output scan cells, and the value of the data applied from the system logic to the input scan cells is captured.

In the Shift-DR state, the previously captured test data is shifted out of the scan chain using the TDO signal, while new test data is shifted in through the TDI signal.

Single-step operation is possible using the `INTEST` instruction.

8.8.4 IDCODE (0b1110)

The `IDCODE` instruction connects the device identification register (or ID register) between TDI and TDO. The ID register is a 32-bit register that allows the manufacturer, part number and version of a component to be determined through the TAP. See [Section 8.9.2, “ARM7TDMI Device Identification \(ID\) Code Register,”](#) for the details of the ID register format.

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

In the Capture-DR state, the device identification code is captured by the ID register. In the Shift-DR state, the previously captured device identification code is shifted out of the ID register through the TDO signal, while data is shifted in using the TDI signal into the ID register. In the Update-DR state, the ID register is unaffected.

8.8.5 BYPASS (0b1111)

The `BYPASS` instruction connects a 1 bit shift register (the `BYPASS` register) between TDI and TDO.

When the `BYPASS` instruction is loaded into the instruction register, all the scan cells are placed in their normal (system) mode of operation. This instruction has no effect on the system pins.

In the Capture-DR state, a logic 0 is captured by the bypass register. In the Shift-DR state, test data is shifted into the bypass register using TDI and out using TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the Update-DR state. Note that all unused instruction codes default to the `BYPASS` instruction.

8.8.6 CLAMP (0b0101)

This instruction connects a 1 bit shift register (the `BYPASS` register) between TDI and TDO.

When the `CLAMP` instruction is loaded into the instruction register, the state of all the output signals is defined by the values previously loaded into the currently loaded scan chain.

Note: This instruction should only be used when scan chain 0 is the currently selected scan chain.

In the Capture-DR state, a logic 0 is captured by the bypass register. In the Shift-DR state, test data is shifted into the bypass register using TDI and out using TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the Update-DR state.

8.8.7 HIGHZ (0b0111)

This instruction connects a 1-bit shift register (the BYPASS register) between TDI and TDO.

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

In the Capture-DR state, a logic 0 is captured by the bypass register. In the Shift-DR state, test data is shifted into the bypass register using TDI and out using TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the Update-DR state.

8.8.8 CLAMPZ (0b1001)

This instruction connects a 1-bit shift register (the BYPASS register) between TDI and TDO.

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

In the Capture-DR state, a logic 0 is captured by the bypass register. In the Shift-DR state, test data is shifted into the bypass register using TDI and out using TDO after a delay of one TCK cycle. Note that the first bit shifted out will be a zero. The bypass register is not affected in the Update-DR state.

8.8.9 SAMPLE/PRELOAD (0b0011)

This instruction is included for production test only, and should never be used.

8.8.10 RESTART (0b0100)

This instruction is used to restart the processor on exit from debug state. The `RESTART` instruction connects the bypass register between TDI and TDO and the TAP controller behaves as if the `BYPASS` instruction had been loaded. The processor will resynchronize back to the memory system once the Run-Test/Idle state is entered.

8.9 Test Data Registers

There are 6 test data registers which may be connected between TDI and TDO. They are: Bypass register, ID Code register, Scan Chain Select register, Scan chain 0, 1, or 2. These are now described in detail.

8.9.1 Bypass Register

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

Length – One bit.

Operating Mode – When the `BYPASS` instruction is the current instruction in the instruction register, serial data is transferred from TDI to TDO in the Shift-DR state with a delay of one TCK cycle.

There is no parallel output from the bypass register.

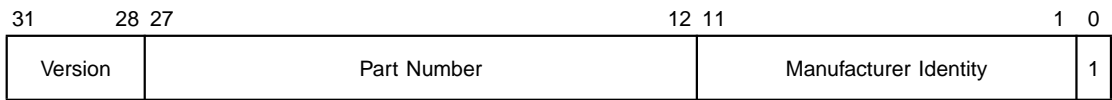
A logic 0 is loaded from the parallel input of the bypass register in the Capture-DR state.

8.9.2 ARM7TDMI Device Identification (ID) Code Register

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

Length – 32 bits. The format of the ID register is as follows:

Figure 8.5 ID Register Format



Please contact your supplier for the correct Device Identification Code.

For the CW001004 the value returned by this register is
0b0001.1111.0000.1111.0000.1111.0000.1111.

For the CW001007 this register is not implemented in the core, and will read back all zeroes, indicating that no valid ID is present. If you wish to implement an ID this must be done through interaction with the JTAG states signals accessible externally to the core.

Operating Mode: – When the `IDCODE` instruction is current, the ID register is selected as the serial path between TDI and TDO.

There is no parallel output from the ID register.

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

8.9.3 Instruction Register

Purpose – Changes the current TAP instruction.

Length – Four bits.

Operating mode – When in the Shift-IR state, the instruction register is selected as the serial path between TDI and TDO.

During the Capture-IR state, the value 0b0001 is loaded into this register. This is shifted out during Shift-IR (LSB first), while a new instruction is shifted in (LSB first). During the Update-IR state, the value in the instruction register becomes the current instruction. On reset, `IDCODE` becomes the current instruction.

8.9.4 Scan Chain Select Register

Purpose – Changes the current active scan chain.

Length – Four bits.

Operating mode – After `SCAN_N` has been selected as the current instruction, when in the Shift-DR state, the Scan Chain Select register is selected as the serial path between TDI and TDO.

During the Capture-DR state, the value `0b1000` is loaded into this register. This is shifted out during Shift-DR (LSB first), while a new value is shifted in (LSB 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 only changes when a `SCAN_N` instruction is executed, or a reset occurs. On reset, scan chain 3 is selected as the active scan chain.

The number of the currently selected scan chain is reflected on the `SCREG[3:0]` outputs. The TAP controller may be used to drive external scan chains in addition to those within the ARM7TDMI macrocell. The external scan chain must be assigned a number and control signals for it can be derived from `SCREG[3:0]`, `IR[3:0]`, `TAPSM[3:0]`, `TCK1` and `TCK2`.

The list of scan chain numbers allocated by ARM7TDMI are shown in [Table 8.2](#). An external scan chain may take any other number. The serial data stream to be applied to the external scan chain is made present on `SDINBS`, the serial data back from the scan chain must be presented to the TAP controller on the `SDOUTBS` input. The scan chain present between `SDINBS` and `SDOUTBS` will be connected between TDI and TDO whenever scan chain 3 is selected, or when any of the unassigned scan chain numbers is selected. If there is more than one external scan chain, a multiplexor must be built externally to apply the desired scan chain output to `SDOUTBS`. The multiplexor can be controlled by decoding `SCREG[3:0]`.

Table 8.2 Scan Chain Number Allocation

Scan Chain Number	Function
0	Macrocell scan test
1	Debug
2	EmbeddedICE macrocell programming
3	External boundary scan
4	Reserved
8	Reserved

8.9.5 Scan Chains 0, 1, and 2

These allow serial access to the core logic, and to EmbeddedICE macrocell for programming purposes. They are described in detail below.

8.9.5.1 Scan Chain 0 and 1

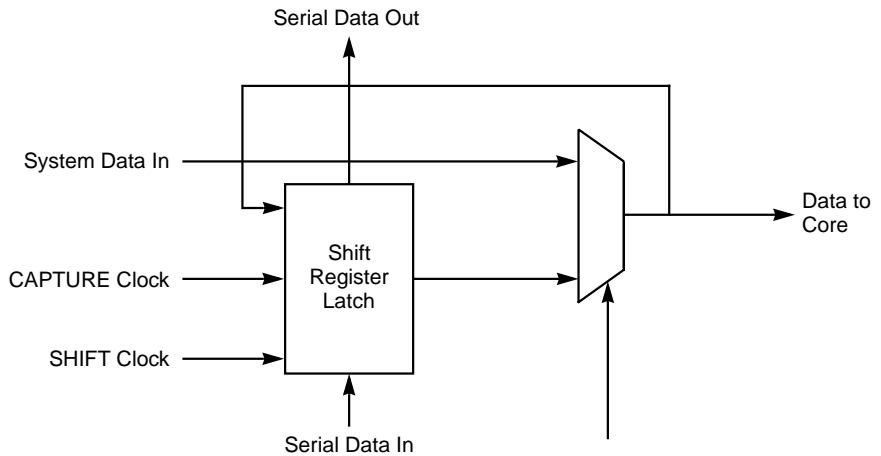
Purpose – Allows access to the processor core for test and debug.

Length – Scan chain 0 is 105 bits, Scan chain 1 is 33 bits.

Each scan chain cell is fairly simple, and consists of a serial register and a multiplexer. The scan cells perform two basic functions, capture and shift.

For input cells, the capture stage involves copying the value of the system input to the core into the serial register. During shift, this value is output serially. The value applied to the core from an input cell is either the system input or the contents of the serial register, and this is controlled by the multiplexer.

Figure 8.6 Input Scan Cell



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

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

There are three basic modes of operation of the scan chains, **INTEST**, **EXTEST** and **SYSTEM**, and these are selected by the various TAP controller instructions. In **SYSTEM** mode, the scan cells are idle. System data is applied to inputs, and core outputs are applied to the system. In **INTEST** mode, the core is internally tested. The data serially scanned in is applied to the core, and the resulting outputs are captured in the output cells and scanned out. In **EXTEST** mode, data is scanned onto the core's outputs and applied to the external system. System input data is captured in the input cells and then shifted out.

Note: The scan cells are not fully JTAG compliant in that they do not have an Update stage. Therefore, while data is being moved around the scan chain, the contents of the scan cell is not isolated from the output. Thus the output from the scan cell to the core or to the external system could change on every scan clock.

This does not affect the core since its internal state does not change until it is clocked. However, the rest of the system needs to be aware that every output could change asynchronously as data is moved around the scan chain. External logic must ensure that this does not harm the rest of the system.

8.9.5.2 Scan Chain 0

Scan chain 0 is intended primarily for interdevice testing (EXTEST), and testing the core (INTEST). Scan chain 0 is selected using the `SCAN_N` instruction: see [Section 8.8.2, “SCAN_N \(0b0010\)”](#).

INTEST allows serial testing of the core. The TAP Controller must be placed in INTEST mode after scan chain 0 has been selected. During Capture-DR, the current outputs from the core's logic are captured in the output cells. During Shift-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known stimuli to the inputs. During Run-Test/Idle, the core is clocked. Normally, the TAP controller should only spend 1 cycle in Run-Test/Idle. The whole operation may then be repeated.

For details of the core's clocks during test and debug, see [Section 8.10, “ARM7TDMI Core Clocks”](#).

EXTEST allows interdevice testing, useful for verifying the connections between devices on a circuit board. The TAP Controller must be placed in EXTEST mode after scan chain 0 has been selected. During Capture-DR, the current inputs to the core's logic from the system are captured in the input cells. During Shift-DR, this captured data is shifted out while a new serial test pattern is scanned in, thus applying known values on the core's outputs. During Update-DR, the value shifted into the data bus D[31:0] scan cells appears on the outputs. For all other outputs, the value appears as the data is shifted round. Note, during Run-Test/Idle, the core is not clocked. The operation may then be repeated.

[Table 8.4](#) lists the Scan Chain 0 bit order.

8.9.5.3 Scan Chain 1

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

Note that this scan chain is 33 bits long—32 bits for the data value, plus the scan cell on the BREAKPT core input. This 33rd bit serves four purposes:

1. Under normal INTEST test conditions, it allows a known value to be scanned into the BREAKPT input.
2. During EXTEST test conditions, the value applied to the BREAKPT input from the system can be captured.
3. While debugging, the value placed in the 33rd bit determines whether the core synchronizes back to system speed before executing the instruction. See [Section 8.12.5, “System Speed Access”](#) for further details.
4. After the core has entered debug state, the first time this bit is captured and scanned out, its value tells the debugger whether the core entered debug state due to a breakpoint (bit 33 LOW), or a watchpoint (bit 33 HIGH).

8.9.5.4 Scan Chain 2

Purpose – Allows EmbeddedICE macrocell's registers to be accessed. The order of the scan chain, from TDI to TDO is: read/write, register address bits 4 to 0, followed by data value bits 31 to 0. See [Figure 9.2](#).

Length – 38 bits.

To access this serial register, scan chain 2 must first be selected using the `SCAN_N` TAP controller instruction. The TAP controller must then be placed in INTEST mode. No action is taken 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 macrocell register to be accessed. During Update-DR, this register is either read or written depending on the value of bit 37 (0 = read). Refer to [Chapter 9, “EmbeddedICE Macrocell,”](#) for further details.

8.9.5.5 Scan Chain 3

Purpose – Allows ARM7TDMI to control an external boundary scan chain.

Length – User-defined length.

Scan chain 3 is provided so that an optional external boundary scan chain may be controlled through the core. Typically this would be used for a scan chain around the pad ring of a packaged device. The following control signals are provided which are generated only when scan chain 3 has been selected. These outputs are inactive at all other times.

DRIVEBS	This would be used to switch the scan cells from system mode to test mode. This signal is asserted whenever either the INTEST, EXTEST, CLAMP or CLAMPZ instruction is selected.
PCLKBS	This is an update clock, generated in the Update-DR state. Typically the value scanned into a chain would be transferred to the cell output on the rising edge of this signal.
ICAPCLKBS, ECAPCLKBS	These are capture clocks used to sample data into the scan cells during INTEST and EXTEST respectively. These clocks are generated in the Capture-DR state.
SHCLKBS, SHCLK2BS	These are nonoverlapping clocks generated in the Shift-DR state used to clock the master and slave element of the scan cells respectively. When the state machine is not in the Shift-DR state, both these clocks are LOW.
nHIGHZ	This signal may be used to drive the outputs of the scan cells to the high impedance state. This signal is driven LOW when the HIGHZ instruction is loaded into the instruction register, and HIGH at all other times.

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

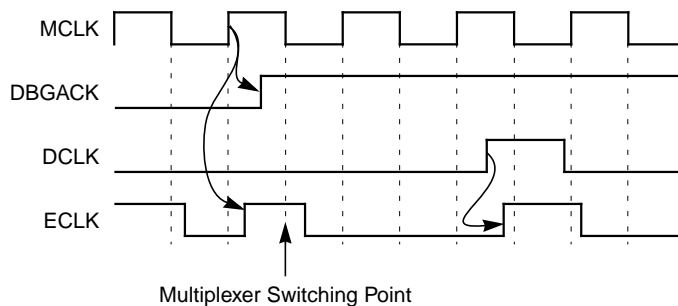
8.10 ARM7TDMI Core Clocks

The core has two clocks, the memory clock, MCLK, and an internally TCK generated clock, DCLK. During normal operation, the core is clocked by MCLK, and internal logic holds DCLK LOW. When the core is in the debug state, the core is clocked by DCLK under control of the TAP state machine, and MCLK may free run. The selected clock is output on the signal ECLK for use by the external system. Note that when the CPU core is being debugged and is running from DCLK, nWAIT has no effect.

8.10.1 Clock Switch During Debug

When the core enters debug state, it must switch from MCLK to DCLK. This is handled automatically by logic in the core. On entry to debug state, the core asserts DBGACK in the HIGH phase of MCLK. The switch between the two clocks occurs on the next falling edge of MCLK. This is shown in [Figure 8.7](#).

Figure 8.7 Clock Switching on Entry to Debug State



The core is forced to use DCLK as the primary clock until debugging is complete. On exit from debug, the core must be allowed to synchronize back to MCLK. This must be done in the following sequence. The final instruction of the debug sequence must be shifted into the data bus scan chain and clocked in by asserting DCLK. At this point, `BYPASS` must be clocked into the TAP instruction register. The core will now automatically resynchronize back to MCLK and start fetching instructions from memory at MCLK speed. Please refer also to [Section 8.11.3, "Exit from Debug State"](#).

8.11 Determining the Core and System State

When the core is in debug state, the core and system's state may be examined. This is done by forcing load and store multiples into the instruction pipeline.

Before the core and system state can be examined, the debugger must first determine whether the processor was in THUMB or ARM state when it entered debug. This is achieved by examining bit 4 of EmbeddedICE's Debug Status register. If this is HIGH, the core was in THUMB state when it entered debug.

8.11.1 Determining the Core's State

If 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. Once this is done, the debugger can always execute the same sequence of instructions to determine the processor's state.

To force the processor into ARM state, the following sequence of THUMB instructions should be executed 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: Since all THUMB instructions are only 16 bits long, the simplest course of action when shifting them into Scan Chain 1 is to repeat the instruction twice. For example, the encoding for `BX R0` is `0x4700`. Thus if `0x47004700` is shifted into scan chain 1, the debugger does not have to keep track of which half of the bus the processor expects to read the data from.

From this point on, the processor's state can be determined by the sequences of ARM instructions described below.

Once the processor is in ARM state, typically the first instruction executed would be:

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

This causes the contents of the registers to be made visible on the data bus. These values can then be sampled and shifted out.

Note: The above use of R0 as the base register for the *STM* instruction is for illustration only, any register could be used.

After determining the values in the current bank of registers, it may be desirable to access the banked registers. This can only be done by changing mode. Normally, a mode change may only occur if the core is already in a privileged mode. However, while in debug state, a mode change from any mode into any other mode may occur. Note that the debugger must restore the original mode before exiting debug state.

For example, assume that the debugger had been asked to return the state of the USER mode 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 visible before
ORR R0, 0x01                ; Select FIQ mode
MSR CPSR, R0                ; Enter FIQ mode
STM R0, {R8-R14}           ; Save banked FIQ registers
```

All these instructions are said to execute at *debug speed*. Debug speed is much slower than system speed since between each core clock, 33 scan clocks occur in order to shift in an instruction, or shift out data. Executing instructions more slowly than usual is fine for accessing the core's state since ARM7TDMI is fully static. However, this same method cannot be used for determining the state of the rest of the system.

While in debug state, only the following instructions may legally 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*

8.11.2 Determining System State

In order to meet the dynamic timing requirements of the memory system, any attempt to access system state must occur synchronously to it. Thus, the core must be forced to synchronize back to system speed. This is controlled by the 33rd bit of scan chain 1.

Any instruction may be placed in scan chain 1 with bit 33 (the BREAKPT bit) LOW. This instruction will then be executed at debug speed. To execute an instruction at system speed, the instruction prior to it must be scanned into scan chain 1 with bit 33 set HIGH.

After the system speed instruction has been scanned into the data bus and clocked into the pipeline, the `BYPASS` instruction must be loaded into the TAP controller. This will cause the core to automatically synchronize back to MCLK (the system clock), execute the instruction at system speed, and then re-enter debug state and switch itself back to the internally generated DCLK. When the instruction has completed, `DBGACK` will be HIGH and the core will have switched back to DCLK. At this point, `INTEST` can be selected in the TAP controller, and debugging can resume.

In order to determine that a system speed instruction has completed, the debugger must look at both `DBGACK` and `nMREQ`. In order to access memory, the core drives `nMREQ` LOW after it has synchronized back to system speed. This transition is used by the memory controller to arbitrate whether the core can have the bus in the next cycle. If the bus is not available, the core may have its clock stalled indefinitely. Therefore, the only way to tell that the memory access has completed, is to examine the state of both `nMREQ` and `DBGACK`. When both are HIGH, the access has completed. Usually, the debugger would be using `EmbeddedICE` macrocell to control debugging, and by reading `EmbeddedICE`'s status register, the state of `nMREQ` and `DBGACK` can be determined. Refer to [Chapter 9, "EmbeddedICE Macrocell,"](#) for more details.

By the use of system speed load multiples and debug speed store multiples, the state of the system's memory can be fed back to the debug host.

There are restrictions on which instructions may have the 33rd bit set. The only valid instructions on which to set this bit are loads, stores, load multiple and store multiple. See also [Section 8.11.3, “Exit from Debug State”](#). When the core returns to debug state after a system speed access, bit 33 of scan chain 1 is set HIGH. This gives the debugger information about why the core entered debug state the first time this scan chain is read.

8.11.3 Exit from Debug State

Leaving debug state involves restoring the core’s internal state, causing a branch to the next instruction to be executed, and synchronizing back to MCLK. After restoring internal state, a branch instruction must be loaded into the pipeline. See [Section 8.12, “PC Behavior During Debug,”](#) for details on calculating the branch.

Bit 33 of scan chain 1 is used to force the core to resynchronize back to MCLK. The second to last instruction of the debug sequence is scanned in with bit 33 set HIGH. The final instruction of the debug sequence is the branch, and this is scanned in with bit 33 LOW. The core is then clocked to load the branch into the pipeline. Now, the `RESTART` instruction is selected in the TAP controller. When the state machine enters the Run-Test/Idle state, the scan chain will revert back to system mode and clock resynchronization to MCLK will occur within the core. The core will then resume normal operation, fetching instructions from memory. This delay, until the state machine is in the Run-Test/Idle state, allows conditions to be setup in other devices in a multiprocessor system without taking immediate effect. Then, when the Run-Test/Idle state is entered, all the processors resume operation simultaneously.

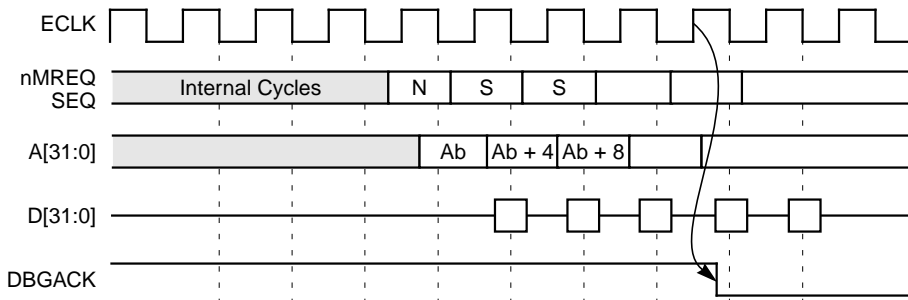
The function of `DBGACK` is to tell the rest of the system when the core is in debug state. This can be used to inhibit peripherals such as watchdog timers which have real-time characteristics. Also, `DBGACK` can be used to mask out memory accesses which are caused by the debugging process. For example, when the core enters debug state after a breakpoint, the instruction pipeline contains the breakpoint instruction plus two other instructions which have been prefetched. On entry to debug state, the pipeline is flushed. Therefore, on exit from debug state, the pipeline must be refilled to its previous state. Thus, because of the debugging process, more memory accesses occur than would normally

be expected. Any system peripheral which may be sensitive to the number of memory accesses can be inhibited through the use of DBGACK.

For example, imagine a fictitious peripheral that simply counts the number of memory cycles. This device should return the same answer after a program has been run both with and without debugging.

[Figure 8.8](#) shows the behavior of the core on exit from the debug state.

Figure 8.8 Debug Exit Sequence



It can be seen from [Figure 8.2](#) that the final memory access occurs in the cycle after DBGACK goes HIGH, and this is the point at which the cycle counter should be disabled. [Figure 8.8](#) shows that the first memory access that the cycle counter has not seen before occurs in the cycle after DBGACK goes LOW, and so this is the point at which the counter should be re-enabled.

Note that when a system speed access from debug state occurs, the core temporarily drops out of debug state, and so DBGACK can go LOW. If there are peripherals which are sensitive to the number of memory accesses, they must be led to believe that the core is still in debug state. By programming the EmbeddedICE macrocell control register, the value on DBGACK can be forced to be HIGH. See [Chapter 9, "EmbeddedICE Macrocell,"](#) for more details.

8.12 PC Behavior During Debug

In order that the core may be forced to branch back to the place at which program flow was interrupted by debug, the debugger must keep track of what happens to the PC. There are five cases: breakpoint, watchpoint, watchpoint when another exception occurs, debug request, and system speed access.

8.12.1 Breakpoint

Entry to the debug state from a breakpoint advances the PC by four addresses, or 16 bytes. Each instruction executed in debug state advances the PC by one address, or 4 bytes. The normal way to exit from debug state after a breakpoint is to remove the breakpoint, and branch back to the previous breakpoint address.

For example, if the 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, + 1 for the final branch). The following sequence shows the data scanned into scan chain 1. This is MSB first, and so the first digit is the value placed in the BREAKPT bit, followed by the instruction data.

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

Note that once in debug state, a minimum of two instructions must be executed before the branch, although these may both be NOPs (*MOV R0, R0*). For small branches, the final branch could be replaced with a subtract with the PC as the destination (*SUB PC, PC, #28* in the above example).

8.12.2 Watchpoints

Returning to program execution after entering debug state from a watchpoint is done in the same way as described above. Debug entry adds four addresses to the PC, and every instruction adds one address. The difference is that since the instruction that caused the watchpoint has executed, the program returns to the next instruction.

8.12.3 Watchpoint with Another Exception

If a watchpoint access simultaneously causes a data abort, the core will enter debug state in abort mode. Entry into debug is held off until the core has changed into abort mode, and fetched the instruction from the abort vector.

A similar sequence is followed when an interrupt, or any other exception, occurs during a watchpoint memory access. The core will enter debug state in the exception mode, and so the debugger must check to see whether this happened. The debugger can deduce whether an exception occurred by looking at the current and previous mode (in the CPSR and SPSR), and the value of the PC. If an exception did take place, the user should be given the choice of whether to service the exception before debugging.

Exiting debug state if an exception occurred is slightly different from the other cases. Here, entry to debug state causes the PC to be incremented by three addresses rather than four, and this must be taken into account in the return branch calculation. For example, suppose that an abort occurred on a watchpoint access and 10 instructions had been executed to determine this. The following sequence could be used to return to program execution.

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

This will force a branch back to the abort vector, causing the instruction at that location to be refetched and executed. Note that after the abort service routine, the instruction which caused the abort and watchpoint will be re-executed. This will cause the watchpoint to be generated and hence the core will enter debug state again.

8.12.4 Debug Request

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

For example, suppose that the user has invoked a debug request, and decides to return to program execution straight away. The following sequence could be used:

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

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

8.12.5 System Speed Access

If a system speed access is performed during debug state, the value of the PC is increased by three addresses. Since system speed instructions access the memory system, it is possible for aborts to take place. If an abort occurs during a system speed memory access, the core enters abort mode before returning to debug state.

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

8.12.6 Summary of Return Address Calculations

The calculation of the branch return address can be summarized as follows:

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

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

8.13 Priorities/Exceptions

Because the normal program flow is broken when a breakpoint or a debug request occurs, debug can be thought of as being another type of exception. Some of the interaction with other exceptions has been described above. This section summarizes the priorities.

8.13.1 Breakpoint with Prefetch Abort

When a breakpoint 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 which does not physically exist, and the returned data is therefore invalid. In such a case the operating system's normal action will be to swap in the page of memory and return to the previously invalid address. This time, when the instruction is fetched, and providing the breakpoint is activated (it may be data dependent), the core will enter debug state.

Thus the prefetch abort takes higher priority than the breakpoint.

8.13.2 Interrupts

When the core enters debug state, interrupts are automatically disabled. If interrupts are disabled during debug, the core will never be forced into an interrupt mode. Interrupts only have this effect on watchpoint accesses. They are ignored at all times on breakpoints.

If an interrupt was pending during the instruction prior to entering debug state, the core will enter debug state in the mode of the interrupt. Thus, on entry to debug state, the debugger cannot assume that the core will be in the expected mode of the user's program. It must check the PC, the CPSR and the SPSR to fully determine the reason for the exception.

Thus, debug takes higher priority than the interrupt, although the core remembers that an interrupt has occurred.

8.13.3 Data Aborts

As described above, when a data abort occurs on a watchpoint access, the core enters debug state in abort mode. Thus the watchpoint has higher priority than the abort, although, as in the case of interrupt, the core remembers that the abort happened.

8.14 Scan Interface Timing

Please be aware that all core AC timing values are technology dependent. To locate the values for your implementation, please refer to the appropriate *ARM7TDMI Microprocessor Core Datasheet*, available from LSI Logic.

Figure 8.9 Scan General Timing

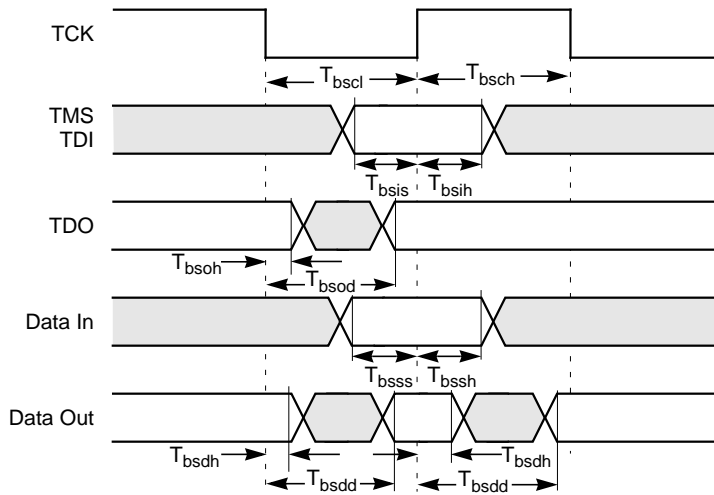


Table 8.3 ARM7TDMI Scan Interface Timing

Symbol	Parameter	Notes
Tbscl	TCK LOW period	
Tbsch	TCK HIGH period	
Tbsis	TDI,TMS setup to [TCr]	
Tbsih	TDI,TMS hold from [TCr]	
Tbsoh	TDO hold time	2
Tbsod	TCK falling edge to TDO valid	2
Tbsss	I/O signal setup to [TCr]	1
Tbssh	I/O signal hold from [TCr]	1
Tbsdh	data output hold time	2
Tbsdd	TCK falling edge to data output valid	2
Tbsr	Reset period	
Tbse	Output Enable time	2
Tbsz	Output Disable time	2

1. For correct data latching, the I/O signals (from the core and the pads) must be setup and held with respect to the rising edge of TCK in the Capture-DR state of the `INTEST` and `EXTEST` instructions.
2. Assumes that the data outputs are loaded with the AC test loads (see AC parameter specification).

Table 8.4 Scan Chain 0 Signal Order

No.	Signal	Type ¹	No.	Signal	Type ¹
1	D[0]	I/O	25	D[24]	I/O
2	D[1]	I/O	26	D[25]	I/O
3	D[2]	I/O	27	D[26]	I/O
4	D[3]	I/O	28	D[27]	I/O
5	D[4]	I/O	29	D[28]	I/O
6	D[5]	I/O	30	D[29]	I/O
7	D[6]	I/O	31	D[30]	I/O
8	D[7]	I/O	32	D[31]	I/O
9	D[8]	I/O	33	BREAKPT	I
10	D[9]	I/O	34	nENIN	I
11	D[10]	I/O	35	nENOUT	O
12	D[11]	I/O	36	LOCK	O
13	D[12]	I/O	37	BIGEND	I
14	D[13]	I/O	38	DBE	I
15	D[14]	I/O	39	MAS[0]	O
16	D[15]	I/O	40	MAS[1]	O
17	D[16]	I/O	41	BL[0]	I
18	D[17]	I/O	42	BL[1]	I
19	D[18]	I/O	43	BL[2]	I
20	D[19]	I/O	44	BL[3]	I
21	D[20]	I/O	45	DCTL ²	O
22	D[21]	I/O	46	nRW	O
23	D[22]	I/O	47	DBGACK	O
24	D[23]	I/O	48	CGENDBGACK	O
(Sheet 1 of 3)					

Table 8.4 Scan Chain 0 Signal Order (Cont.)

No.	Signal	Type ¹	No.	Signal	Type ¹
49	nFIQ	I	72	TBIT	O
50	nIRQ	I	73	nWAIT	I
51	nRESET	I	74	A[31]	O
52	ISYNC	I	75	A[30]	O
53	DBGREQ	I	76	A[29]	O
54	ABORT	I	77	A[28]	O
55	CPA	I	78	A[27]	O
56	nOPC	O	79	A[26]	O
57	IFEN	I	80	A[25]	O
58	nCPI	O	81	A[24]	O
59	nMREQ	O	82	A[23]	O
60	SEQ	O	83	A[22]	O
61	nTRANS	O	84	A[21]	O
62	CPB	I	85	A[20]	O
63	nM[4]	O	86	A[19]	O
64	nM[3]	O	87	A[18]	O
65	nM[2]	O	88	A[17]	O
66	nM[1]	O	89	A[16]	O
67	nM[0]	O	90	A[15]	O
68	nEXEC	O	91	A[14]	O
69	ALE	I	92	A[13]	O
70	ABE	I	93	A[12]	O
71	APE	I	94	A[11]	O

(Sheet 2 of 3)

Table 8.4 Scan Chain 0 Signal Order (Cont.)

No.	Signal	Type ¹	No.	Signal	Type ¹
95	A[10]	O	101	A[4]	O
96	A[9]	O	102	A[3]	O
97	A[8]	O	103	A[2]	O
98	A[7]	O	104	A[1]	O
99	A[6]	O	105	A[0]	O
100	A[5]	O	–	–	–

(Sheet 3 of 3)

1. I - Input, O - Output, I/O - Input/Output

2. DCTL is not described in this manual. DCTL is an output from the processor used to control the unidirectional data out latch, DOOUT [31:0]. DCTL is not visible from the periphery of ARM7TDMI.

8.15 Debug Timing

Please be aware that all core AC timing values are technology dependent. To locate the values for your implementation, please refer to the appropriate *ARM7TDMI Microprocessor Core Datasheet*, available from LSI Logic.

Table 8.5 ARM7TDMI Debug Interface Timing

Symbol	Parameter
Ttdbgd	TCK falling to DBGACK, DBGRQI changing
Ttpfd	TCKf to TAP outputs
Ttpfh	TAP outputs hold time from TCKf
Ttprd	TCKr to TAP outputs
Ttprh	TAP outputs hold time from TCKr
Ttckr	TCK to TCK1, TCK2 rising
Ttckf	TCK to TCK1, TCK2 falling

(Sheet 1 of 2)

Table 8.5 ARM7TDMI Debug Interface Timing (Cont.)

Symbol	Parameter
Tecapd	TCK to ECAPCLK changing
Tdckf	DCLK induced: TCKf to various outputs valid
Tdckfh	DCLK induced: Various outputs hold from TCKf
Tdckr	DCLK induced: TCKr to various outputs valid
Tdckrh	DCLK induced: Various outputs hold from TCKr
Ttrstd	nTRSTf to TAP outputs valid
Ttrsts	nTRSTr setup to TCKr
Tsdtf	SDOUTBS to TDO valid
Tclkbs	TCK to Boundary Scan Clocks
Tshbsr	TCK to SHCLKBS, SHCLK2BS rising
Tshbsf	TCK to SHCLKBS, SHCLK2BS falling
(Sheet 2 of 2)	

Chapter 9

EmbeddedICE

Macrocell

This chapter describes the ARM7TDMI EmbeddedICE macrocell.

This chapter contains the following sections:

- [Section 9.1, “Overview,” page 9-1](#)
 - [Section 9.2, “Watchpoint Registers,” page 9-3](#)
 - [Section 9.3, “Programming Breakpoints,” page 9-8](#)
 - [Section 9.4, “Programming Watchpoints,” page 9-10](#)
 - [Section 9.5, “Debug Control Register,” page 9-11](#)
 - [Section 9.6, “Debug Status Register,” page 9-12](#)
 - [Section 9.7, “Coupling Breakpoints and Watchpoints,” page 9-14](#)
 - [Section 9.8, “Disabling EmbeddedICE Macrocell,” page 9-17](#)
 - [Section 9.9, “EmbeddedICE Macrocell Timing,” page 9-17](#)
 - [Section 9.10, “Programming Restriction,” page 9-17](#)
 - [Section 9.11, “Debug Communication Channel,” page 9-18](#)
-

9.1 Overview

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

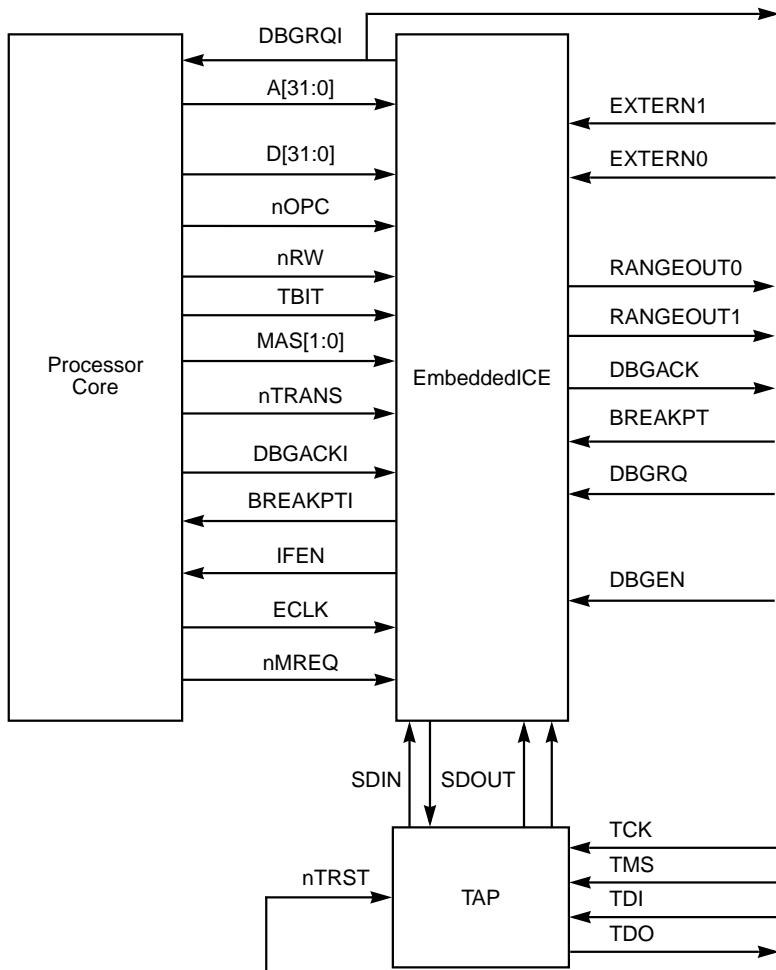
EmbeddedICE macrocell is programmed in a serial fashion using the TAP controller. It consists of two real-time watchpoint units, together with a control and status register. One or both of the watchpoint units can be programmed to halt the execution of instructions by the core through its BREAKPT signal. Execution is halted when a match occurs between the values programmed into EmbeddedICE macrocell and the values

currently appearing on the address bus, data bus and various control signals. Any bit can be masked so that its value does not affect the comparison.

Figure 9.1 shows the relationship between the core, EmbeddedICE macrocell and the TAP controller.

Note: Only those signals that are pertinent to EmbeddedICE macrocell are shown.

Figure 9.1 EmbeddedICE Block Diagram



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

Two independent registers, Debug Control and Debug Status, provide overall control of the EmbeddedICE macrocell operation.

9.2 Watchpoint Registers

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

1. Address Value and Address Mask
2. Data Value and Data Mask
3. Control Value and Control Mask

Each register is independently programmable, and has its own address: see [Table 9.1](#).

Table 9.1 Function and Mapping of EmbeddedICE Registers

Address	Width	Function
0b00000	3	Debug Control
0b00001	5	Debug Status
0b00100	6	Debug Communications Control Register
0b00101	32	Debug Communications Data Register
0b01000	32	Watchpoint 0 Address Value
0b01001	32	Watchpoint 0 Address Mask
0b01010	32	Watchpoint 0 Data Value
0b01011	32	Watchpoint 0 Data Mask
(Sheet 1 of 2)		

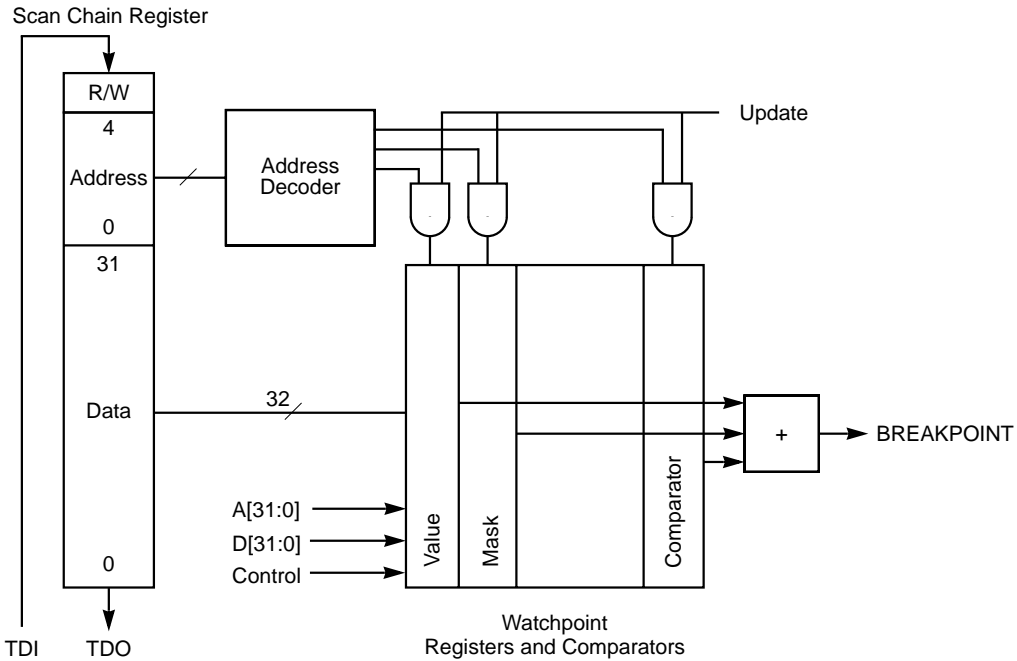
Table 9.1 Function and Mapping of EmbeddedICE Registers (Cont.)

Address	Width	Function
0b01100	9	Watchpoint 0 Control Value
0b01101	8	Watchpoint 0 Control Mask
0b10000	32	Watchpoint 1 Address Value
0b10001	32	Watchpoint 1 Address Mask
0b10010	32	Watchpoint 1 Data Value
0b10011	32	Watchpoint 1 Data Mask
0b10100	9	Watchpoint 1 Control Value
0b10101	8	Watchpoint 1 Control Mask
(Sheet 2 of 2)		

9.2.1 Programming and Reading Watchpoint Registers

A register is programmed by scanning data into the EmbeddedICE macrocell scan chain (scan chain 2). The scan chain consists of a 38-bit shift register comprising a 32-bit data field, a 5-bit address field and a read/write bit. This is shown in [Figure 9.2](#).

Figure 9.2 EmbeddedICE Macrocell Block Diagram



The data to be written is scanned into the 32-bit data field, the address of the register into the 5-bit address field and a 1 into the read/write bit.

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

The register addresses are shown in [Table 9.1](#).

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

9.2.2 Using the Mask Registers

For each Value register in a register pair, there is a Mask register of the same format. Setting a bit to 1 in the Mask register causes the comparator to disregard the corresponding bit in the Value register.

For example, if a watchpoint is required on a particular memory location but the data value is irrelevant, the Data Mask register can be programmed to 0xFFFFFFFF (all bits set to 1) to make the entire Data Bus field ignored.

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 will always match, irrespective of the value register or the input value.

Setting the mask bit to 0 means that the comparator will only match if the input value matches the value programmed into the value register.

9.2.3 Control Registers

The Control Value and Control Mask registers are mapped identically in the lower eight bits, as shown below.

Figure 9.3 Watchpoint Control Value and Mask Format

8	7	6	5	4	3	2	1	0
ENABLE	RANGE	CHAIN	EXTERN	nTRANS	nOPC	MAS[1]	MAS[0]	nRW

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

The bits have the following functions:

- ENABLE** **8**
If a watchpoint match occurs, the BREAKPT signal will only be asserted when the ENABLE bit is set. This bit only exists in the value register: it cannot be masked.
- RANGE** **7**
Can be connected to the range output of another watchpoint register. In the EmbeddedICE macrocell, the RANGEOUT output of Watchpoint 1 is connected to the RANGE input of Watchpoint 0. This allows the two watchpoints to be coupled for detecting conditions that occur simultaneously, e.g., for range checking.

CHAIN**6**

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

In the EmbeddedICE macrocell, the CHAINOUT output of Watchpoint 1 is connected to the CHAIN input of Watchpoint 0. The CHAINOUT output is derived from a latch; the address/control field comparator drives the write enable for the latch and the input to the latch is the value of the data field comparator. The CHAINOUT latch is cleared when the Control Value register is written or when nTRST is LOW.

EXTERN**5**

An external input to EmbeddedICE macrocell which allows the watchpoint to be dependent upon some external condition. The EXTERN input for Watchpoint 0 is labelled EXTERN0 and the EXTERN input for Watchpoint 1 is labelled EXTERN1.

nTRANS**4**

Compares against the not translate signal from the core in order to distinguish between user mode (nTRANS = 0) and nonuser mode (nTRANS = 1) accesses.

nOPC**3**

Used to detect whether the current cycle is an instruction fetch (nOPC = 0) or a data access (nOPC = 1).

MAS[1:0]**[2:1]**

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

The encoding is shown in the following table.

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

nRW

0

Compares against the not read/write signal from the core in order to detect the direction of bus activity. nRW is zero for a read cycle and one for a write cycle.

For each of the bits 8:0 in the Control Value register, there is a corresponding bit in the Control Mask register. This removes the dependency on particular signals.

9.3 Programming Breakpoints

Breakpoints can be classified as hardware breakpoints or software breakpoints.

Hardware breakpoints – Monitor the address value and can be set in any code, even in code that is in ROM or code that is selfmodifying.

Software breakpoints – Monitor a particular bit pattern being fetched from any address. One EmbeddedICE macrocell watchpoint can thus be used to support any number of software breakpoints. Software breakpoints can normally only be set in RAM because an instruction has to be replaced by the special bit pattern chosen to cause a software breakpoint.

9.3.1 Hardware Breakpoints

To make a watchpoint unit cause hardware breakpoints (i.e., on instruction fetches):

1. Program its Address Value register with the address of the instruction that will generate the breakpoint.
2. For a breakpoint in ARM state, program bits [1:0] of the Address Mask register to 1. For a breakpoint in THUMB state, program bit 0 of the Address Mask to 1. In both cases the remaining bits are set to 0.
3. Program the Data Value register only if you require a data dependent breakpoint: i.e., only if the actual instruction code fetched must be matched as well as the address. If the data value is not required, program the Data Mask register to 0xFFFFFFFF (all bits to 1), otherwise program it to 0x00000000.

4. Program the Control Value register with nOPC = 0.
5. Program the Control Mask register with nOPC = 0, all other bits to 1.
6. If you need to make the distinction between user and nonuser mode instruction fetches, program the nTRANS Value and Mask bits as in steps 4 and 5 above.
7. If required, program the EXTERN, RANGE and CHAIN Value and Mask bits in the same way as in steps 4 and 5 above.

9.3.2 Software Breakpoints

To make a watchpoint unit cause software breakpoints (i.e., on instruction fetches of a particular bit pattern):

1. Program its Address Mask register to 0xFFFFFFFF (all bits set to 1) so that the address is disregarded.
2. Program the Data Value register with the particular bit pattern that has been chosen to represent a software breakpoint.
3. If a THUMB software breakpoint is being programmed, the 16-bit pattern must be repeated in both halves of the Data Value register. For example, if the bit pattern is 0xDFFF, then 0xDFFFDFDF must be programmed. When a 16-bit instruction is fetched, EmbeddedICE macrocell only compares the valid half of the data bus against the contents of the Data Value register. In this way, a single Watchpoint register can be used to catch software breakpoints on both the upper and lower halves of the data bus.
4. Program the Data Mask register to 0x00000000.
5. Program the Control Value register with nOPC = 0.
6. Program the Control Mask register with nOPC = 0, all other bits to 1.
7. If you wish to make the distinction between user and nonuser mode instruction fetches, program the nTRANS bit in the Control Value and Control Mask registers accordingly.
8. If required, program the EXTERN, RANGE and CHAIN bits in the same way as in steps 5 and 6 above.

Note: The address value register need not be programmed.

9.3.2.1 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.

9.3.2.2 Clearing the Breakpoint

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

9.4 Programming Watchpoints

To make a watchpoint unit cause watchpoints (i.e., on data accesses):

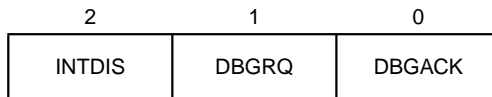
1. Program its Address Value register with the address of the data access to be watchpointed.
2. Program the Address Mask register to 0x00000000.
3. Program the Data Value register only if you require a data dependent watchpoint; i.e. only if the actual data value read or written must be matched as well as the address. If the data value is irrelevant, program the Data Mask register to 0xFFFFFFFF (all bits set to 1) otherwise program it to 0x00000000.
4. Program the Control Value register with nOPC = 1, nRW = 0 for a read or nRW = 1 for a write, MAS[1:0] with the value corresponding to the appropriate data size.
5. Program the Control Mask register with nOPC = 0, nRW = 0, MAS[1:0] = 0, all other bits to 1. Note that nRW or MAS[1:0] may be set to 1 if both reads and writes or data size accesses are to generate watchpoints respectively.
6. If you wish to make the distinction between user and nonuser mode data accesses, program the nTRANS bit in the Control Value and Control Mask registers accordingly.
7. If required, program the EXTERN, RANGE and CHAIN bits in the same way as steps 4 and 5 above.

Note: The above are just examples of how to program the watchpoint register to generate breakpoints and watchpoints; many other ways of programming the registers are possible. For instance, simple range breakpoints can be provided by setting one or more of the address mask bits.

9.5 Debug Control Register

The Debug Control register is 3 bits wide. If the register is accessed for a write (with the read/write bit HIGH), the control bits are written. If the register is accessed for a read (with the read/write bit LOW), the control bits are read. The layout of the Debug Control register follows in [Figure 9.4](#)

Figure 9.4 Debug Control Register Format



As shown in [Figure 9.6](#), the value stored in bit 1 of the control register is synchronized and then ORed with the external DBGRQ before being applied to the processor. The output of this OR gate is the signal DBGRQI which is brought out externally from the macrocell.

The synchronization between control bit 1 and DBGRQI is to assist in multiprocessor environments. The synchronization latch only opens when the TAP controller state machine is in the Run-Test/Idle state. This allows an enter debug condition to be setup in all the processors in the system while they are still running. Once the condition is setup in all the processors, it can then be applied to them simultaneously by entering the Run-Test/Idle state.

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

If Bit 2 (INTDIS) is asserted, the interrupt enable signal (IFEN) of the core is forced LOW. Thus all interrupts (nIRQ and nFIQ) are disabled during debugging (DBGACK = 1) or if the INTDIS bit is asserted. The IFEN signal is driven according to the following table

Table 9.2 IFEN Signal Control

DBGACK	INTDIS	IFEN
0	0	1
1	x	0
x	1	0

9.6 Debug Status Register

The Debug Status register is 5 bits wide. If it is accessed for a write (with the read/write bit set HIGH), the status bits are written. If it is accessed for a read (with the read/write bit LOW), the status bits are read.

Figure 9.5 Debug Status Register Format

4	3	2	1	0
TBIT	nMREQ	IFEN	DBGRQ	DBGACK

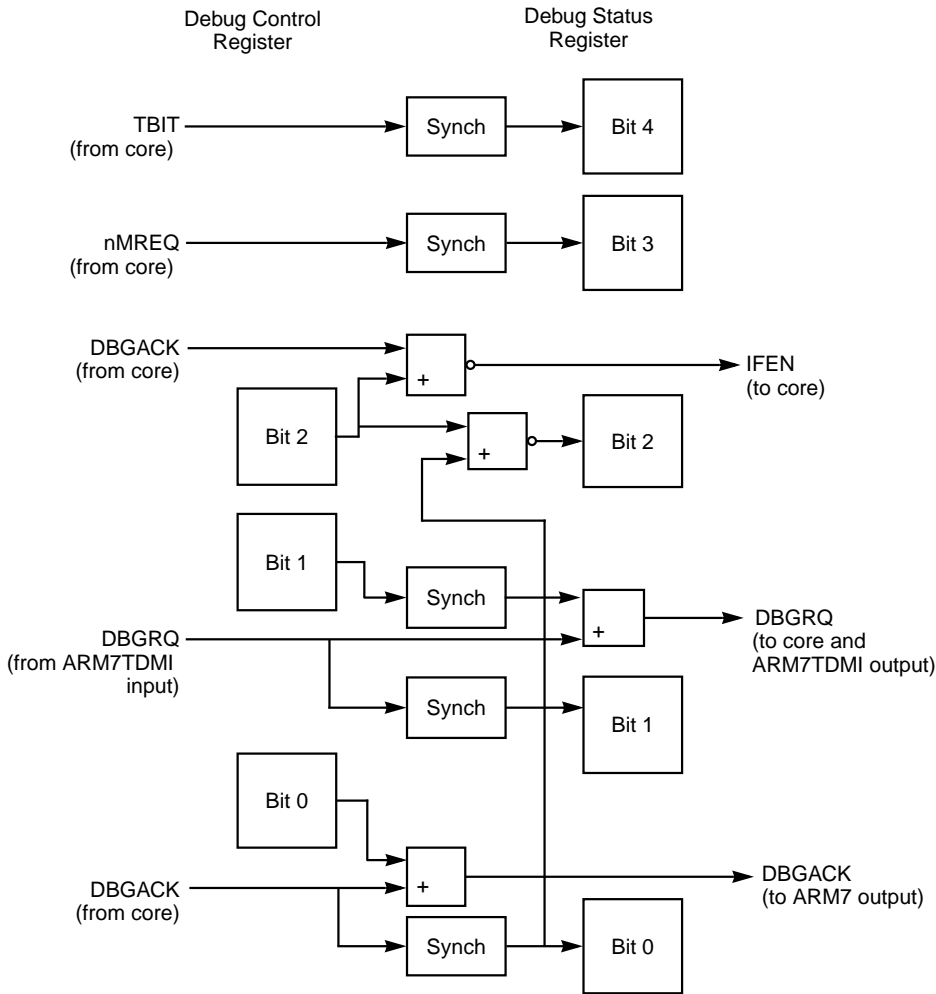
The function of each bit in this register is as follows:

- TBIT** **4**
 Allows TBIT to be read. This enables the debugger to determine what state the processor is in, and hence which instructions to execute.
- nMREQ** **3**
 Allows the state of the nMREQ signal from the core (synchronized to TCK) to be read. This allows the debugger to determine that a memory access from the debug state has completed.

IFEN	Allows the state of the core interrupt enable signal (IFEN) to be read. Since the capture clock for the scan chain may be asynchronous to the processor clock, the DBGACK output from the core is synchronized before being used to generate the IFEN status bit.	2
DBGRQ	Allows the value on the synchronized version of DBGRQ to be read.	1
DBGACK	Allows the value on the synchronized version of DBGACK to be read.	0

The structure of the debug status register bits is shown in [Figure 9.6](#).

Figure 9.6 Structure of TBIT, nMREQ, DBGACK, DBGQR and INTDIS Bits



9.7 Coupling Breakpoints and Watchpoints

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

For the next few examples, let:

$A_v[31:0]$ be the value in the Address Value register

$A_m[31:0]$ be the value in the Address Mask register

$A[31:0]$ be the Address Bus from the core

$D_v[31:0]$ be the value in the Data Value register

$D_m[31:0]$ be the value in the Data Mask register

$D[31:0]$ be the Data Bus from the core

$C_v[8:0]$ be the value in the Control Value register

$C_m[7:0]$ be the value in the Control Mask register

$C[9:0]$ be the combined Control Bus from the core, other watchpoint registers and the EXTERN signal.

9.7.1 CHAINOUT Signal

The CHAINOUT signal is then derived as follows:

WHEN ($\{A_v[31:0], C_v[4:0]\}$ XNOR $\{A[31:0], C[4:0]\}$) OR
 $\{A_m[31:0], C_m[4:0]\} == 0xFFFFFFFF$)

CHAINOUT = ($\{D_v[31:0], C_v[6:4]\}$ XNOR $\{D[31:0], C[7:5]\}$) OR
 $\{D_m[31:0], C_m[7:5]\} == 0x7FFFFFFF$)

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

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, the above function can be implemented with a watchpoint and breakpoint chained together. The watchpoint address is set to a known memory location containing the current process ID, the watchpoint data is set to the required process ID and the ENABLE bit is set to "off".

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

9.7.2 RANGEOUT Signal

The RANGEOUT signal is then derived as follows:

$$\text{RANGEOUT} = (((\{Av[31:0], Cv[4:0]\} \text{XNOR } \{A[31:0], C[4:0]\}) \text{ OR } \{Am[31:0], Cm[4:0]\}) == 0xFFFFFFFF) \text{ AND } (((\{Dv[31:0], Cv[7:5]\} \text{XNOR } \{D[31:0], C[7:5]\}) \text{ OR } \{Dm[31:0], Cm[7:5]\}) == 0x7FFFFFFF)$$

The RANGEOUT output of watchpoint register 1 provides the RANGE input to watchpoint register 0. This allows two breakpoints to be coupled together to form range breakpoints. Note that selectable ranges are restricted to being powers of 2. This is best illustrated by an example.

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, the watchpoint registers should be programmed as follows:

1. Watchpoint 1 is programmed with an address value of 0x00000000 and an address mask of 0x0000001F. The ENABLE bit is cleared. All other Watchpoint 1 registers are programmed as normal for a breakpoint. An address within the first 32 bytes will cause the RANGE output to go HIGH but the breakpoint will not be triggered.
2. Watchpoint 0 is programmed with an address value of 0x00000000 and an address mask of 0x000000FF. The ENABLE bit is set and the RANGE bit programmed to match a 0. All other Watchpoint 0 registers are programmed as normal for a breakpoint.

If Watchpoint 0 matches but Watchpoint 1 does not (i.e., the RANGE input to Watchpoint 0 is 0), the breakpoint will be triggered.

9.8 Disabling EmbeddedICE Macrocell

EmbeddedICE macrocell may be disabled by wiring the DBGEN input LOW.

When DBGEN is LOW, BREAKPT and DBGRQ to the core are forced LOW, DBGACK from the core is also forced LOW and the IFEN input to the core is forced HIGH, enabling interrupts to be detected by the core.

When DBGEN is LOW, EmbeddedICE macrocell is also put into a low power mode.

9.9 EmbeddedICE Macrocell Timing

The EXTERN1 and EXTERN0 inputs are sampled by EmbeddedICE macrocell on the falling edge of ECLK. Sufficient setup and hold time must therefore be allowed for these signals.

9.10 Programming Restriction

The EmbeddedICE macrocell watchpoint units should only be programmed when the clock to the core is stopped. This can be achieved by putting the core into the debug state.

The reason for this restriction is that if the core continues to run at ECLK rates when EmbeddedICE macrocell is being programmed at TCK rates, it is possible for the BREAKPT signal to be asserted asynchronously to the core.

This restriction does not apply if MCLK and TCK are driven from the same clock, or if it is known that the breakpoint or watchpoint condition can only occur some time after EmbeddedICE macrocell has been programmed.

Note: This restriction does not apply in any event to the Debug Control or Status registers.

9.11 Debug Communication Channel

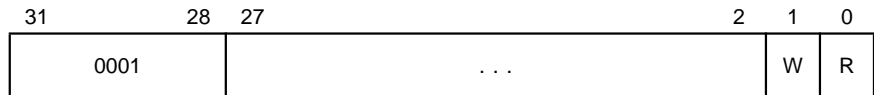
The EmbeddedICE macrocell contains a communication channel for passing information between the target and the host debugger. This is implemented as coprocessor 14.

The communications channel consists of a 32-bit wide Communications Data Read register, a 32-bit wide Communications Data Write register and a 6-bit wide Communications Control register for synchronized handshaking between the processor and the asynchronous debugger. These registers live in fixed locations in EmbeddedICE macrocell's memory map (as shown in [Table 9.1](#)) and are accessed from the processor using the MCR and MRC instructions to coprocessor 14.

9.11.1 Debug Communications Control Registers

The Debug Communications Control register is read only and allows synchronized handshaking between the processor and the debugger.

Figure 9.7 Debug Communications Control Register



The function of each register bit is described below:

0001	[31:28]
W	1
Denotes whether the Communications Data Write register is free or not. If the Communications Data Write register is free ($W = 0$) then new data may be written. If it is not free ($W = 1$), then the processor must poll until $W = 0$. From the debugger's point of view, if $W = 1$ then some new data has been written which may then be scanned out.	
R	0
Denotes whether there is some new data in the Communications Data Read register. If $R = 1$, then there is some new data which may be read using an MRC instruction. If $R = 0$ then the Communications Data Read	

register is free and new data may be placed there through the scan chain. If $R = 1$, then this denotes that data previously placed there through the scan chain has not been collected by the processor and so the debugger must wait.

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

The following instructions should be used:

```
MRC P14, 0, Rd, C0, C0, 0
```

Returns the Debug Communications Control register into Rd

```
MCR P14, 0, Rn, C1, C0, 0
```

Writes the value in Rn to the Communications Data Write register

```
MRC P14, 0, Rd, C1, C0, 0
```

Returns the Debug Data Read register into Rd

Since the THUMB instruction set does not contain coprocessor instructions, it is recommended that these are accessed using SWI instructions when in THUMB state.

9.11.2 Communication Through the Communications Channel

Communication between the debugger and the processor occurs as follows. When the processor wishes to send a message to EmbeddedICE macrocell, it first checks that the Communications Data Write register is free for use. This is done by reading the Debug Communications Control register to check that the W bit is clear. If it is clear then the Communications Data Write register is empty and a message is written by a register transfer to the coprocessor. The action of this data transfer automatically sets the W bit. If on reading the W bit it is found to be set, then this implies that previously written data has not been picked up by the debugger and thus the processor must poll until the W bit is clear.

As the data transfer occurs from the processor to the Communications Data Write register, the W bit is set in the Debug Communications Control register. When the debugger polls this register it sees a

synchronized version of both the R and W bit. When the debugger sees that the W bit is set it can read the Communications Data Write register and scan the data out. The action of reading this data register clears the W bit of the Debug Communications Control register. At this point, the communications process may begin again.

Message transfer from the debugger to the processor is carried out in a similar fashion. Here, the debugger polls the R bit of the Debug Communications Control register. If the R bit is low then the Data Read register is free and so data can be placed there for the processor to read. If the R bit is set, then previously deposited data has not yet been collected and so the debugger must wait.

When the Communications Data Read register is free, data is written there using the scan chain. The action of this write sets the R bit in the Debug Communications Control register. When the processor polls this register, it sees an MCLK synchronized version. If the R bit is set then this denotes that there is data waiting to be collected, and this can be read using a CPRT load. The action of this load clears the R bit in the Debug Communications Control register. When the debugger polls this register and sees that the R bit is clear, this denotes that the data has been taken and the process may now be repeated.

Chapter 10

Instruction Cycle Operations

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

- [Section 10.1, “Introduction,” page 10-2](#)
- [Section 10.2, “Branch and Branch with Link,” page 10-2](#)
- [Section 10.3, “THUMB Branch with Link,” page 10-3](#)
- [Section 10.4, “Branch and Exchange \(BX\),” page 10-4](#)
- [Section 10.5, “Data Operations,” page 10-5](#)
- [Section 10.6, “Multiply and Multiply Accumulate,” page 10-7](#)
- [Section 10.7, “Load Register,” page 10-9](#)
- [Section 10.8, “Store Register,” page 10-10](#)
- [Section 10.9, “Load Multiple Registers,” page 10-10](#)
- [Section 10.10, “Store Multiple Registers,” page 10-12](#)
- [Section 10.11, “Data Swap,” page 10-13](#)
- [Section 10.12, “Software Interrupt and Exception Entry,” page 10-14](#)
- [Section 10.13, “Coprocessor Data Operation,” page 10-15](#)
- [Section 10.14, “Coprocessor Data Transfer \(Memory to Coprocessor\),” page 10-16](#)
- [Section 10.15, “Coprocessor Data Transfer \(from Coprocessor to Memory\),” page 10-18](#)
- [Section 10.16, “Coprocessor Register Transfer \(Load from Coprocessor\),” page 10-20](#)
- [Section 10.17, “Coprocessor Register Transfer \(Store to Coprocessor\),” page 10-21](#)

- [Section 10.18, “Undefined Instructions and Coprocessor Absent,” page 10-22](#)
- [Section 10.19, “Unexecuted Instructions,” page 10-23](#)
- [Section 10.20, “Instruction Speed Summary,” page 10-23](#)

10.1 Introduction

In the following tables nMREQ and SEQ (which are pipelined up to one cycle ahead of the cycle to which they apply) are shown in the cycle in which they appear, so they predict the type of the next cycle. The address, MAS[1:0], nRW, nOPC, nTRANS and TBIT (which appear up to half a cycle ahead) are shown in the cycle to which they apply. The address is incremented for prefetching of instructions in most cases. Since the instruction width is 4 bytes in ARM state and 2 bytes in THUMB state, the increment will vary accordingly. Hence the letter L is used to indicate instruction length (4 bytes in ARM state and 2 bytes in THUMB state). Similarly, the letter i indicates the width of the instruction fetch (i = 2 in ARM state and i = 1 in THUMB state) representing word and halfword accesses respectively.

10.2 Branch and Branch with Link

A branch instruction calculates the branch destination in the first cycle, while performing a prefetch from the current PC. This prefetch is done in all cases, since by the time the decision to take the branch has been reached it is already too late to prevent the prefetch.

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

The third cycle performs a fetch from the destination + L, refilling the instruction pipeline, and if the branch with link R14 is modified (4 is subtracted from it) to simplify return from `SUB PC,R14,#4` to `MOV PC,R14`. This makes the `STM. . {R14} LDM. . {PC}` type of subroutine work correctly. The cycle timings are shown below in [Table 10.1](#).

Table 10.1 Branch Instruction Cycle Operations¹

Cycle	Address	MAS[1:0] ²	nRW	Data	nMREQ	SEQ	nOPC
1	pc + 2L ³	i	0	(pc + 2L)	0	0	0
2	alu ⁴	i	0	(alu) ⁵	0	1	0
3	alu + L	i	0	(alu + L)	0	1	0
	alu + 2L						

1. This table applies to branches in ARM and THUMB state, and to Branch with Link in ARM state only.
2. i = 2 in ARM state and i = 1 in THUMB state.
3. pc is the address of the branch instruction.
4. alu is an address calculated by the core.
5. (alu) are the contents of that address.

10.3 THUMB Branch with Link

A THUMB Branch with Link operation consists of two consecutive THUMB instructions.

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

The second instruction acts in a similar fashion to the ARM Branch with Link instruction, thus its first cycle calculates the final branch destination while performing a prefetch from the current PC.

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

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

The cycle timings of the complete operation are shown in [Table 10.2](#).

Table 10.2 THUMB Long Branch with Link

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

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

10.4 Branch and Exchange (BX)

A Branch and Exchange operation takes 3 cycles and is similar to a Branch.

In the first cycle, the branch destination and the new core state are extracted from the register source, while performing a prefetch from the current PC. This prefetch is performed in all cases, since by the time the decision to take the branch has been reached, it is already too late to prevent the prefetch.

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

The third cycle performs a fetch from the destination + 2 or + 4 depending on the new specified state, refilling the instruction pipeline. The cycle timings are shown in [Table 10.3](#).

Table 10.3 Branch and Exchange Instruction Cycle Operations

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

1. W and w represent the instruction width before and after the BX respectively. In ARM state the width equals 4 bytes and in THUMB state the width equals 2 bytes. For example, when changing from ARM to THUMB state, W would equal 4 and w would equal 2.
2. l and i represent the memory access size before and after the BX respectively. In ARM state, the MAS[1:0] is 2 and in THUMB state MAS[1:0] is 1. When changing from THUMB to ARM state, l would equal 1 and i would equal 2.
3. T and t represent the state 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 would equal 0 and t would equal 1.

10.5 Data Operations

A data operation executes in a single datapath cycle except where the shift is determined by the contents of a register. A register is read onto the A bus, and a second register or the immediate field onto the B bus. The ALU combines the A bus source and the shifted B bus source according to the operation specified in the instruction, and the result (when required) is written to the destination register. (Compares and tests do not produce results, only the ALU status flags are affected.)

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

When the shift length is specified by a register, an additional datapath cycle occurs before the above operation to copy the bottom 8 bits of that register into a holding latch in the barrel shifter. The instruction prefetch will occur during this first cycle, and the operation cycle will be internal (i.e., will not request memory). This internal cycle can be merged with the following sequential access by the memory manager as the address remains stable through both cycles.

The PC may be one or more of the register operands. When it is the destination, external bus activity may be affected. If the result is written to the PC, the contents of the instruction pipeline are invalidated, and the address for the next instruction prefetch is taken from the ALU rather than the address incrementer. The instruction pipeline is refilled before any further execution takes place, and during this time exceptions are 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 cycle timings are shown below [Table 10.4](#).

Table 10.4 Data Operation Instruction Cycle Operations

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

1. i = 2 in ARM state and i = 1 in THUMB state
2. Shifted register with destination equals PC is not possible in THUMB state.

10.6 Multiply and Multiply Accumulate

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

The cycle timings are shown in the following four tables, where m is the number of cycles required by the multiplication algorithm; see [Section 10.20, “Instruction Speed Summary,”](#) for more information.

Table 10.5 Multiply Instruction Cycle Operations

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

1. $i = 2$ in ARM state and $i = 1$ in THUMB state.

Table 10.6 Multiply Accumulate Instruction Cycle Operations

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

1. $i = 2$ in ARM state and $i = 1$ in THUMB state.

Table 10.7 Multiply Long Instruction Cycle Operation

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

1. $i = 2$ in ARM state and $i = 1$ in THUMB state.

Table 10.8 Multiply Accumulate¹ Long Instruction Cycle Operation

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

1. Multiply Accumulate is not possible in THUMB state.

2. $i = 2$ in ARM state and $i = 1$ in THUMB state.

10.7 Load Register

The first cycle of a load register instruction performs the address calculation. The data is fetched from memory during the second cycle, and the base register modification is performed during this cycle (if required). During the third cycle the data is transferred to the destination register, and external memory is unused. This third cycle may normally be merged with the following prefetch to form one memory N-cycle. The cycle timings are shown below in [Table 10.9](#).

Either the base or the destination (or both) may be the PC, and the prefetch sequence will be changed if the PC is affected by the instruction.

The data fetch may abort, and in this case the destination modification is prevented

Table 10.9 Load Register Instruction Cycle Operations

	Cycle	Address	MAS[1:0] ¹	nRW	Data	nMREQ	SEQ	nOPC	nTRANS
normal	1	pc + 2L	i	0	(pc + 2L)	0	0	0	c ²
	2	alu	b/h/w ³	0	(alu)	1	0	1	d ⁴
	3	pc + 3L	i	0	–	0	1	1	c
		pc + 3L							
dest = pc ⁵	1	pc +	2	0	(pc + 8)	0	0	0	c
	2	alu		0	pc'	1	0	1	d
	3	pc + 12	2	0	–	0	0	1	c
	4	pc'	2	0	(pc')	0	1	0	c
	5	pc' + 4	2	0	(pc' + 4)	0	1	0	c
		pc' + 8							

1. i = 2 in ARM state and i = 1 in THUMB state.
2. c represents current mode-dependent value.
3. b, h and w are byte, halfword and word as defined in [Section 9.5, “Debug Control Register.”](#)
4. d will either be 0 if the T bit has been specified in the instruction (eg. LDRT), or c at all other times.
5. Destination equals PC is not possible in THUMB state.

10.8 Store Register

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

Table 10.10 Store Register Instruction Cycle Operations

Cycle	Address	MAS[1:0] ¹	nRW	Data	nMREQ	SEQ	nOPC	nTRANS
1	pc + 2L	i	0	(pc + 2L)	0	0	0	c ²
2	alu	b/h/w ³	1	Rd	0	0	1	d ⁴
	pc + 3L							

1. i = 2 in ARM state and i = 1 in THUMB state.
2. c represents current mode dependent value.
3. b, h, and w are byte, halfword and word as defined in [Section 9.5, “Debug Control Register.”](#)
4. d will either be 0 if the T bit has been specified in the instruction (e.g., SDRT), or c at all other times.

10.9 Load Multiple Registers

The first cycle of LDM is used to calculate the address of the first word to be transferred, while performing a prefetch from memory. The second cycle fetches the first word, and performs the base modification. During the third cycle, the first word is moved to the appropriate destination register while the second word is fetched from memory, and the modified base is latched internally in case it is needed to patch up after an abort. The third cycle is repeated for subsequent fetches until the last data word has been accessed, then the final (internal) cycle moves the last word to its destination register. The cycle timings are shown in [Table 10.11](#).

The last cycle may be merged with the next instruction prefetch to form a single memory N-cycle.

If an abort occurs, the instruction continues to completion, but all register writing after the abort is prevented. The final cycle is altered to restore the modified base register (which may have been overwritten by the load activity before the abort occurred).

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

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

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

Table 10.11 Load Multiple Registers Instruction Cycle Operations

	Cycle	Address	MAS[1:0] ¹	nRW	Data	nMREQ	SEQ	nOPC
1 register	1	pc + 2L	i	0	(pc + 2L)	0	0	0
	2	alu	2	0	(alu)	1	0	1
	3	pc + 3L	i	0	–	0	1	1
		pc + 3L						
1 register dest = pc	1	pc + 2L	i	0	(pc + 2L)	0	0	0
	2	alu	2	0	pc'	1	0	1
	3	pc + 3L	i	0	–	0	0	1
	4	pc'	i	0	(pc')	0	1	0
	5	pc' + L	i	0	(pc' + L)	0	1	0
		pc' + 2L						
n registers (n > 1)	1	pc + 2L	i	0	(pc + 2L)	0	0	0
	2	alu	2	0	(alu)	0	1	1
	•	alu + •	2	0	(alu + •)	0	1	1
	n	alu + •	2	0	(alu + •)	0	1	1
	n + 1	alu + •	2	0	(alu + •)	1	0	1
	n + 2	pc + 3L	i	0	–	0	1	1
		pc + 3L						
(Sheet 1 of 2)								

Table 10.11 Load Multiple Registers Instruction Cycle Operations (Cont.)

	Cycle	Address	MAS[1:0] ¹	nRW	Data	nMREQ	SEQ	nOPC
n registers (n > 1) incl pc	1	pc + 2L	i	0	(pc + 2L)	0	0	0
	2	alu	2	0	(alu)	0	1	1
	•	alu + •	2	0	(alu + •)	0	1	1
	n	alu + •	2	0	(alu + •)	0	1	1
	n + 1	alu + •	2	0	pc'	1	0	1
	n + 2	pc + 3L	i	0	–	0	0	1
	n + 3	pc'	i	0	(pc')	0	1	0
	n + 4	pc' + L	i	0	(pc' + L)	0	1	0
		pc' + 2L						

(Sheet 2 of 2)

1. i = 2 in ARM state and i = 1 in THUMB state.

10.10 Store Multiple Registers

Store multiple registers proceeds very much as load multiple, without the final cycle. The restart problem is much more straightforward here, as there is no wholesale overwriting of registers. The cycle timings are shown in [Table 10.12](#), below.

Table 10.12 Store Multiple Registers Instruction Cycle Operations

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

(Sheet 1 of 2)

Table 10.12 Store Multiple Registers Instruction Cycle Operations (Cont.)

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

1. i = 2 in ARM state and i = 1 in THUMB state.

10.11 Data Swap

This is similar to the load and store register instructions, but the actual swap takes place in cycles 2 and 3. In the second cycle, the data is fetched from external memory. In the third cycle, the contents of the source register are written out to the external memory. The data read in cycle 2 is written into the destination register during the fourth cycle. The cycle timings are shown below in [Table 10.13](#).

The LOCK output of the core is driven HIGH for the duration of the swap operation (cycles 2 and 3) to indicate that both cycles should be allowed to complete without interruption.

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

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

Table 10.13 Data Swap Instruction Cycle Operations¹

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

1. Data swap cannot be executed in THUMB state.
2. b and w are byte and word as defined in [Section 9.5, “Debug Control Register.”](#)

10.12 Software Interrupt and Exception Entry

Exceptions (and software interrupts) force the PC to a particular value and refill the instruction pipeline from there. During the first cycle the forced address is constructed, and a mode change may take place. The return address is moved to R14 and the CPSR to SPSR_svc.

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

The third cycle is required only to complete the refilling of the instruction pipeline. The cycle timings are shown below in [Table 10.14](#).

Table 10.14 Software Interrupt Instruction Cycle Operations

Cycle	Address	MAS [1:0] ¹	nRW	Data	nMREQ	SEQ	nOPC	nTRANS	Mode	TBIT
1	pc + 2L ²	i	0	(pc + 2L)	0	0	0	C ³	old mode	T ⁴
2	Xn ⁵	2	0	(Xn)	0	1	0	1	exception mode	0
3	Xn + 4	2	0	(Xn + 4)	0	1	0	1	exception mode	0
	Xn + 8									

1. i = 2 in ARM state and i = 1 in THUMB state.
2. pc is for software interrupts is the address of the SWI instruction. For exceptions is the address of the instruction following the last one to be executed before entering the exception. For prefetch aborts is the address of the aborting instruction. For data aborts is the address of the instruction following the one which attempted the aborted data transfer.
3. C represents the current mode dependent value.
4. T represents the current state dependent value.
5. Xn is the appropriate trap address.

10.13 Coprocessor Data Operation

This operation cannot occur in THUMB state. A coprocessor data operation is a request from the core for the coprocessor to initiate some action. The action need not be completed for some time, but the coprocessor must commit to doing it before driving CPB LOW.

If the coprocessor can never do the requested task, it should leave CPA and CPB HIGH. If it can do the task, but can't commit right now, it should drive CPA LOW but leave CPB HIGH until it can commit. The core will busy-wait until CPB goes LOW. The cycle timings are shown in [Table 10.15](#).

Table 10.15 Coprocessor Data Operation Instruction Cycle Operations¹

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

1. This operation cannot occur in THUMB state.

10.14 Coprocessor Data Transfer (Memory to Coprocessor)

This operation cannot occur in THUMB state. Here the coprocessor should commit to the transfer only when it is ready to accept the data. When CPB goes LOW, the core will produce addresses and expect the coprocessor to take the data at sequential cycle rates. The coprocessor is responsible for determining the number of words to be transferred, and indicates the last transfer cycle by driving CPA and CPB HIGH.

The core spends the first cycle (and any busy-wait cycles) generating the transfer address, and performs the write-back of the address base during the transfer cycles. The cycle timings are shown in [Table 10.16](#).

Table 10.16 Coprocessor Data Transfer Instruction Cycle Operations¹

	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
1 register ready	1	pc + 8	2	0	(pc + 8)	0	0	0	0	0	0
	2	(alu)	2	0	(alu)	0	0	1	1	1	1
		pc + 12									
1 register not ready	1	pc + 8	2	0	(pc + 8)	1	0	0	0	0	1
	2	pc + 8	2	0	–	1	0	1	0	0	1
	•	pc + 8	2	0	–	1	0	1	0	0	1
	n	pc + 8	2	0	–	0	0	1	0	0	0
	n + 1	alu	2	0	(alu)	0	0	1	1	1	1
		pc + 12									
m registers (m > 1) ready	1	pc + 8	2	0	(pc + 8)	0	0	0	0	0	0
	2	alu	2	0	(alu)	0	1	1	1	0	0
	•	alu + •	2	0	(alu + •)	0	1	1	1	0	0
	n	alu + •	2	0	(alu + •)	0	1	1	1	0	0
	n + 1	alu + •	2	0	(alu + •)	0	0	1	1	1	1
		pc + 12									
(Sheet 1 of 2)											

Table 10.16 Coprocessor Data Transfer Instruction Cycle Operations¹ (Cont.)

	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
m registers (m > 1) not ready	1	pc + 8	2	0	(pc + 8)	1	0	0	0	0	1
	2	pc + 8	2	20	–	1	0	1	0	0	1
	•	pc + 8	2	0	–	1	0	1	0	0	1
	n	pc + 8	2	0	–	0	0	1	0	0	0
	n + 1	alu	2	0	(alu)	0	1	1	1	0	0
	•	alu + •		0	(alu + •)	0	1	1	1	0	0
	n + m ²	alu + •	2	0	(alu + •)	0	1	1	1	0	0
	n + m + 1	alu + •	2	2	(alu + •)	0	0	1	1	1	1
	pc + 12										

(Sheet 2 of 2)

1. This operation cannot occur in THUMB state.
2. m is number of registers being transferred, n is the number of cycles.

10.15 Coprocessor Data Transfer (from Coprocessor to Memory)

This operation cannot occur in THUMB state. The core controls these instructions exactly as for memory to coprocessor transfers, with the one exception that the nRW line is inverted during the transfer cycle. The cycle timings are show in [Table 10.17](#).

Table 10.17 Coprocessor Data Transfer Instruction Cycle Operations¹

	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
1 register ready	1	pc + 8	2	0	(pc + 8)	0	0	0	0	0	0
	2	alu	2	0	(alu)	0	0	1	1	1	1
		pc + 12									
1 register not ready	1	pc + 8	2	0	(pc + 8)	1	0	0	0	0	1
	2	pc + 8	2	0	–	1	0	1	0	0	1
	•	pc + 8	2	0	–	1	0	1	0	0	1
	n	pc + 8	2	0	–	0	0	1	0	0	0
	n + 1	alu	2	0	(alu)	0	0	1	1	1	1
		pc + 12									
m registers (m > 1) ready	1	pc + 8	2	0	(pc + 8)	0	0	0	0	0	0
	2	alu	2	0	(alu)	0	1	1	1	0	0
	•	alu + •	2	0	(alu + •)	0	1	1	1	0	0
	n	alu + •	2	0	(alu + •)	0	1	1	1	0	0
	n + 1	alu + •	2	0	(alu + •)	0	0	1	1	1	1
		pc + 12									
(Sheet 1 of 2)											

Table 10.17 Coprocessor Data Transfer Instruction Cycle Operations¹ (Cont.)

	Cycle	Address	MAS [1:0]	nRW	Data	nMREQ	SEQ	nOPC	nCPI	CPA	CPB
m registers (m > 1) not ready	1	pc + 8	2	0	(pc + 8)	1	0	0	0	0	1
	2	pc + 8	2	0	–	1	0	1	0	0	1
	•	pc + 8	2	0	–	1	0	1	0	0	1
	n	pc + 8	2	0	–	0	0	1	0	0	0
	n + 1	alu	2	0	(alu)	0	1	1	1	0	0
	•	alu + •		0	(alu + •)	0	1	1	1	0	0
	n + m	alu + •	2	0	(alu + •)	0	1	1	1	0	0
	n + m + 1	alu + •	2	0	(alu + •)	0	0	1	1	1	1
		pc + 12									
(Sheet 2 of 2)											

1. This operation cannot occur in THUMB state.

10.16 Coprocessor Register Transfer (Load from Coprocessor)

This operation cannot occur in THUMB state. Here the busy-wait cycles are much as above, but the transfer is limited to one data word, and the core puts the word into the destination register in the third cycle. The third cycle may be merged with the following prefetch cycle into one memory N-cycle as with all the register load instructions. The cycle timings are shown in [Table 10.18](#).

Table 10.18 Coprocessor Register Transfer (Load from Coprocessor)¹

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

1. This operation cannot occur in THUMB state.

10.17 Coprocessor Register Transfer (Store to Coprocessor)

This operation cannot occur in THUMB state. This is the same operation as the load from coprocessor, except that the last cycle is omitted. The cycle timings are shown in [Table 10.19](#).

Table 10.19 Coprocessor Register Transfer (Store to Coprocessor)¹

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

1. This operation cannot occur in THUMB state.

10.18 Undefined Instructions and Coprocessor Absent

This operation cannot occur in THUMB state. When a coprocessor detects a coprocessor instruction which it cannot perform, and this must include all undefined instructions, it must not drive CPA or CPB LOW. These will remain HIGH, causing the undefined instruction trap to be taken. Cycle timings are shown in [Table 10.20](#).

Table 10.20 Undefined Instruction Cycle Operations¹

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

1. Coprocessor Instructions cannot occur in THUMB state.
2. i = 2 in ARM state and i = 1 in THUMB state.
3. C represents the current mode-dependent value.
4. T represents the current state-dependent value.

10.19 Unexecuted Instructions

Any instruction whose condition code is not met will fail to execute. It will add one cycle to the execution time of the code segment in which it is embedded (see [Table 10.21](#)).

Table 10.21 Unexecuted Instruction Cycle Operations

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

1. i = 2 in ARM state and i = 1 in THUMB state.

10.20 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental

number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in [Table 10.22](#). These figures assume that the instruction is actually executed. Unexecuted instructions take one cycle.

n The number of words transferred

- m =1 Bits [32:8] of the multiplier operand are all zero or one.
- 2 Bits[32:16] of the multiplier operand are all zero or one.
- 3 Bits[31:24] of the multiplier operand are all zero or all one.
- 4 Otherwise.

b The number of cycles spent in the coprocessor busy-wait loop.

If the condition is not met all the instructions take one S-cycle. The cycle types N (Nonsequential), S (Sequential), I (Internal), and C (Coprocessor register transfer) are defined in [Chapter 6, "Memory Interface."](#)

Table 10.22 ARM Instruction Speed Summary

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

Table 10.22 ARM Instruction Speed Summary (Cont.)

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

Chapter 11

Production Test

This chapter describes the ARM7TDMI core production test interface, and contains the following sections:

- [Section 11.1, “Core Testing Strategy Overview,” page 11-1](#)
 - [Section 11.2, “Scan Test Pin Definitions,” page 11-2](#)
 - [Section 11.3, “Full-Scan Production Testing,” page 11-2](#)
-

11.1 Core Testing Strategy Overview

The core implements a full scan methodology for production testing. In addition to the two existing core functional debug scan chains, an additional scan chain spans the entire core and can use the SCAN_EN and MCLK signals in any of the functional clock domains. This new scan chain is 1709 cells long and encompasses both positive and negative edge driven devices. To allow either return to zero (RT0) or return to one (RT1) scan clocks, data lock-up latches have been inserted on the test inputs between flip-flops with complimentary clock polarity.

For production testing of the register file (internal three-port memory), the core uses the production scan chain to serially load RAMBIST data into the internal RAM ports. During scan mode, control of WENCTEST, the write enable signal, is always available at the core periphery. To minimize vector overhead during serial RAMBIST pattern loading, the core leaves the production scan chain segment surrounding the internal memory isolated and accessible during the RAM test mode.

11.2 Scan Test Pin Definitions

Table 11.1 lists the core signals associated with production scan testing. For more information on any of the signals listed, please see [Chapter 2, “Signal Descriptions.”](#)

Table 11.1 Scan Test Pins

Pin Name	Pin Definition	ATPG Patterns	Register File Patterns	Normal Operation
FULLSCAN	Master scan mode select	'1'	'1'	0
RAMTEST	Ramtest scan mode select	'0'	'1'	0
SCAN_EN	Global scan enable	Scan_en	Scan_en	0
SCAN_IN	Full scan chain input	Scan_in	'X'	X
SCAN_OUT	Full scan chain output	Scan_out	'X'	–
RAMSCAN_IN	Ramtest scan chain input	'X'	Scan_in	X
RAMSCAN_OUT	Ramtest scan chain output	'X'	Scan_out	–
WENCTEST	Ramtest write enable	'X'	Write Enable	X
MCLK	Global scan clock	Scan Clock	Scan Clock	–
nTRST	JTAG asynchronous reset	Scan Clock	Scan Clock	0
nRESET	Reset	Scan Clock	Scan Clock	1

11.3 Full-Scan Production Testing

Although the production scan chain uses a single scan clock (MCLK), both nRESET and nTRST can affect flip-flop state and have been added so that ATPG can detect additional faults. MCLK may be either return to one (RT1) or return to zero (RT0) for the core scan testing.

Using a two-stage pattern generation method in Mentor Fastscan, the overall fault coverage for the core is 98.2%. The first stage utilizes conventional ATPG simulation and the second stage applies a sequential RAM test to further increase the fault coverage.

11.3.1 Register File Testing

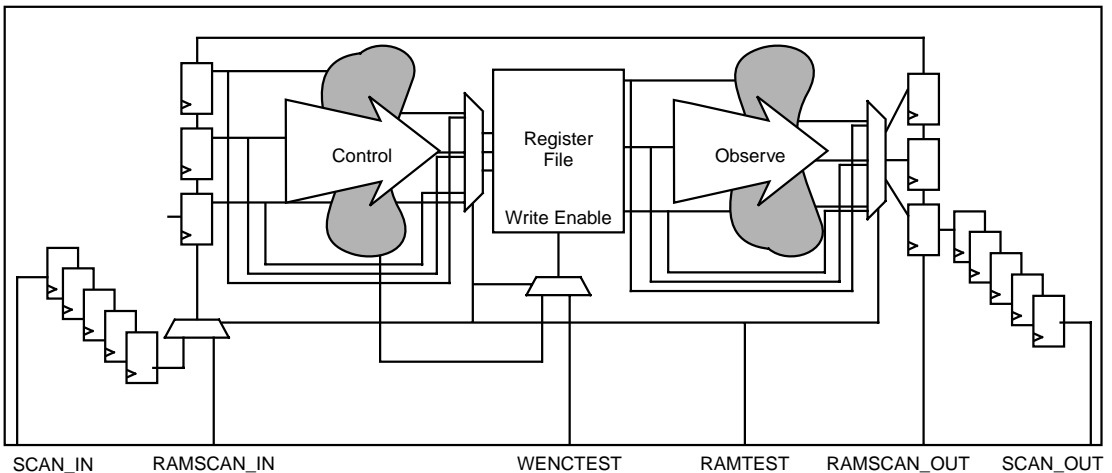
The core implements the register file as a three port (two read, one write) RAM. To completely test the entire register file, the core is designed to allow testing of both the RAM and the RAM control logic.

RAM Control Logic Test – This is possible by substituting the register file netlist for a RAM test model. This netlist model enables the ATPG tool to understand the RAM control logic and produce ATPG vectors that propagate RAM faults, which allows coverage of the circuit elements directly connected to the RAM.

RAM Test – This is accomplished by multiplexing the register file ports to the nearest scan elements, which allows for controllable RAM inputs and observable RAM outputs. To control RAM inputs and observe RAM outputs, the core serially loads and unloads data through the newly formed truncated scan path.

The RAM scan chain ends are always available at the core periphery. The register file write enable control line is also multiplexed with WENCTEST to allow external control during RAM testing. This arrangement is shown in [Figure 11.1](#).

Figure 11.1 Register File Testing Scan Path



Serial test patterns are generated directly from the memory RAMBIST pattern. A test wrapper (for VHDL or Verilog) is provided with the core that performs the necessary data manipulation and generates the strobes needed to drive the ramtest circuitry. Test duration is minimized by simultaneous pattern loading and response unloading. The parallel RAMBIST patterns contain 270 patterns giving a simulation duration of approximately 22,000 vectors.

Chapter 12

Specifications

The maximum MCLK operating frequency and other AC timing values are dependent upon the technology used to implement the core. All AC timing parameters are listed in the *CW00100x ARM7TDMI Microprocessor Core Datasheet*, available from LSI Logic. For example, the CW001004 datasheet includes AC timing for the core implemented in LSI Logic's G10 technology.

Appendix A

ARM7TDMI Changes

This appendix describes the differences between this manual and the *ARM7TDMI Data Sheet* (ARM document number ARM DDI 0029E), but does not describe any layout changes. The purpose of this appendix is to enable a reader familiar with the ARM document to identify areas where LSI Logic's ARM7TDMI microprocessor core differs from the common ARM implementation.

- The preface and front material have been changed.
- [Chapter 1, "Introduction"](#) includes a new section that provides an overview of the LSI Logic ARM7TDMI microprocessor core.
- The logic diagram in [Chapter 1](#) has moved to [Chapter 2](#) and now also includes eight new production test signals.
- [Chapter 2](#) includes descriptions of new pins added for production test.
- Any information about transistor sizes is technology-dependent and has been removed.
- The signal descriptions of TCK1 and TCK2 have changed to reflect that the LSI Logic ARM7TDMI core implementation uses edge-sensitive logic rather than level-sensitive logic.
- Detailed information on the ARM and THUMB Instruction Sets has been removed from [Chapter 4, "ARM Instruction Set Summary"](#) and [Chapter 5, "THUMB Instruction Set Summary"](#). See the *ARM Architecture Reference Manual* for a complete description of all instructions.
- [Section 3.12, "Pipeline Architecture,"](#) was added to describe the operation of the core pipeline.
- [Chapter 8](#) does not include the section, "Clock Switch During Test," as this test mode is not required for the LSI Logic implementation.

- [Chapter 11](#) has been added to describe the LSI Logic production test implementation.

Customer Feedback

We would appreciate your feedback on this document. Please copy the following page, add your comments, and fax it to us at the number shown.

If appropriate, please also fax copies of any marked-up pages from this document.

Important: Please include your name, phone number, fax number, and company address so that we may contact you directly for clarification or additional information.

Thank you for your help in improving the quality of our documents.

Reader's Comments

Fax your comments to: LSI Logic Corporation
Technical Publications
M/S E-198
Fax: 408.433.4333

Please tell us how you rate this document: *ARM7TDMI Microprocessor Core Technical Manual*. Place a check mark in the appropriate blank for each category.

	Excellent	Good	Average	Fair	Poor
Completeness of information	_____	_____	_____	_____	_____
Clarity of information	_____	_____	_____	_____	_____
Ease of finding information	_____	_____	_____	_____	_____
Technical content	_____	_____	_____	_____	_____
Usefulness of examples and illustrations	_____	_____	_____	_____	_____
Overall manual	_____	_____	_____	_____	_____

What could we do to improve this document?

If you found errors in this document, please specify the error and page number. If appropriate, please fax a marked-up copy of the page(s).

Please complete the information below so that we may contact you directly for clarification or additional information.

Name _____ Date _____

Telephone _____ Fax _____

Title _____

Department _____ Mail Stop _____

Company Name _____

Street _____

City, State, Zip _____

U.S. Distributors by State

H. H. Hamilton Hallmark
W. E. Wyle Electronics

Alabama

Huntsville
H. H. Tel: 205.837.8700
W. E. Tel: 800.964.9953

Alaska

H. H. Tel: 800.332.8638

Arizona

Phoenix
H. H. Tel: 602.736.7000
W. E. Tel: 800.528.4040
Tucson
H. H. Tel: 520.742.0515

Arkansas

H. H. Tel: 800.327.9989

California

Irvine
H. H. Tel: 714.789.4100
W. E. Tel: 800.626.9953
Los Angeles
H. H. Tel: 818.594.0404
W. E. Tel: 800.288.9953
Sacramento
H. H. Tel: 916.632.4500
W. E. Tel: 800.627.9953
San Diego
H. H. Tel: 619.571.7540
W. E. Tel: 800.829.9953
San Jose
H. H. Tel: 408.435.3500
Santa Clara
W. E. Tel: 800.866.9953
Woodland Hills
H. H. Tel: 818.594.0404

Colorado

Denver
H. H. Tel: 303.790.1662
W. E. Tel: 800.933.9953

Connecticut

Cheshire
H. H. Tel: 203.271.5700
Wallingford
W. E. Tel: 800.605.9953

Delaware

North/South
H. H. Tel: 800.526.4812
Tel: 800.638.5988

Florida

Fort Lauderdale
H. H. Tel: 305.484.5482
W. E. Tel: 800.568.9953
Orlando
H. H. Tel: 407.657.3300
W. E. Tel: 407.740.7450
Tampa
W. E. Tel: 800.395.9953
St. Petersburg
H. H. Tel: 813.507.5000

Georgia

Atlanta
H. H. Tel: 770.623.4400
W. E. Tel: 800.876.9953

Hawaii

H. H. Tel: 800.851.2282

Idaho

H. H. Tel: 801.266.2022

Illinois

North/South
H. H. Tel: 847.797.7300
Tel: 314.291.5350
Chicago
W. E. Tel: 800.853.9953

Indiana

Indianapolis
H. H. Tel: 317.575.3500
W. E. Tel: 888.358.9953

Iowa

Cedar Rapids
H. H. Tel: 319.393.0033

Kansas

Kansas City
H. H. Tel: 913.663.7900

Kentucky

Central/Northern/ Western
H. H. Tel: 800.984.9503
Tel: 800.767.0329
Tel: 800.829.0146

Louisiana

North/South
H. H. Tel: 800.231.0253
Tel: 800.231.5575

Maine

H. H. Tel: 800.272.9255

Maryland

Baltimore
H. H. Tel: 410.720.3400
W. E. Tel: 800.863.9953

Massachusetts

Boston
H. H. Tel: 978.532.9808
W. E. Tel: 800.444.9953

Michigan

Detroit
H. H. Tel: 313.416.5800
W. E. Tel: 888.318.9953
Grandville
H. H. Tel: 616.531.0345

Minnesota

Minneapolis
H. H. Tel: 612.881.2600
W. E. Tel: 800.860.9953

Mississippi

H. H. Tel: 800.633.2918

Missouri

St. Louis
H. H. Tel: 314.291.5350

Montana

H. H. Tel: 800.526.1741

Nebraska

H. H. Tel: 800.332.4375

Nevada

Las Vegas
H. H. Tel: 800.528.8471
W. E. Tel: 702.765.7117

New Hampshire

H. H. Tel: 800.272.9255

New Jersey

North/South
H. H. Tel: 201.515.1641
Tel: 609.222.6400

Oradell

W. E. Tel: 201.261.3200
Pine Brook
W. E. Tel: 800.862.9953

New Mexico

Albuquerque
H. H. Tel: 505.293.5119

New York

Long Island
H. H. Tel: 516.434.7400
W. E. Tel: 800.861.9953
Rochester
H. H. Tel: 716.475.9130
W. E. Tel: 800.319.9953
Syracuse
H. H. Tel: 315.453.4000

North Carolina

Raleigh
H. H. Tel: 919.872.0712
W. E. Tel: 800.560.9953

North Dakota

H. H. Tel: 800.829.0116

Ohio

Cleveland
H. H. Tel: 216.498.1100
W. E. Tel: 800.763.9953
Dayton
H. H. Tel: 614.888.3313
W. E. Tel: 800.763.9953

Oklahoma

Tulsa
H. H. Tel: 918.459.6000

Oregon

Portland
H. H. Tel: 503.526.6200
W. E. Tel: 800.879.9953

Pennsylvania

Pittsburgh
H. H. Tel: 412.281.4150
Philadelphia
H. H. Tel: 800.526.4812
W. E. Tel: 800.871.9953

Rhode Island

H. H. 800.272.9255

South Carolina

H. H. Tel: 919.872.0712

South Dakota

H. H. Tel: 800.829.0116

Tennessee

East/West
H. H. Tel: 800.241.8182
Tel: 800.633.2918

Texas

Austin
H. H. Tel: 512.219.3700
W. E. Tel: 800.365.9953
Dallas
H. H. Tel: 214.553.4300
W. E. Tel: 800.955.9953
El Paso
H. H. Tel: 800.526.9238
Houston
H. H. Tel: 713.781.6100
W. E. Tel: 800.888.9953
Rio Grande Valley
H. H. Tel: 210.412.2047

Utah

Draper
W. E. Tel: 800.414.4144
Salt Lake City
H. H. Tel: 801.365.3800
W. E. Tel: 800.477.9953

Vermont

H. H. Tel: 800.272.9255

Virginia

H. H. Tel: 800.638.5988

Washington

Seattle
H. H. Tel: 206.882.7000
W. E. Tel: 800.248.9953

Wisconsin

Milwaukee
H. H. Tel: 414.513.1500
W. E. Tel: 800.867.9953

Wyoming

H. H. Tel: 800.332.9326

Sales Offices and Design Resource Centers

LSI Logic Corporation
Corporate Headquarters
Tel: 408.433.8000
Fax: 408.433.8989

NORTH AMERICA

California

Irvine
◆ Tel: 714.553.5600
Fax: 714.474.8101

San Diego

Tel: 619.613.8300
Fax: 619.613.8350

Wireless Design Center

Tel: 619.350.5560
Fax: 619.350.0171

Silicon Valley

◆ Tel: 408.433.8000
Fax: 408.954.3353

Colorado

Boulder
Tel: 303.447.3800
Fax: 303.541.0641

Florida

Boca Raton
Tel: 561.989.3236
Fax: 561.989.3237

Illinois

Schaumburg
◆ Tel: 847.995.1600
Fax: 847.995.1622

Kentucky

Bowling Green
Tel: 502.793.0010
Fax: 502.793.0040

Maryland

Bethesda
Tel: 301.897.5800
Fax: 301.897.8389

Massachusetts

Waltham
◆ Tel: 781.890.0180
Fax: 781.890.6158

Minnesota

Minneapolis
◆ Tel: 612.921.8300
Fax: 612.921.8399

New Jersey

Edison
◆ Tel: 732.549.4500
Fax: 732.549.4802

New York

New York
Tel: 716.223.8820
Fax: 716.223.8822

North Carolina

Raleigh
Tel: 919.785.4520
Fax: 919.783.8909

Oregon

Beaverton
Tel: 503.645.0589
Fax: 503.645.6612

Texas

Austin
Tel: 512.388.7294
Fax: 512.388.4171

Dallas

◆ Tel: 972.509.0350
Fax: 972.509.0349

Houston

Tel: 281.379.7800
Fax: 281.379.7818

Washington

Issaquah
Tel: 425.837.1733
Fax: 425.837.1734

Canada

Ontario

Ottawa
◆ Tel: 613.592.1263
Fax: 613.592.3253

Toronto

◆ Tel: 416.620.7400
Fax: 416.620.5005

Quebec

Montreal
◆ Tel: 514.694.2417
Fax: 514.694.2699

INTERNATIONAL

Australia

New South Wales
◆ Reptechnic Pty Ltd
◆ Tel: 612.9953.9844
Fax: 612.9953.9683

China

Beijing
◆ LSI Logic International
Services Inc
Tel: 86.10.6804.2534.40
Fax: 86.10.6804.2521

Denmark

Ballerup
◆ LSI Logic Development
Centre
Tel: 45.44.86.55.55
Fax: 45.44.86.55.56

France

Paris
◆ LSI Logic S.A.
Immeuble Europa
◆ Tel: 33.1.34.63.13.13
Fax: 33.1.34.63.13.19

Germany

Munich
◆ LSI Logic GmbH
◆ Tel: 49.89.4.58.33.0
Fax: 49.89.4.58.33.108

Stuttgart

Tel: 49.711.13.96.90
Fax: 49.711.86.61.428

Hong Kong

Hong Kong
◆ AVT Industrial Ltd
Tel: 852.2428.0008
Fax: 852.2401.2105

India

Bangalore
◆ LogiCAD India Private Ltd
◆ Tel: 91.80.526.2500
Fax: 91.80.338.6591

Israel

Ramat Hasharon
◆ LSI Logic
◆ Tel: 972.3.5.480480
Fax: 972.3.5.403747

Netanya

◆ VLSI Development Centre
Tel: 972.9.657190
Fax: 972.9.657194

Italy

Milano
◆ LSI Logic S.P.A.
◆ Tel: 39.039.687371
Fax: 39.039.6057867

Japan

Tokyo
◆ LSI Logic K.K.
◆ Tel: 81.3.5463.7821
Fax: 81.3.5463.7820

Osaka

◆ Tel: 81.6.947.5281
Fax: 81.6.947.5287

Korea

Seoul
◆ LSI Logic Corporation of
Korea Ltd
◆ Tel: 82.2.528.3400
Fax: 82.2.528.2250

The Netherlands

Eindhoven
◆ LSI Logic Europe Ltd
Tel: 31.40.265.3580
Fax: 31.40.296.2109

Singapore

Singapore
◆ LSI Logic Pte Ltd
◆ Tel: 65.334.9061
Fax: 65.334.4749

Sweden

Stockholm
◆ LSI Logic AB
◆ Tel: 46.8.444.15.00
Fax: 46.8.750.66.47

Switzerland

Brugg/Biel
◆ LSI Logic Sulzer AG
Tel: 41.32.536363
Fax: 41.32.536367

Taiwan

Taipei
◆ LSI Logic Asia-Pacific
◆ Tel: 886.2.2718.7828
Fax: 886.2.2718.8869

Avnet-Mercurius

Corporation, Ltd
Tel: 886.2.2503.1111
Fax: 886.2.2503.1449

Jeilin Technology

Corporation, Ltd
Tel: 886.2.2248.4828
Fax: 886.2.2242.4397

Lumax International

Corporation, Ltd
Tel: 886.2.2788.3656
Fax: 886.2.2788.3568

United Kingdom

Bracknell
◆ LSI Logic Europe Ltd
◆ Tel: 44.1344.426544
Fax: 44.1344.481039

◆ Sales Offices with
Design Resource Centers